An Exact and Interpretable Solution to Wordle

Dimitris Bertsimas
Boeing Professor of Operations Research, Massachusetts Institute of Technology, Cambridge, MA 02139, dbertsim@mit.edu
Alex Paskov
Operations Research Center, Massachusetts Institute of Technology, Cambridge, MA 02139, apaskov@mit.edu

In this paper, we propose and scale a framework based on Exact Dynamic Programming to solve the game of Wordle, which has withstood many attempts to be solved by a variety of methods ranging from Reinforcement Learning to Information Theory. First, we derive a mathematical model of the game, present the resultant Bellman Equation, and outline a series of optimizations to make this approach tractable. Secondly, using Optimal Classification Trees with Hyperplanes, we present a methodology to showcase the policies in a compact yet interpretable manner – an exciting result as this allows readers to understand the policies and look for novel strategies themselves. We present experiments illuminating why some approximate methods have struggled to solve the game – and which our framework successfully circumvents due to its exact nature. We have implemented our algorithm in wordleopt.com. We show that the best starting word is SALET, the algorithm finds all hidden words in at most five guesses and the average number of guesses starting with SALET is 3.421.

Key words: Dynamic Programming, Optimal Decision Trees, Reinforcement Learning, Artificial Intelligence, Interpretability, Wordle

1. Introduction

Over the past year, the explosive popularity of Wordle has attracted the efforts of many to solve the game using a variety of methods: from Reinforcement Learning to custom mathematical frameworks, such as in [Anderson and Meyer (2022)] and [Katz and Conlen (2022)]. Yet at the same time lack of consensus on the "best" policies produced by these methods prompts interest in a framework that produces a certifiably optimal strategy. This lack of consensus has one main source: approaches to solving Wordle have always contained some degree of approximation due the computational difficulties of the task – which Lokshtanov and Subercaseaux (2022) show is NP-Hard. Ultimately, this approximation breaks any guarantee of optimality.

Exact Dynamic Programming is one such method that holds the potential to provably solve the game, as this field has seen success in a variety of domains from algorithm design to healthcare, and fundamentally solves problems without any degree of approximation [Bertsekas (2005)]. Despite this, no work exists on applying Exact Dynamic Programming to Wordle, likely because of scaling difficulties pertaining to the Curse of Dimensionality. In this paper, we present advances on all

these fronts. Namely, we propose a methodology that provably solves Wordle via Exact Dynamic Programming, and present a series of computational and theoretical results for this framework that yield an algorithm capable of tractably solving the game. Excitingly, we also extend the approach from [Bertsimas and Paskov (2022)] to compactly and interpretably portray the optimal policies to the readers.

1.1. Literature

The use of greedy heuristics, and Neural Networks combined with reinforcement learning methods, to generate a policy is so pervasive that indeed to the best of our knowledge, no prior work exists on using Exact Dynamic Programming to certifiably solve Wordle. Furthermore, among the literature that exists to have found the "best" words to guess for Wordle, there is contradiction — pointing again to the fact that no one has certifiably solved the game. For instance, [Katz and Conlen (2022)] introduces WordleBot and claims that CRANE is the best starting word for the game; however, their approach uses a combination of information theory-based metrics, a 2-step greedy algorithm, and a reduced action space. [(Bonthron 2022)], [Liu (2022)], Glaiel (2021), [Anderson and Meyer (2022)] all proceed via methods such as Neural Network-backed Reinforcement Learning, rank-one approximations with latent semantic indexing, or other heuristics, but produce disparate best first guesses such as SLATE, ROATE, or SOARE — all of which we later disprove as optimal first-moves.

Further, limited prior work exists on using decision trees to produce interpretable strategies for games. For instance, in a similar vein to this work but in a different domain, [Bertsimas and Paskov (2022)] uses Optimal Decision Trees to depict strong poker strategies in a human readable format. However, to the best of our knowledge no prior work has attempted to generate interpretable representations of Wordle policies.

1.2. Contributions and Structure

In this paper, we derive a finite-state Markov Decision Process to model the game of Wordle, present a tractable algorithm to solve the game to optimality using Dynamic Programming, and extend the methodology first outlined in Bertsimas and Paskov (2022) to Wordle to present the policy in a highly interpretable and human-readable manner. We analyze and demonstrate the efficacy of the optimal strategy discovered by Dynamic Programming. Finally, we comment on why prior application of Reinforcement Learning to this problem has struggled to produce an optimal policy.

The main contributions of this paper are:

1. Derivation of a finite-state Markov Decision Process (MDP) for the game of Wordle, and presentation of a tractable algorithm to solve the MDP for the optimal policy of the game, using Dynamic Programming. To our knowledge, this is the first methodology that certifiably produces

a globally optimal policy for Wordle. We show that the best starting word is SALET, it finds all hidden words in at most 5 guesses and the average number of guesses starting with SALET is 3.421.

2. A feature representation and methodology to produce interpretable Decision Trees demonstrating the optimal policy in a human-readable fashion. As a result of this, humans have the opportunity to observe intelligent strategies from the trees and directly improve their ability to play Wordle.
3. Production of a website – wordleopt.com – presenting the framework and results in an interactive format, by allowing the user to query optimal guesses given a state of the game.
4. A discussion of why prior applications of Reinforcement Learning to Wordle have struggled to produce near-optimal policies.

