Evaluation of Search-Based Algorithms for Solving *Wordle*

Nathalie Abello-Sudarsky, Sebastian Tremblay, Simone Ritcheson
Khoury College of Computer Sciences
Northeastern University, Boston, MA

[https://github.com/nathalieabello/wordle_bot]

Abstract

The New York Times' daily *Wordle* is a popular game with the objective of identifying a hidden five-letter word within six guesses. This study aims to evaluate four solver strategies for *Wordle* — Naive, Greedy, Minimax, and Monte Carlo Tree Search. The findings show that simpler, adaptive strategies like Greedy, outperform both naive approaches and complex algorithms in deterministic environments like *Wordle*. These results highlight how efficient heuristics offer the best balance of accuracy and speed, providing broader lessons on aligning algorithm design with the structure of the problem.

Introduction

Wordle is a daily word guessing game popularized by The New York Times, in which a player must discover a hidden five-letter word within six guesses. After each attempt, players receive feedback for each letter: green for a correct letter in the correct position, yellow for a correct letter in the wrong position, and gray for letters not in the word.

This project approaches *Wordle* as a single-agent search problem operating within a partially observable environment, where the agent must find the hidden target word through limited feedback. The environment is defined by a discrete state space composed of valid five-letter words and deterministic transitions that reflect the game's feedback mechanism. The primary objective is to identify the secret word within six guesses, using each round of feedback to strategically narrow down the set of plausible candidates.

Wordle is structured with only partial information provided after each guess, so the solver must make informed decisions that balance their observations (the feedback) with their knowledge of the 12,972-word state space (Kinkelin 2025). As such, this problem exemplifies constrained search under uncertainty. Each transition filters the state space based on cumulative feedback, and effective solvers must adapt their strategy as new information becomes available.

The goal of each solver is to produce a sequence of guesses that efficiently reduces this uncertainty, converging on the correct answer before exhausting the six-guess limit. Each guess must be selected not only for its potential correctness, but also for its ability to eliminate large portions of the remaining search space. This requires intelligent use of heuristics that draw on principles from information theory and probabilistic reasoning.

Beyond the game itself, developing optimal *Wordle* solvers offers insight into the design of efficient algorithms for information gathering and decision-making under constraints. The clear success metrics, such as win rate and average number of guesses, make *Wordle* an ideal testbed for evaluating various search strategies. To explore the problem space, we implemented and compared four distinct solver strategies:

- **Naive Baseline:** Selects guesses uniformly at random from the remaining valid words.
- **Greedy Search:** Chooses the word expected to yield the greatest immediate entropy reduction.
- **Minimax Search:** Selects the guess that minimizes the size of the largest remaining candidate set across all possible feedback patterns.
- **Monte Carlo Tree Search (MCTS):** Approximates optimal play by simulating many possible future guess–feedback sequences and propagating outcomes.

Each of these methods represents a different trade-off between computational complexity and search efficiency. Implementing varied approaches in these methods allows for a diverse evaluation of algorithmic strategies under the constrained and uncertain conditions imposed by *Wordle*.

Background

To understand the approach to solving *Wordle*, it's helpful to look at the basic ideas behind each of the four algorithms that were implemented. Each solver we implemented builds on different strategies for making those decisions.

The Naive solver is the simplest of the four algorithms implemented. The Naive Bayes Algorithm is a probabilistic classifier based on Bayes' theorem, assuming strong independence assumptions between features. It is a simple algorithm that served as a baseline for algorithmic evaluations. The algorithm is called "naive" due to the relationships between features; it assumes that the presence or absence of a feature is unrelated to the presence or absence of any other feature. While this assumption may work mathematically or conceptually, it is often not true in real-world situations.

The Greedy approach tries to be smarter by picking the guess that gives the most information right away. It does this

by looking at how much each possible guess would reduce uncertainty. Specifically, how it would shrink the list of possible words after seeing the different types of feedback. This is based on a concept from information theory called entropy. Entropy of a random variable quantifies the average level of uncertainty or information associated with the variable's potential states or possible outcomes. The word with the highest expected information gain is selected.

The Minimax algorithm takes a more cautious, worst-case approach. Instead of aiming for the average best result, it looks at the worst feedback it could receive for each guess and tries to minimize the size of the remaining word list in that case. This helps avoid getting stuck with large lists of possibilities after a bad guess. It's more conservative, but it's also very computationally expensive since it has to consider every possible feedback scenario for each word.

Monte Carlo is an algorithm that uses random simulations to explore future guesses to understand which paths are most promising. Instead of checking every outcome like Minimax or Greedy, it builds a tree of possible game states and uses sampling to guide its search. It balances trying new guesses (exploration) with sticking to guesses that have worked well in the past (exploitation). Over time, the tree helps the algorithm make more informed decisions. At low depths, Monte Carlo offers a great trade-off between accuracy and runtime while maintaining the ability to improve performance with higher depths and compute costs. As such, Monte Carlo is a strong and practical option for *Wordle*.

Related Work

There are a few other methods that could be used to solve *Wordle*, each with its own trade-offs. Reinforcement learning is one option; it could learn a strategy over time by being rewarded for better guesses. But for a game as short and constrained as *Wordle*, with the small state space, the complexity of applying a very large-scale learning model could be disproportionate to the problem size. Approaches like reinforcement learning would need extensive training data to truly operate and generalize the problem well (Yang 2024), which is difficult to obtain due to the small state space and the lack of diverse game courses.

More generally, traditional machine learning could be applied, like training a model on previous games to predict good guesses. However, traditional machine learning methods struggle with generalization, and in this case, due to the limited and specific nature of *Wordle* and its possible answers, it would most likely do the same. The small number of these answers, $\sim$2,000, limits the diversity of training examples.

Since optimal guessing often relies on information principles, like entropy, rule-based or search-based methods are often more effective and interpretable for this game. Another common method is using frequency-based heuristics, where guesses are chosen based on how often certain letters appear in the English language. While simple and often effective for early rounds, they do not adapt well once you start getting feedback. While this method can be effective for early guesses, it doesn't adapt well once feedback is received, as it doesn't account for the positional information provided by the game's responses (Zaiontz 2025).

In the paper Search *Algorithms for Mastermind* (Rhodes 2019), Rhodes uses Simulated Annealing (SA) and Maximum Expected Reduction in Consistency (MERC) to solve the Mastermind game. Similarly to Wordle, it focuses on maximizing information gain to efficiently narrow down the set of possible codes. Both games use deductive reasoning and feedback-based guessing; using the methods in Rhodes' paper can enhance the effectiveness and validity of *Wordle* strategies, especially by optimizing guess selection to maximize information gain.

Project Description

In the context of our problem, we define our dictionary as $\mathcal{D} \subseteq \Sigma^5$, such that Σ is the 26-letter English alphabet, and the hidden target word $w^* \in \mathcal{D}$ must be identified in at most six attempts.

After each guess, the solver receives constrained feedback on letter placement and correctness, making the environment partially observable and requiring strategic inference over successive rounds. The feedback received after a guess $g \in \mathcal{D}$ is a 5-tuple $f(g, w^*) \in \{\text{green}, \text{yellow}, \text{gray}\}^5$, where:

- **Green:** the letter is in the correct position.
- **Yellow:** the letter is in the wrong position, but in the word.
- **Gray:** the letter is not present in the target word.

Using this feedback, the candidate list gets narrowed down, only keeping words consistent with all the cumulative feedback received, such that $\mathcal{S} \subseteq \mathcal{D}$. The transition function is defined as follows:

$$T(\mathcal{S}, g_t, r) = \{\, w \in \mathcal{S} \mid f(g_t, w) = r \}$$

where g_t is the t-th guess and r is the resulting feedback pattern. The game ends when a guess g_j exactly matches the target word. That is, all feedback is green: $f(g_j, w^*) = (\text{green}, \text{green}, \text{green}, \text{green}, \text{green})$.

Our implementation uses two word lists — the set of target words and valid guess words:

- The target word set contains 2,315 words from the original source code (Freshman 2024), which were used as daily solutions before the *New York Times* acquisition in January 2022. We used this as our test set to get our performance statistics.
- The full guess set contains 12,972 valid five-letter English words, also taken directly from the original source code (Kinkelin 2025).

Naive Solver

The baseline Naive solver operates by filtering out invalid words after each guess based on the feedback pattern, then selecting a random word uniformly from the remaining candidate set $\mathcal{S}_t$. While simple, this approach provides a meaningful benchmark for understanding the value of more advanced strategies.

Greedy Solver

A greedy search strategy was also implemented, with the goal of selecting at each step the guess that is expected to most effectively reduce future uncertainty. Formally, for each potential guess g, the solver computes the partitioning of the current candidate set $\mathcal{S}_t$ based on all possible feedback patterns. This partitioning is defined as

$$\Pi(g) = \left\{ \mathcal{S}_t^{(r)} \mid r \in \{\text{green}, \text{yellow}, \text{gray}\}^5 \right\}$$

where

$$\mathcal{S}_t^{(r)} = \{w \in \mathcal{S}_t \mid f(g, w) = r\}$$

such that each $\mathcal{S}_t^{(r)}$ represents the subset of candidate words that would remain valid if the feedback pattern r is observed after guessing g.

To quantify the expected information gain of a guess, an entropy-based metric is used. The initial entropy of the candidate set is given by $H(\mathcal{S}_t) = \log_2 |\mathcal{S}_t|$.

For each partition $\mathcal{S}_t^{(r)}$, the probability of receiving feedback pattern r is $P(r \mid g) = \frac{|\mathcal{S}_t^{(r)}|}{|\mathcal{S}_t|}$. The expected entropy after making guess g is then calculated as $E[H \mid g] = \sum_r P(r \mid g) \cdot \log_2 |\mathcal{S}_t^{(r)}|$. The expected information gain from guess g is the reduction in entropy, defined as $H(\mathcal{S}_t) - E[H \mid g]$.