The structure of the paper is as follows. In Section 2, we describe the game of Wordle and outline the framework of our approach – specifically, the derivation of the Markov Decision Process modeling Wordle, the Dynamic Programming algorithm to solve the game, computational optimizations making the game tractable to solve, the learning algorithm used to represent the optimal policy, and the feature representation used to train the learning algorithm. In Section 3, we present a suite of results and analysis; in particular, the best starting word to use, lessons from the optimal policy beyond the first guess, the interpretable Decision Trees portraying the optimal policy, an analysis on the strength of the optimal policy over typical human play, and a discussion of why Reinforcement Learning struggles to uncover a near-optimal policy. In Section 4, we summarize our findings and report our conclusions.

2. The Framework

In this section, we review the rules of Wordle, outline the Dynamic Programming Formulation of the game, present the Algorithm used to solve the game, and discuss the Learning Algorithm and Feature Representation used to learn the optimal policies.

2.1. Wordle

Wordle is a word-game that has exploded in popularity over the past few year, drawing in millions of players from around the world [Malik (2022)]. The player's objective is to guess a hidden, five-letter word in at most six guesses, and the player is given information after each guess they make indicating whether their guess's characters match that of the hidden word. More specifically, each character of the players guess is highlighted as green, yellow, or grey – indicating that the character is in the hidden word and in the correct position, that the character is in the hidden word but in the incorrect position, or finally that the character does not appear in the hidden word, respectively. Note, a frequently unknown subtlety arises in the case of repeated characters. Namely, if a guess contains a

particular letter k times and the hidden word contains it $n < k$ times, then the final $k - n$ tiles of that character are colored grey (indicating that the character doesn't appear more than n times), even though the character appears in the word. Ultimately, if the player does not manage to guess the hidden word after six guesses, they lose the game; otherwise, they win!

Figure 1 provides an example game of Wordle, in which the player wins the game after four guesses, as indicated by the five green tiles in the final guess. In the first guess, POKER, we see that all but the fourth tile are colored grey, precisely because the letters P, O, K, and R are not in the hidden word VIXEN; the E is colored green as it is indeed in the hidden word and occurs in the fourth spot. In the second guess and third guesses, the letters N and V are colored yellow because they are present in VIXEN but do not occur in the first and third spots, respectively.

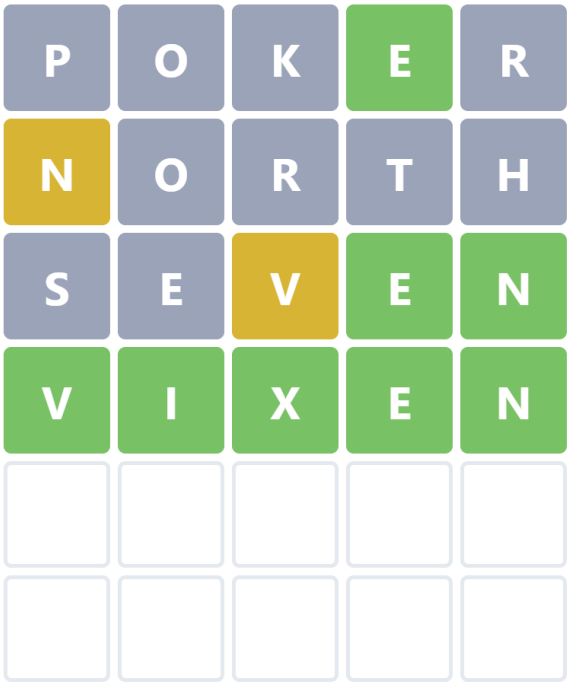


Figure 1 Example of a game of Wordle, in which the player wins after four guesses, revealing the hidden word VIXEN.

2.2. Dynamic Programming Formulation

The game of Wordle can trivially be modelled as Partially Observable Markov Decision Process (POMDP), in which the hidden information is the word we try to guess; this formulation, however, does not lead to a scalable algorithm to solve the game. In this section, we provide a formulation of the game into a tractable problem with a finite state-space, by outlining the definition of a state s, control a, cost function c, and state transitions of the underlying Markov Decision Process (MDP); we then

provide the resultant Dynamic Programming equation we use to solve the game. Note, we assume that each of the possible solutions of the game is equally likely at start (which aligns with implementations of the game), and that it is straightforward to modify the formulation to accommodate non-uniform starting probabilities.

- **State**: We define a state s_t as an unordered collection of the guesses and corresponding tile-colorings seen so far, which clearly captures all information given to the player. At the start of the game, the state is an empty collection (since no guesses have been made); as the game progresses, this collection grows with exactly the information the player sees in the game. To given an example, consider again the example game depicted in Figure 1. After the third guess, SEVEN, the state is $s_3 = \{(\text{POKER},⬛⬛⬛🟩⬛), (\text{NORTH},🟨⬛⬛⬛⬛), (\text{SEVEN},⬛⬛🟨🟩🟩)\}$.

  We provide a second definition of a state which is bijective to the first definition, yet much more computationally convenient. Namely, we define a state as a collection of words, which represents the set of all possible solutions that are valid, given the information presented to the player. Note that this clearly equivalent to the first definition and satisfies the Markov Assumption of Dynamic Programming, as knowledge of future states depends only on the current understanding of which solutions remain valid. At the start of the game, the state is a set of 2,315 words, signifying that all possible solutions are valid; as the game progresses, this set decreases in size as information is revealed. To give an example, consider that the current state at time 4 is $s_4 = \{\text{AWAKE, BLUSH, FOCAL}\}$, and the hidden word is FOCAL. If we make the guess THETA and see coloring ⬛⬛⬛⬛🟨, then from the coloring we know that the letters E and H are not present in the hidden word and thus that AWAKE and BLUSH are no longer possible solutions — transitioning us to the next state $s_5 = \{\text{FOCAL}\}$, which reveals the hidden word to us.

  Important Note: In all future discussion, we use the latter definition as it leads to more convenient and scalable algorithms for solving the game.
- **Control**: We define an action as a word in the set of all possible valid guesses A, which contains 10,657 words. Note, the number of guesses being greater than the number of solutions is a design choice of the game itself.
- **Cost Function**: We define the cost of a guess as 1, and the cost of losing the game as ∞. As such, $c_t(s,a) = 1, \forall t \leq 5$, and $c_6(s) = \infty$. This way, if a policy has finite cost, we know that it wins the game with certainty (that is, regardless of the hidden word). Additionally, this cost-structure directly allows us to minimize the expected number of guesses required to win the game, leading to the optimal policy for the game.
- **Transition Function**: For a given state s_t and guess a, denote $P(s_t, a)$ as the collection of all states we can transition into, given that we are in s_t and guess a. Note that $|P(s_t, a)| \leq 3^5$, which

is the total number of colorings of the current guess. For a fixed action a and state s_t, $P(s_t, a)$ can be constructed by taking guess a conditioned on each potential solution $w \in s_t$ being the solution, and recording what the next-state s_{t+1} is. The associated transition probabilities are computed by counting the number of times a particular next-state s_{t+1} appears while iterating through $w \in s_t$, and dividing this count by $|s_t|$. For a particular $s_{t+1} \in P(s_t, a)$, we denote this transition probability as $p_t(s_{t+1}; s_t, a)$.

Given this formulation of Wordle as a Markov Decision Process, we can apply the Bellman equation to solve for the optimal value of a state via:

$$V_t^*(s_t) = \min_{a \in A} \left\{ c_t(s_t, a) + \sum_{s_{t+1} \in P(s_t, a)} p_t(s_{t+1}; s_t, a) V_{t+1}^*(s_{t+1}) \right\},$$

which can more explicitly be written as

$$V_t^*(s_t) = \begin{cases} 1, & \text{if } t < 6, |s_t| = 1, \\ \min\limits_{a \in A} \left\{ 1 + \sum\limits_{s_{t+1} \in P(s_t, a)} p_t(s_{t+1}; s_t, a) V_{t+1}^*(s_{t+1}) \right\}, & \text{if } t < 6, |s_t| > 1, \\ \infty, & \text{if } t = 6. \end{cases} \tag{1}$$

Overview of The Algorithm

Firstly, observe that performing traditional back-propagation by enumerating all possible states at successively earlier times is computationally intractable, as the number of possible states at each time is exponential in the number of solutions — that is, $2^{2315} \approx 10^{697}$. As such, we instead solve the equation for $V_0^*(s_0)$ directly via recursion, in which s_0 is the starting state with all 2,315 solutions are present. Specifically, in evaluating $V_0^*(s_0)$, we iteratively search through the different guesses in A, and recursively solve $V_{t+1}^*(s_{t+1})$ for states that can be transitioned into. Note that while this still involves a significant amount of computation (because of the large action space and large number of possible states), a number of important computational optimizations make the problem feasible to solve, which we outline in the following section.

Finally, we note that a nice property of this algorithm is that it is trivially parallelizable. In particular, we can enumerate all possible next-states s_{t+1} from s_0 for any $a \in A$, and evaluate the value function of these next-states across many different processes.

Scaling The Algorithm

In this section, we enumerate a number of observations that lead to significant speed-ups in solving the problem.