The solver selects the guess g^* that maximizes this expected information gain: $g^* = \arg\max_{g \in \mathcal{S}_t}$.

This strategy favors guesses that partition the remaining candidate set into balanced, equally sized subsets, thus maximizing the solver's progress on average across all possible feedback outcomes.

Algorithm 1: Greedy Information Gain

```
Function SelectGuess(candidates):
    best_guess ← None
    best_score ← −∞
    foreach guess in candidates do
        score ← InfoGain(guess, candidates)
        if score > best_score then
            best_score ← score
            best_guess ← guess
    return best_guess
```

Algorithm 2: Calculate Expected Information Gain

```
Function InfoGain(guess, candidates):
    initial_entropy ← log2(len(candidates))
    expected_entropy ← 0
    partitions ← DivideByFeedback(guess,
      candidates)
    foreach partition in partitions do
        p ← len(partition) / len(candidates)
        expected_entropy += p · log2(1/p)
    return initial_entropy − expected_entropy
```

Algorithm 3: Divide Candidates By Feedback

```
Function DivideByFeedback(guess,
  candidates):
    partitions ← empty dictionary
    foreach candidate in candidates do
        feedback ← GetFeedback(guess, candidate)
        AddToPartition(partitions, feedback,
          candidate)
    return partitions
```

It is important to note that this method optimizes for the expected case, as it accounts for the probability distribution over all feedback patterns.

Minimax Solver

Minimax search strategies were also explored, with a focus on depth-limited implementations to manage computational cost. It is important to note that the Minimax algorithm represented the feedback as the other "player" as the opposing agent, to go with the two-player nature of Minimax. In this approach, the solver is modeled as the "*Minimizer*", aiming to reduce the size of the worst-case remaining solution set, while the feedback mechanism is treated as an *adversarial Maximizer* that selects the feedback pattern preserving the largest possible set of candidate words. This adversarial framing allows the solver to prepare for the worst-case scenario following each guess.

At the first level of the search tree, the solver selects a candidate guess $g \in \mathcal{S}_t$. At the next level, each possible feedback pattern defines a branch, leading to a reduced candidate set $T(\mathcal{S}_t, g, r)$. The recursive Minimax value function for a guess is then defined as:

$$V(\mathcal{S}_t, g, d) = \begin{cases} \max\limits_r \left(\min\limits_{g'} V\left(T(\mathcal{S}_t, g, r), g', d-1\right) \right) & \text{if } d > 0 \\ |\mathcal{S}_t| & \text{if } d = 0 \end{cases}$$

Here, d denotes the remaining search depth. When the depth limit is reached, the evaluation function returns the size of the current candidate set as a proxy for residual uncertainty. Otherwise, the environment (modeled adversarially) selects the feedback pattern r that maximizes the resulting candidate set size, and the solver responds with the guess g' that minimizes this worst-case outcome.

To decrease run times, pruning techniques are applied during evaluation. If a candidate guess g results in a branch whose worst-case outcome already exceeds the best-known alternative, further exploration of that path is skipped. This pruning accelerates the search by eliminating suboptimal guesses early in the evaluation process.

In addition to pruning, candidates are preordered using a heuristic to estimate information gain. This technique ranks each word based on the average number of shared letters with all other candidates in $\mathcal{S}_t$:

$$\text{score}(w) = \frac{1}{|\mathcal{S}_t|} \sum_{\text{other} \in \mathcal{S}_t} |\text{set}(w) \cap \text{set}(\text{other})|$$

Words with higher scores are more likely to result in a wide variety of feedback patterns, which helps efficiently

Algorithm 4: Minimax Logic

```
Function SelectGuess(candidates, depth):
    if depth ≥ MAX_DEPTH or len(candidates) ≤ 2 then
        if len(candidates) ≤ 2 then
            return (candidates[0], len(candidates))
        else
            return (EvaluateGuesses(candidates), len(candidates))
    best_guess, best_score ← None, ∞;
    foreach guess in candidates do
        outcomes ← GetOutcomes(guess, candidates)
        if all partitions in outcomes have ≥ 1 word then
            return (guess, 1)
        worst_score ← 0
        foreach (feedback, words) in outcomes do
            if len(words) = 1 then
                score ← 1
            else
                _, score ← Minimax(words, depth + 1)
            worst_score ← max(worst_score, score)
            if worst_score ≥ best_score then
                break
        if worst_score < best_score then
            best_score ← worst_score
            best_guess ← guess
            if best_score = 1 then
                break
    return (best_guess, best_score)
```

Algorithm 5: Minimax Evaluate Guesses

```
Function EvaluateGuesses(candidates):
    best_guess, best_score ← None, ∞
    foreach guess in candidates do
        outcomes ← GetOutcomes(guess, candidates)
        worst_case ← Evaluate(outcomes)
        if worst_case < best_score then
            best_score ← worst_score
            best_guess ← guess
            if best_score = 1 then
                break
    return best_guess
```

partition the search space. Words with more unique letters are ranked higher, to avoid picking words with multiple instances of the same letter, as more diverse letter distributions likely provide more information. By selecting guesses from this preordered list, the Minimax algorithm benefits from earlier and more aggressive pruning, especially when the candidate set is large.