1. Observe that $t=5$ corresponds to the situation in which the player has 1 guess left to make. If we denote s_5 as an arbitrary state at $t=5$, clearly if $|s_5|>1$ then $V_5^*(s_5)=\infty$ as there is no action that can win the game with certainty; otherwise when $|s_5|=1$, then $V_5^*(s_5)=1$ as per (1). This fact significantly reduces the amount computation needed to solve Wordle, as it effectively reduces the time-horizon of the game by 1.
2. If $|s_t|=2$ and $t<6$, then $V_t^*(s_t)=1.5$. This is because guessing either of the words in the state will win the game with 50% probability, or will transition into another state with cardinality 1, which has value 1. As such, $V_t^*(s_t)=1\times 50\%+2\times 50\%=1.5$.
3. Given a guess and a potential tile-coloring of that guess, it is straightforward to calculate whether an arbitrary solution to the game is still feasible given the information revealed by the tile-coloring. If we pre-compute this boolean for every combination of guess, coloring, and solution — resulting in a list of length $|A|\times 3^5\times 2315=5,995,042,065$ — then we can compute the transition vector $P(s_t,a)$ and associated probabilities in a number of array look-ups linear to $|s_t|$. Specifically, if we fix guess a and condition on $w\in s_t$ being the solution to the game, we produce a tile-coloring; then we check if each possible solution in s_t is feasible under this, by looking through the pre-computed booleans. Ultimately, this pre-computation produces a very efficient methodology to calculate transitions.
4. When solving for $V_t^*(s_t)$ in Eq. (1), as we iterate through actions $a\in A$, we found it extremely beneficial to prune sub-optimal actions early, before evaluating each term in the summation. To do so, first observe that for an arbitrary state s_t, a lowerbound on $V_t^*(s_t)$ is given by

$$LB(s_t)=1\times\frac{1}{|s_t|}+2\times\frac{|s_t|-1}{|s_t|}=\frac{2|s_t|-1}{|s_t|}.$$

Intuitively, this says that the best we can hope for is to guess the correct word in the state on the first guess with probability $\frac{1}{|s_t|}$, and otherwise only needing one more guess to win, with probability $\frac{|s_t|-1}{|s_t|}$ (which corresponds to the first guess having been incorrect, but determining the correct word with certainty). To perform pruning, for a given guess a, we apply this lowerbound to $V_{t+1}^*(s_{t+1})$ for every next-state $s_{t+1}\in P(s_t,a)$. And as we progressively solve the different $V_{t+1}^*(s_{t+1})$ exactly, we update the summation in (1) with the exact values. If the lower-bound on the objective ever exceeds the best guess seen thus far, we prune guess a and proceed to the next guess in A.

Ultimately, we found that this pruning scheme can quickly filter out a strong majority of guesses in A, effectively reducing the large action space. We also found it helpful to exactly solve $V_{t+1}^*(s_{t+1})$ for next-states with the largest transition probabilities first, as they most significantly increase the summation.

5. Finally, after solving a large number of states, we found that the optimal guess to make in a state is often a possible solution to the state itself; that is, if a^* is the $argmin$ for a given state s_t in Eq. (1), a majority of the time we observe that $a^* \in s_t$. As such, while iterating through actions, we first evaluate those that lie in s_t first.

The Algorithm

In this subsection, we provide descriptions of the algorithm used to generate the transition function and corresponding probabilities, as well as the algorithm used to calculate $V_t^*(s_t)$ with the aforementioned optimizations.

Algorithm 1 Get Transition Information

Input: State s_t, String action
Output: Dictionary[Set→Float] next_states
transition_info ← Dict()
for solution ∈ s_t **do**
  tile_coloring ← get_coloring(action, solution) ▷ Coloring from action given solution
  s_{temp} ← Set()
  for temp_solution ∈ s_t **do**
    if is_valid_solution(action, tile_coloring, temp_solution) **then**
      s_{temp}.add(solution)
    end if
  end for
  transition_info[s_{temp}] += $\frac{1}{|s_t|}$
end for
return transition_info

We conclude this section by discussing the limits of feasibility. The current form of Wordle — with 6 rounds, 5 letter words, and a guess and solution space of sizes 10,657 and 2,315, respectively — took days to solve via an efficient C++ implementation of the algorithm, parallelized across a 64-core computer. From inspection of (1), we see that scalability is primarily dictated by the number of rounds, vocabulary size, and word length. Thus, for instance, if we were tasked with solving a variant of Wordle with 7 rounds, the current form of the algorithm would likely not scale (because of the exponential blow-up of (1)) and thus further optimizations would need to be implemented (such as constraining the policy to 5 or 6 rounds, or proving time-independence of the value function, which we do not do or use in this paper). However, increasing the vocabulary size within moderation would

still result in a solvable game, as optimizations 3, 4, and 5 above help avoid large increases in the computation needed by pruning sub-optimal guesses and solutions.

2.3. The Learning Algorithm: Optimal Classification Trees with Hyperplanes

For decades, Classification and Regression Trees (CART) was the leading method of creating decision trees, which are among the most interpretable machine learning methods; see Breiman et al. (1984). One major shortcoming of this algorithm, however, is that the decision tree is generated in a top-down manner, and thus lacks any guarantees of optimality as splits in the tree are generated without knowledge of deeper splits. Indeed, as demonstrated in Bertsimas and Dunn (2019), one can create simple situations in which the CART algorithm fails to suitably learn from data, simply because of the greedy-nature of the algorithm. In Bertsimas and Dunn (2017) (see also Bertsimas and Dunn (2019)), Optimal Classification Trees (OCT) were developed to improve upon the CART algorithm, by bridging the gap between the greedy nature of CART, and globally optimal decision trees. Specifically, this learning algorithm uses Mixed-Integer Optimization to generate the entire decision tree in a single step — thus allowing each split in the tree to be determined with full knowledge and consideration of the other splits in the tree. As a result of this, OCT generates decision trees that strongly outperform the CART method, and in-fact are comparable to the performance of state-of-the-art methods such as Random Forest and Boosted Trees.

Expanding upon OCT, Optimal Classification Trees with Hyperplanes (OCT-H) were developed to allow a split within the decision tree to consider linear combinations of features, rather than a single feature. This flexibility often allows the method to generate shorter decision trees and trees with higher performance than OCT. This high performance and high interpretability of OCT-H will be vital for displaying the optimal policies for Wordle in a succinct yet human-understandable manner. Indeed, it is this combination of performance and interpretability that led us to using this method, over other non-interpretable yet highly performant methods such as Neural Networks or Boosted Trees.

In Figure 2, we provide a demonstration of how we utilize OCT-H to convey policies. Specifically, each node in the decision tree asks a simple question pertaining to the coloring of the guess. The green, yellow, or gray tiles indicate whether such a coloring was seen at the corresponding position illustrated for the first guess the player made; white tiles indicate that the color at that position is irrelevant. For instance, the question posed by the root (or top) node in the tree asks whether the player sees a yellow tile in the first position or a green tile in the second position. Note that because the feature vector $\boldsymbol{x}$ has 0-1 coordinates, a hyperplane $\boldsymbol{a}^T\boldsymbol{x} \leq b$ is logically equivalent to an OR among different conditions; if the answer is yes to either, we proceed to the right in the tree and see that ROUND is the best word to guess, or otherwise we proceed to the left and are faced with another

question. Ultimately, we would traverse down the tree until we reach a leaf node which corresponds to a guess to make.

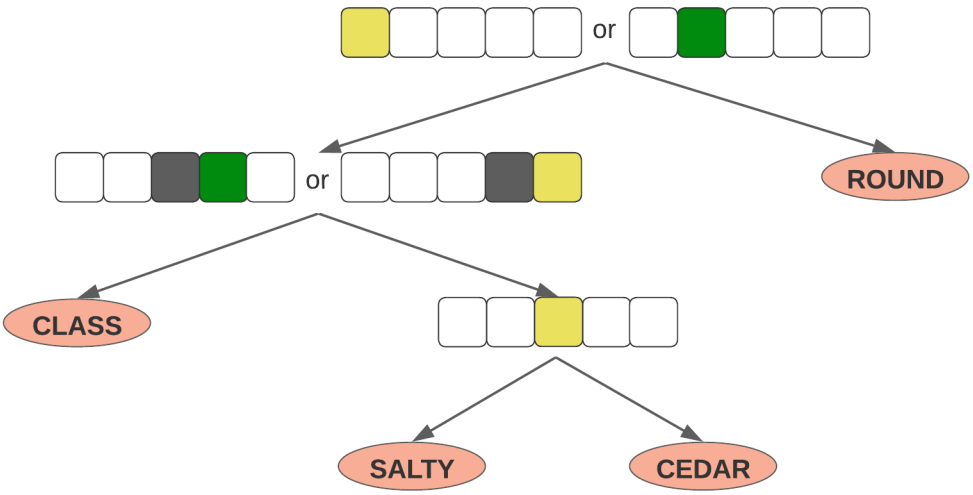


Figure 2 Figure illustrating the interpretability of a sample Wordle policy as conveyed by an Optimal Classification Tree with Hyperplanes.

Finally, we note a trade-off in using OCT and OCT-H, namely that of training time. For reference, the CART method can generate decision trees very quickly due to its greedy, top-down nature; meanwhile, training an OCT or OCT-H can be time consuming, taking up to several hours in some cases.

2.4. Interpretable Feature Representation

Since we seek to produce an interpretable decision tree portraying an optimal policy, the features we feed into the tree themselves must be interpretable. As such, the choice of feature representation is critical both because it must be rich enough to learn the policy for Wordle, while also being interpretable.

To create a vector of features from a game-state, we use a simple one-hot encoding of the colorings seen so far in the game-state. Specifically, we assign 15 features to each guess made so far and the corresponding tile-coloring. The first 3 features are binary 0/1, denoting whether the first tile is green, yellow, or gray, respectively; we then repeat this for the remaining 4 colors of the guess, assigning features 4-6 to the second color, 7-9 to the third color, and so on. To give an example, consider the situation depicted in Figure 2, in which the player has made 2 guesses — SERVE, and then TENSE. The coloring of the first guess is transformed into the first 15 binary numbers in the Figure,

as instructed above, and the next guess is transformed into the following 15 features. Ultimately, then, the description of a game-state with n guesses yields a feature representation with $15n$ features.