Monte Carlo Solver

Finally, we implemented Monte Carlo Tree Search (MCTS). MCTS represents the search space as a game tree, using simulated games to estimate the optimal traversal path. Each node in the tree represents a potential game state defined by the remaining list of candidate words $\mathcal{S}_t \subseteq \Sigma^5$ while each edge corresponds to transitions based on making a specific guess and receiving a particular feedback pattern. The search process comprises of four steps: (1) Selection, (2) Expansion, (3) Simulation, and (4) Backpropagation.

MCTS begins with the selection step, starting at the root node, which represents the current candidate set $\mathcal{S}_t$. The algorithm iteratively traverses the tree by selecting the child node n that maximizes the Upper Confidence Bound 1 (UCB1) score:

$$\mathrm{UCB}(n) = \frac{v(n)}{N(n)} + C\sqrt{\frac{\ln N\big(\mathrm{parent}(n)\big)}{N(n)}},$$

where $v(n)$ is the node's total accumulated reward, $N(n)$ is its visit count, $N(\mathrm{parent}(n))$ is the parent's visit count, and C is the tunable exploration constant. The first term promotes exploitation by favoring nodes with higher average value, indicating better past outcomes. In contrast, the second term encourages exploration of less-visited nodes. This selection phase continues until a node is reached that has not been fully expanded (i.e., it has potential guesses from its $\mathcal{S}_t$ that have not yet led to child nodes).

At this leaf node, the *expansion* phase occurs. First, a guess g is chosen from the node's untried moves using the heuristic preordering detailed in the Minimax Solver. This is done by selecting the word with the highest heuristic value of the remaining candidates. Then, a random target word $w^\star$ is sampled from the top 15 most likely words. The feedback $r = f(g, w^\star)$ is computed, and a new child node is created, representing the filtered candidate set $T(\mathcal{S}_t, g, r)$, and added to the tree.

From the newly expanded child node, a full game simulation is performed to estimate the value of reaching this state. The simulated game-play is initialized with the candidate set and the number of remaining guesses ($G - k$, where G is the maximum guesses and k is the depth of the new node). The simulation, importantly, uses the same target word $w^\star$ sampled during the previous expansion phase. Each guess within the simulation is chosen using the heuristic ordering outlined above. Using the same target word for the simulations is essential since the feedback generated for the node edge represents the current game state.

Once the simulation finishes by successfully guessing $w^\star$ or hitting the guess limit, it returns a reward value R:

$$R(w^\star) = \begin{cases} \mu\left(1 - \dfrac{\ell}{G}\right) & \text{if } w^\star \text{ is found in } \ell \le G \text{ guesses,} \\ -\mu & \text{otherwise} \end{cases}$$

where μ is a tunable scaling factor. This reward structure incentivizes solutions found with fewer guess, while simulations that fail to solve the puzzle within the guess limit

Algorithm 6: MCTS Select Guess

```
Function SelectGuess(candidates):
    if first move then
        return "arose"
    root ← NewNode(candidates)
    for sim ← 1 to MAX_SIMULATIONS do
        current ← root
        guesses ← 0
        target ← random top 15 word by info gain
        while current has no untried moves and has children do
            current ← child with highest UCB1
            guesses++
        if current has untried moves then
            guess ← best word from untried moves
            remove guess from untried
            guesses++
            feedback ← WordleFeedback(guess, target)
            current ← AddChild(current, feedback)
        reward ← Simulate(current, target, guesses)
        while current ≠ null do
            current.visits++
            current.value += reward
            current ← current.parent
    return child of root with most visits
```

Algorithm 7: MCTS Simulation

```
Function Simulate(node, target, guesses):
    if no candidates in node then
        return 0.0
    remaining ← MAX_GUESSES - guesses
    game ← NewWordle(target, remaining)
    while game not over do
        guess ← SelectBestWord(game.candidates, ordered_words)
        if guess is None then
            return −1× REWARD_MULTIPLIER
        game.submit(guess)
    if game won then
        reward ← (1 - guesses_used / max_guesses)
    else
        reward ← 0
    return reward × REWARD_MULTIPLIER
```

receive a large negative reward. By prioritizing faster solutions, the MCTS algorithm biases the search tree toward more promising guesses.

The reward R obtained from the simulation is then propagated back up the tree along the exact path taken during the selection and expansion phases. For each node n on this path — from the simulated node back to the root — its visit count $N(n)$ is incremented, and the reward R is added to its accumulated value $v(n)$. This updates the statistics used for future UCB1 calculations.