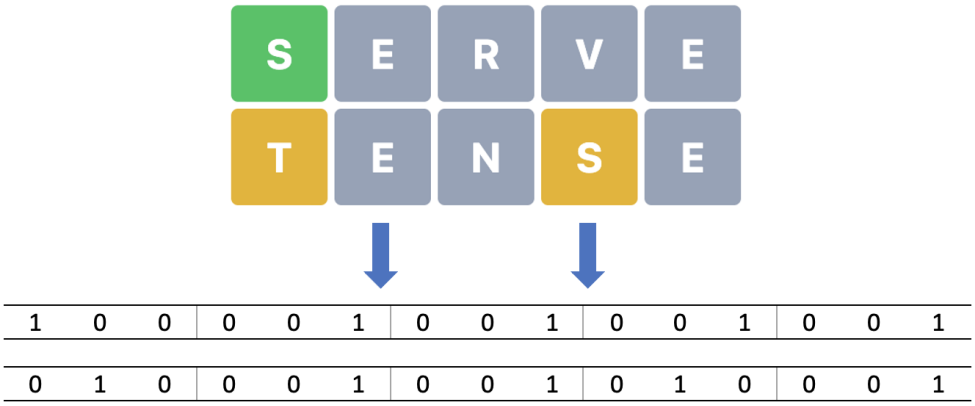


Figure 3 Figure depicting the interpretable feature representation from an example game-state. Pale lines are drawn after every third feature to emphasize the correspondence between a colored tile and its corresponding 3 binary features.

3. Results and Discussion

In this section, we present a detailed analysis of the optimal policy and discuss results of why prior heuristics have struggled to uncover the optimal policy.

3.1. The Best Starting Word

A common question for Wordle is what the best first guess is. Table 1 answers this by presenting the five best initial guesses to take in Wordle. From the Table, we see that SALET is the best first-guess, yielding an expected number of guesses of 3.421, in which this expectation assigns equal probability to each of the 2315 solutions being the hidden word. Starting with SALET, the algorithm finds every hidden word in at most five guesses.

Interestingly, from these top-five options we see that it is highly valuable to have the letters A, E, and T in your initial guess, because all five of the words contain these letters; including an S, L, or R appear to also be of high importance as these letters appear in most of the top words. As such, including these letters in an initial guess appears to yield the best results, as opposed to other common strategies such as including as many vowels as possible in the first guess. Indeed, to corroborate this, note that guessing SALET reveals an average of 1.683 colored tiles in the first move, whereas a word such as AUDIO containing 4 vowels (the most a single, five-letter word can contain) reveals only an average of 1.320 colored tiles in the first move.

Table 1 Table listing the five initial guesses in Wordle yielding the lowest expected number of guesses to win, if an optimal policy is followed.

Initial Guess	Expected Number of Guesses
SALET	**3.42117**
REAST	3.42246
TRACE	3.42376
CRATE	3.42376
SLATE	3.42462

3.2. The Optimal Policy Beyond the First Guess

Figure 4 demonstrates the efficacy with which the optimal policy plays Wordle. More specifically, it shows at most how many guesses are needed to win the game, conditioned on each of the 2315 possible solutions to the game. For instance, after just two guesses, the optimal policy will have won the game 4.0% of the time, and after just one more guess, that probability increases to 57.1%. Interestingly, we also see that the optimal policy will never need to use all six guesses, as it wins the game with certainty in at most five guesses – with just 3.3% of situations requiring that final, fifth guess. This fact — that not all rounds are even necessary to win the game with certainty — highlights the advantage of optimality, as some prior heuristic approaches (such as Bonthron (2022)) produce policies that don't even win the game with certainty despite utilizing all 6 guesses; see Section 3.5 for further discussion on the trouble of applying heuristics to Wordle.

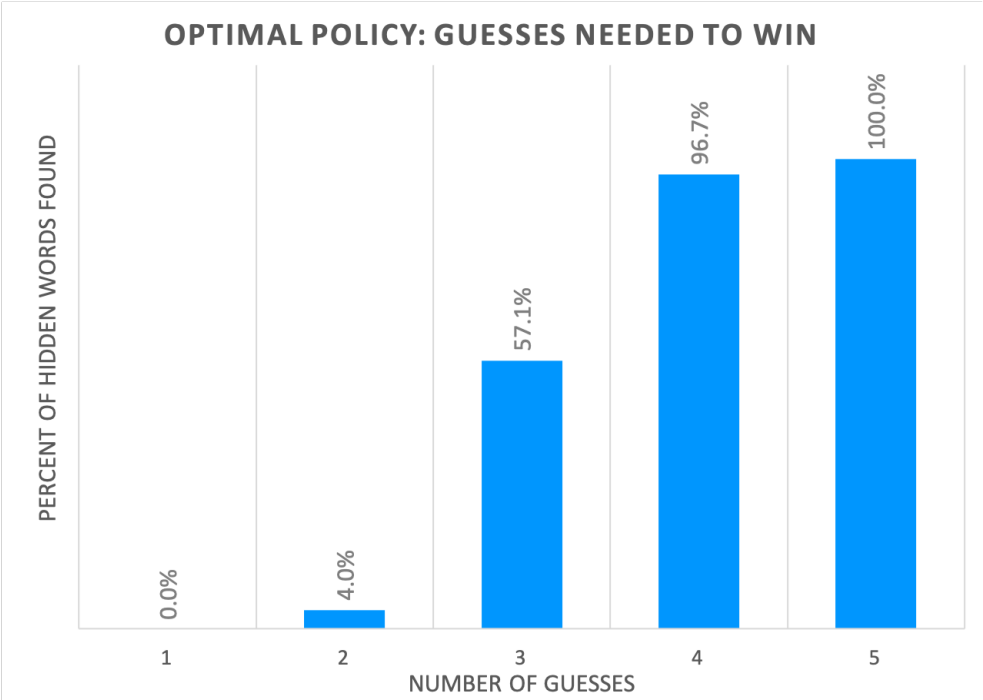
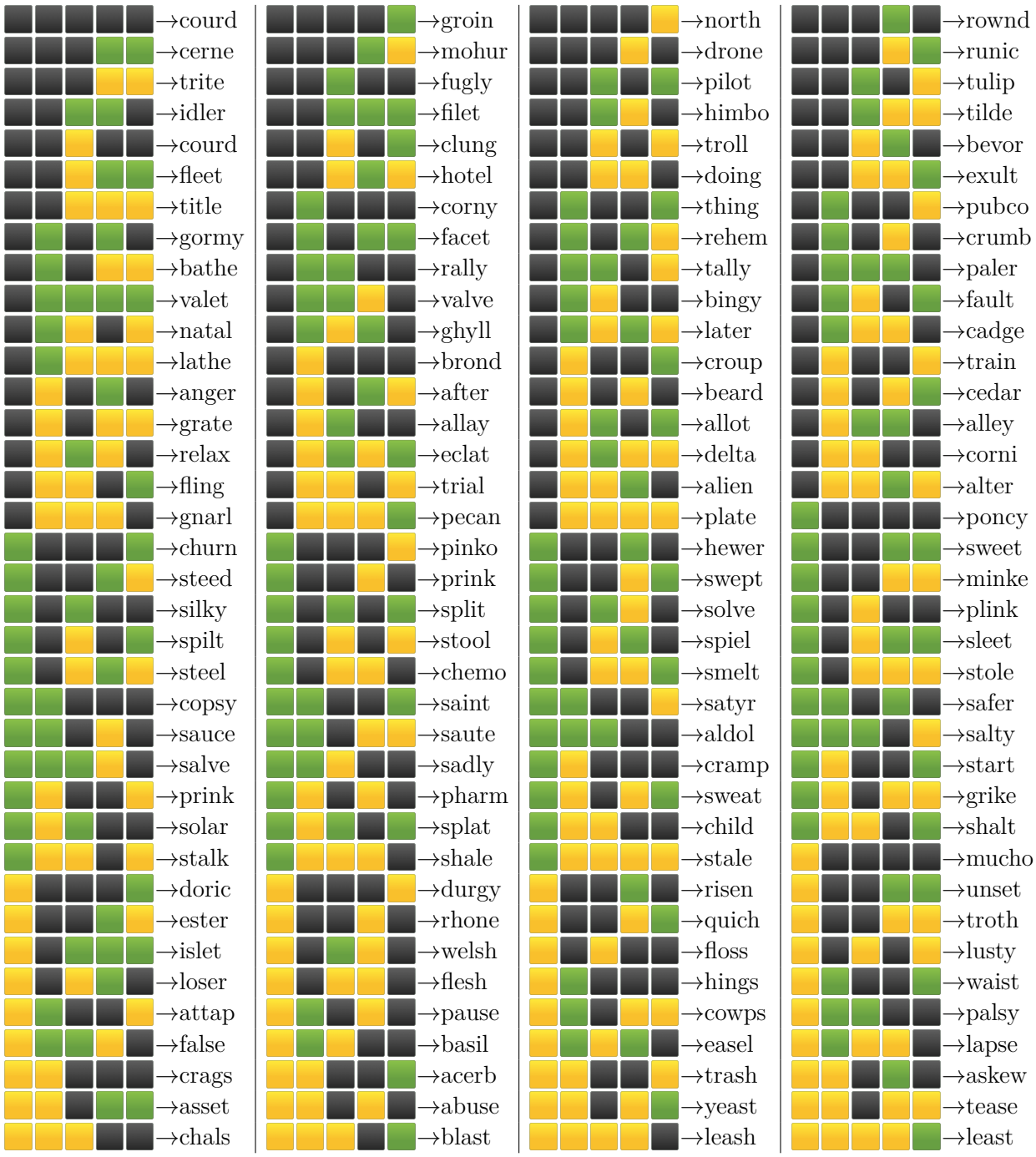


Figure 4 Graph demonstrating the number of guesses needed to win the game under the optimal policy, conditioned on each of the possible hidden words.

3.3. Interpretable Representation of The Optimal Strategy

In this section, we present and discuss one of the most exciting aspects of our work: the interpretable decision trees portraying an optimal policy for Wordle. As discussed previously, we emphasize the novelty of using such trees to portray the policy, and the benefits that optimal trees and hyperplanes bring.

Before showing the trees, we preface this section by acknowledging that using a decision tree to display and understand the optimal policy may not be as crucial for the second guess, as a simple lookup table for the different cases can be created, as we portray below. However, a lookup table for the third, fourth, or fifth guess is extremely dense and takes up tens of pages to depict the large number of situations. In contrast, our decision trees are all approximately the same depth (depths 10, 10, and 7 for guesses 3, 4, and 5, respectively) and hence still maintain high readability and, importantly, interpretability.