After executing the configured number of simulations, the solver selects the guess to play in the real game. This is determined by choosing the guess g associated with the immediate child of the root node that has the highest visit count $N(n)$. This selection process prioritizes the most explored path, which is often more reliable than selecting based on the highest average reward, especially with limited simulation budgets.

Experiments

The four metrics this project focused on to evaluate our solvers were win rate, average number of guesses, time taken, and guess distribution, each respectively over 2,315 games. Looking at these four metrics showed a holistic view of how each solver was operating. In the first implementation, the algorithms failed to eliminate certain candidate words correctly when letters appeared in the target word but in incorrect positions. Specifically, words like "CARTS" remained in the candidate list even after a guess of "CRATE" with a yellow "C". Once this logic was corrected, the performance of all solvers improved a lot.

Table 1: Solver Performance Comparison

Solver	Win Rate	Avg. Guesses	Runtime (s)
Naive	90.8%	4.69	8.91
Greedy	95.9%	4.25	22.24
Minimax	94.0%	4.54	1203.03
MCTS	92.4%	4.54	2388.41

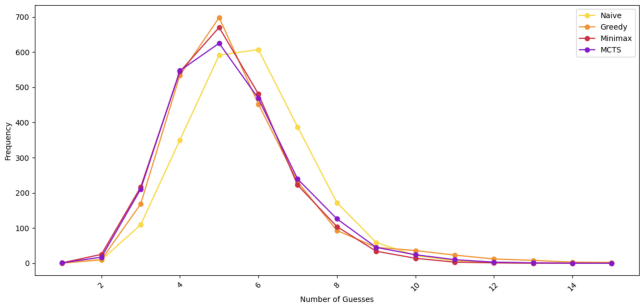


Figure 1: Guess Distribution by Solver

Naive Search

The Naive solver was purely meant to be used as a baseline for comparison to evaluate how well the other solvers performed. However, it performed rather well despite its simplicity. It achieved a 90.8% win rate and required an average of 4.69 guesses. The bug fix was still evident despite the simplicity of the algorithm, but its inherent limitations prevented it from competing with more advanced strategies.

Greedy Search

The Greedy solver showed to be the most effective and practical strategy in our project findings, with a 95.9% win

rate, an average of 4.25 guesses per game, and a runtime of just 22 seconds for 2,315 games.

We hypothesize that its success is due to the simple and precise approach, as it selects the guess that most efficiently reduces the possible word space based on current feedback. This makes it particularly well suited to *Wordle*, where players receive no prior information about which feedback patterns are more likely, and each puzzle is selected at random. Unlike Minimax or Monte Carlo Tree Search (MCTS), which take into account and plan around unknown future states, Greedy stays entirely in the moment, which maximizes immediate information gain with each guess. Focusing on the present state makes sense given the structure of the game; not knowing future outcomes and trying to optimize for worst-case scenarios or expected paths can lead to unnecessary complexity.

With that said, Greedy does occasionally fall short on particularly difficult words, cases that require 9 or more guesses, where Minimax and MCTS are better at minimizing late-game uncertainty. However, because the win rate penalizes any result past six guesses equally, and the average guess count can be skewed by the sheer number of fast wins Greedy racks up, its occasional failures do not significantly hurt its overall performance metrics. In short, while not always the most cautious approach, Greedy's aggressive reduction of the search space makes it a highly efficient and effective solver for most *Wordle* games.

Minimax Search

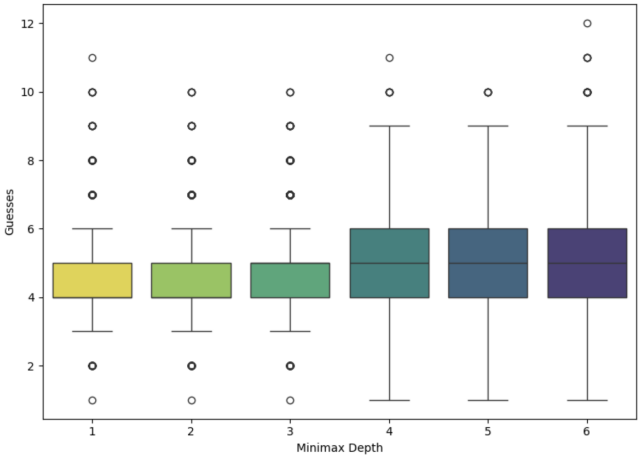
Minimax solvers, tested at depths 1 to 6 (see Figure 2), showed a consistent decline in performance as depth increased. At a depth of one, Minimax achieved its highest win rate of 94.4% and the lowest average guess count of 4.48. However, this base case does not involve true tree search as it effectively performs a one-ply worst-case optimization rather than branching gameplay possibilities. For this reason, we treat depth two as the shallowest genuine instance of Minimax search and use it as the baseline for the remainder of our analysis.

Contrary to our initial intuition, increasing search depth did not improve results. Win rates fell to 87.2% by depth six, average guesses increased to 4.92, and runtimes increased drastically, rising from around 24 minutes at depth one to almost 39 minutes at depth six.

These results reveal that shallow Minimax searches (depths 2–3) strike a strong balance between reasoning about worst-case outcomes and avoiding unnecessary complexity. Deeper searches resulted in diminishing returns. Due to Minimax's assumption of an adversarial "Maximizer", the maximizing step anticipates an opponent selecting the most challenging feedback in response to each guess. But in reality, this is just a modeling of the game, but the true nature of the game's feedback is solely determined by the fixed hidden word.

Treating this process as adversarial leads to overfitting to a false assumption, and as a result, deeper searches are overly pessimistic. The solver wastes computational effort analyzing branches of the game tree that may never happen, introducing noise rather than clarity into its decision-making. Also, deeper searches disproportionately amplifies the effect of feedback patterns where many similar candidates share feedback paths (e.g., "cater" or "bulky"). These edge cases become increasingly problematic at greater depth, making the solver converge more slowly.

Figure 2: Guess Distribution by Minimax Depth



In short, while deeper Minimax search may seem theoretically appealing, its assumptions break down in a fixed-answer game like *Wordle*. The result is slower performance, reduced accuracy, and overfitting to hypothetical adversaries that do not exist.

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) solvers displayed a plateauing effect across rollout counts. Rollout settings from 10 to 500 consistently produced win rates from 92.0% to 92.4%, with average guesses hovering at approximately 4.54. Increasing the number of simulations led to longer runtimes, from just under three minutes at 10 rollouts to over four hours at 500, yet yielded negligible improvement in win rate or guess efficiency. This behavior highlights a key characteristic of MCTS in deterministic environments like *Wordle*. Unlike stochastic or adversarial games, where higher rollout counts allow MCTS to better estimate the value of uncertain paths, *Wordle*'s fixed answer structure offers limited payoff for additional simulations once an adequate estimate is reached. After an initial set of rollouts provides a reasonable approximation of the search space, further simulations offer minimal incremental information, resulting in the observed performance plateau.

Hyperparameter Optimization

Three key parameters significantly influence the Monte Carlo Tree Search solving process and the quality of the selected guesses: (1) the number of simulations per node, (2) the exploration constant in the UCB1 formula, and (3) the reward multiplier. To balance computational cost with solution accuracy, we experimented with different combinations of these parameters to minimize the average number of guesses.

Table 2: MCTS Depth Performance Comparison

Depth	Win Rate	Avg. Guesses	Runtime (s)
10	92.1%	4.54	167.89
25	92.1%	4.54	461.05
50	92.0%	4.55	1060.99
100	92.2%	4.54	2466.53
250	92.1%	4.55	7377.94
500	92.4%	4.54	15802.79

To determine the optimal hyperparameters, we leveraged the hyperparameter optimization framework Optuna. We configured Optuna with a Tree-Structured Parzen Estimator (TPE) sampler due to its efficiency within hybrid state spaces containing both continuous (reward multiplier, exploration constant) and discrete (simulation depth) values. We additionally leveraged a median pruner to stop trials preemptively if their intermediate performance was significantly worse than the median performance of prior trials at a similar stage.

We hypothesized that higher simulation counts would correlate with lower guess counts due to their increased likelihood to converge on an optimal value. We predicted that a higher exploration constant would lead to better results due to the unpredictable and irreproducible structure of the trials. We did not have a prior belief of how the reward multiplier would impact the model's performance.

We ran the optimization process for 50 trials to minimize the average number of guesses over the 100-word test set. The optimization study successfully identified parameter sets with better performance than the initial configurations, which had an average guess count of 4.54.

Table 3: Best MCTS Trials

Trial	Avg. Guesses	# Simulations	Exploration Constant	Reward Multiplier
14	4.36	124	0.332	0.639
37	4.36	149	1.472	3.391
9	4.38	50	1.714	4.970
5	4.38	173	1.680	1.386
11	4.38	171	1.387	1.632
22	4.38	192	1.920	1.166
28	4.38	126	1.179	1.181
38	4.38	161	1.425	1.541

One of the most interesting findings from the study is that the minimum average guess count (4.36) was achieved by two distinct sets of parameters (Trial 14 and Trial 37). The parameters for Trial 14 (Simulations: 124, Exploration: 0.332, Reward: 0.639) and Trial 37 (Simulations: 149, Exploration: 1.472, Reward: 3.391) are quite different, particularly in their exploration constant and reward multiplier values. This suggests that the performance landscape might have multiple optima or relatively flat regions where different combinations of parameters can achieve similar performance.

The hypothesis that the performance landscape was flat was further evidenced by the fact that the most optimal guess count (4.36) was only marginally better than the results of the first trial (4.42). Once we ran the MCTS on the entire test set again, we saw a minimal performance increase to an average of 4.50 guesses and a 92.6 win rate. One possible explanation for this is that the test words we used for the optimization trials were not representative of the entire search space. Given more time and computing power, it would be beneficial to re-run the optimization study by using the entire search space for trials or a select group of words that best represent the entire state space.

Lastly, our initial predictions generally aligned with the results. The hypothesis that increasing simulations would improve performance was partially supported, as almost all of the best results occurred with simulation counts over the median. The exploration constant, however, was more distributed across the imposed range [0, 3]. Although there were outliers on either side, the majority of the best-performing trials' exploration constants centered around $\sqrt{2}$, the de facto standard for the exploration constant in UCB1 formulas.

Findings

Although Minimax and Monte Carlo Tree Search have more advanced reasoning, they don't perform as well in the context of *Wordle*. The Greedy solver consistently outperformed others across win rate, guess efficiency, and runtime, suggesting that strategies focused on immediate information gain are best suited for this game in particular. Minimax overfit to the adversarial modeling of the game that doesn't truly reflect *Wordle*'s nature, while MCTS showed diminishing returns on deeper simulations. Overall, the simplicity and efficiency of the Greedy solver made it the most effective and practical choice for this problem in particular.

In future work, it would be valuable to explore hybrid strategies that combine the fast convergence of the Greedy solver with selective depth from Minimax or targeted simulation from MCTS. Additionally, further analysis of the high-difficulty word clusters could inform specialized strategies for these edge cases, potentially pushing solver performance even closer to perfection.

Starting Words

Each solver relies on the initial feedback to guide its next move, so a well-chosen first guess could drastically shrink the search space early on for any strategy. Picking a starting guess that provides maximum information, for the average case, would help eliminate the largest number of incorrect candidates from the beginning, and take some of the strain off of the algorithms themselves.

The start words tested were derived from a list provided by the New York Times' Year in Review of players' common first guesses (New York Times, 2023). The list included "slate", "stare", "crane", "audio", "adieu", and more.

Among the deterministic solvers, Greedy and Minimax demonstrated the clearest impact. With "slate" as the initial guess, both achieved high win rates — 95.7% for Greedy and 95.1% for Minimax — along with the lowest average guess counts (4.25 and 4.43, respectively). Compared to earlier runs without start-word optimization, Minimax improved by 2.7 percentage points in win rate and dropped by 0.27 guesses on average. "Stare" and "crane" also performed competitively, with win rates just below "slate."

Table 4: Start Word Performance Comparison

Solver	First Guess	Win Rate	Avg. Guesses
Naive	slate	91.5%	4.63
Naive	stare	91.1%	4.68
Naive	crane	90.4%	4.73
Naive	adieu	89.8%	4.90
Naive	audio	88.6%	4.94
Naive	xylyl	81.4%	5.43
Greedy	slate	95.7%	4.25
Greedy	stare	95.6%	4.31
Greedy	crane	90.8%	4.27
Greedy	adieu	94.8%	4.46
Greedy	audio	94.6%	4.42
Greedy	xylyl	92.0%	4.89
Minimax	slate	95.1%	4.43
Minimax	stare	94.9%	4.48
Minimax	crane	94.8%	4.49
Minimax	adieu	92.7%	4.67
Minimax	audio	92.6%	4.73
Minimax	xylyl	89.8%	5.10
MCTS	slate	92.4%	4.54
MCTS	stare	91.7%	4.59
MCTS	crane	92.4%	4.56
MCTS	adieu	90.9%	4.75
MCTS	audio	90.9%	4.69
MCTS	xylyl	85.3%	5.19

Interestingly, vowel-heavy words like "adieu" and "audio," despite their popularity, consistently underperformed relative to the top choices. For instance, Minimax with "audio" dropped to 92.6%, and MCTS with "adieu" to 90.9%. While these differences may seem minor, a 1–2% drop corresponds to roughly 130–260 additional losses across the $\sim$13,000 target words.

The key reason behind this lies in letter frequency and positional distribution. Words like "slate," "stare," and "crane" contain a mix of the most common consonants (S, T, L, R, N) and include one or two high-frequency vowels (A and E), increasing the chance of overlapping with both letters and their correct positions. These words strike a balance between breadth and depth of information: they test a variety of letters while avoiding overloading the guess with low-impact ones.

In contrast, while "adieu" and "audio" cover more vowels, they lack common consonants, reducing the chance of early green or yellow feedback. Vowels are valuable, but most target words are not vowel-dense; consonants carry more discriminative power when pruning candidates. These vowel-heavy openers tend to result in broader but less useful feedback, which the solver then needs more guesses to refine.

The stochastic solvers, like MCTS and Naive, mirrored these trends. MCTS with "slate" reached a win rate of 92.4% with 4.54 average guesses, showing a modest improvement over unoptimized runs. Even the Naive strategy benefited, with "slate" yielding a 91.5% win rate — over 10 percentage points higher than "xylyl," which includes rare letters and repeated characters that provide little early feedback.

In summary, the effectiveness of a first guess lies in how much it partitions the solution space. Words that balance commonly occurring consonants and vowels and avoid obscure or repetitive letters are most useful across all solver types. Start word selection is not just a player superstition but a strategic decision that drastically improves solver performance.

Overall, the most effective start words balanced commonly occurring consonants (like S, L, T, R, and N) with at least one or two vowels. This combination helps partition the search space more effectively by maximizing potential feedback variation. The data clearly supports the value of choosing an informative first guess, especially in strategies that depend on efficient search and pruning.

Target Word Difficulty

To understand common challenges across games, we grouped target words as either easy or hard based on solver success. It's important to note that all solvers used "slate" as their initial guess, for fair comparison in this analysis.

- **Easy:** the word was guessed correctly in 2 to 3 guesses.
- **Hard:** the word was guessed correctly in ≥ 8 guesses.

Easy words often contained regular placements of vowels and consonants and also lacked repeated or rare letters. Words like "caste", "least", "stale", and "alert" provided valuable feedback after the first guess due to their shared letters with "slate". An interesting example was the target word "lathe". The feedback right after the initial guess reduced the solution space from $\sim$13,000 words to just four: "table", "lathe", "latke", and "telae". All solvers were able to guess the target word of "lathe" in just two guesses, primarily due to the reduction of candidate words. This shows how certain patterns can dramatically reduce the state space ($>$99.9%) when aligned with the starting word's letters and placement.

On the other hand, hard words' candidate lists converged more slowly. These words often exhibited overlapping letters and patterns with many other candidates. Words like "baker", "maker", "faker", or "hitch", "witch", and "ditch" left solvers with minimal options for high information-gain guesses. Even after most letters are discovered, solvers are left with multiple possibilities, only different by one letter. Similarly, common suffixes (-ER, -OR, -AR) and interchangeable initial letters (W-, B-, M-) don't narrow down the search tree by much, demanding deeper exploration. Other hard words like "jazzy" contained rare letters like J, Q, X, and Z. Some others like "mummy", "puppy", and "sassy" featured repeated letters, which solvers consider low information-gain guesses and don't choose to guess until they are left with no better options.

These findings suggest that difficult target words do not necessarily align with word rarity but with word structures, which sometimes do align. Primarily, the degree of ambiguity remaining after each guess and the solver's ability to pick among structurally similar alternatives affect how easy or difficult it is for a solver to guess.

Conclusion

We evaluated four *Wordle* solvers — Naive, Greedy, Minimax, and MCTS — over the full answer set of 2,315 words to assess their win rate, average guess count, and computational cost. Our key findings were:

- **Naive Baseline:** Solid win rate (90.8 %), very low runtime (9s).
- **Greedy Information-Gain:** Highest win rate (95.9 %), fastest runtime (22s), average 4.25 guesses.
- **Minimax (depth-2):** Competitive accuracy (94.0 %), but high runtime (1,203s), making it less practical.
- **MCTS:** Maxed at 92.4% win rate with diminishing returns beyond 10 rollouts; runtimes rose from 168s (10 rollouts) to over 2,466s (100 rollouts).

These experiments illustrate that in constrained environments like *Wordle*, more complex algorithms do not necessarily produce better results. Simpler, adaptive strategies that focus on maximizing information gain consistently outperformed both naive randomness and overly exhaustive exploration. Ultimately, this underscores the importance of aligning algorithmic design not only with accuracy goals but also with the computational dynamics and limitations of the problem space.

Contributions

N. Abello-Sudarsky: Minimax Search, Start Word & Target Word Difficulty Analysis

S. Tremblay: Monte Carlo Tree Search, Hyperparameter Optimization

S. Ritcheson: Naive Baseline, Greedy Search, Background, Related Work

References

Freshman, C. 2024.
Original Wordle Answers from Source Code in Alphabetical Order.
GitHub Gist.
https://gist.github.com/cfreshman/a03ef2cba789d8cf00c08f767e0fad7b

Katz, J.; and Bhatia, A. 2023.
Wordle Bot: Year in Review.
The New York Times, December 17, 2023.
https://www.nytimes.com/2023/12/17/upshot/wordle-bot-year-in-review.html

Kinkelin, J. 2025.
Wordle Competition: Combined Word List.
GitHub Repository.
https://github.com/Kinkelin/WordleCompetition/blob/main/data/official/combined_wordlist.txt

Rhodes, A. D. 2019.
Search Algorithms for Mastermind.
arXiv preprint arXiv:1908.06183, August 16, 2019.
https://arxiv.org/abs/1908.06183

Yang, K. 2024.
On Reinforcement Learning Generalization.
Medium, January 16, 2024.
https://medium.com/@kaige.yang0110/on-reinforcement-learning-generalization-99ce03774a69

Zaiontz, C. 2025.
Wordle Letter Frequency and Patterns.
Real Statistics Using Excel.
https://real-statistics.com/wordle-strategy/wordle-letter-frequency-and-patterns/