In Figure 5, we visualize a section of the OCT-H for the second guess of the optimal strategy (which requires that SALET was the first word guessed, as it is the best starting word). Note that nodes containing a box with "..." inside are truncated sections of the tree, which is not depicted in its entirety due to space limitations of the page. Also, see Section 2.3 for specifics on how to traverse through the tree, and an example on doing this.

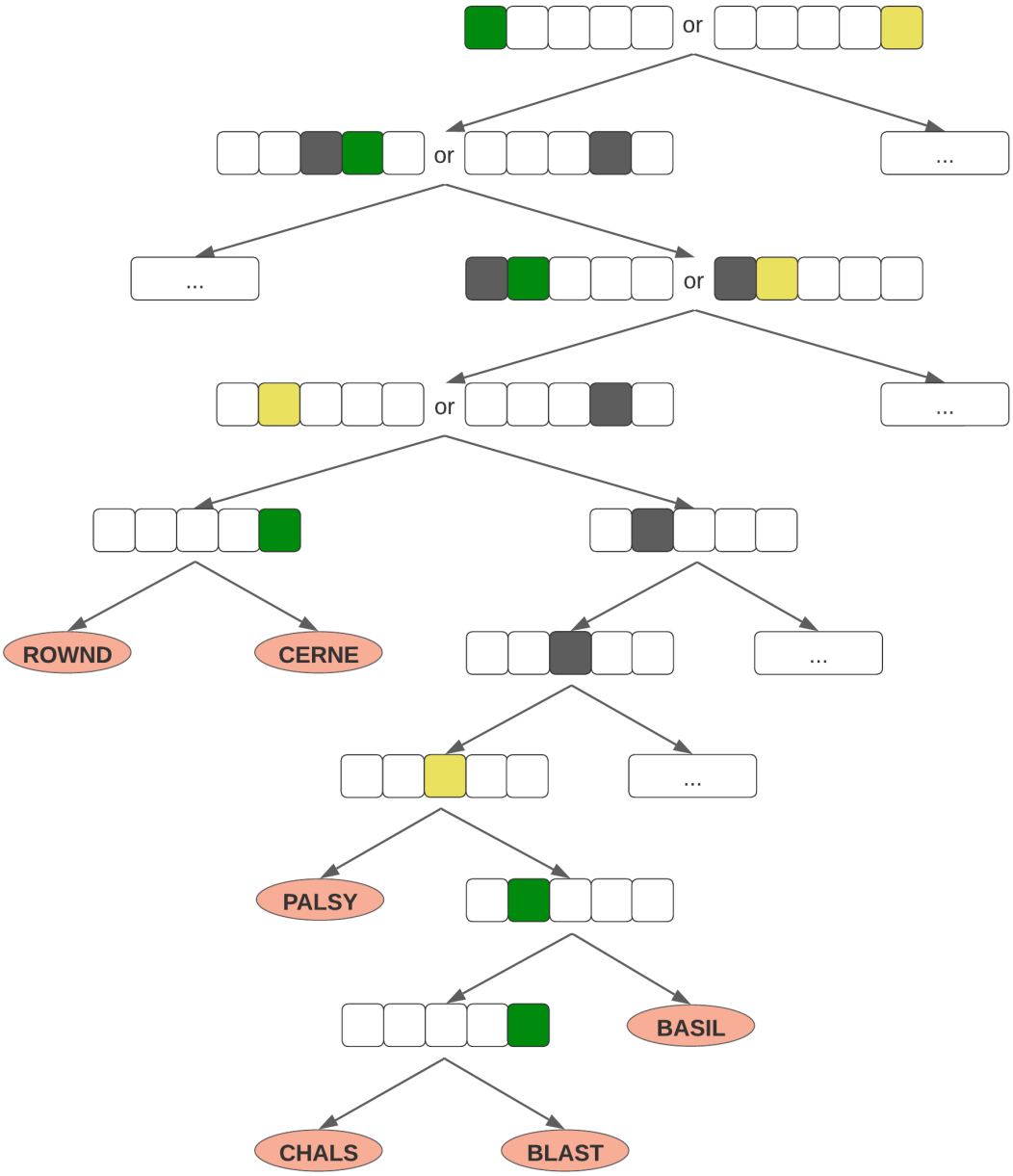


Figure 5 Figure depicting Optimal Classification Tree with Hyperplanes of an optimal policy for the second guess in Wordle, in which SALET was the first word guessed.

From the Figure, we see the interpretability afforded to us through the approach of using decision trees, in stark contrast to using a learning algorithm such as a Neural Network, which operates and learns policies as a black box. Specifically, through a short sequence of yes/no questions, anyone is able to understand the policy and, upon analyzing the tree, learn new strategies and understand important aspects of the optimal policy. Furthermore, we see the importance of using OCT-H, as

using CART to generate a decision tree to perfectly learn the optimal policy requires we create a tree with depth 12, rather than depth 8 as required by OCT-H – which is a significantly less interpretable tree that is harder to visualize. And finally, the utility of the hyperplanes is also showcased, as the first few layers of the tree ask multi-feature questions that are enabled by the hyperplanes – which allows the tree to be shorter by 2 layers, again improving its utility and practicality.

3.4. The Optimal Policy's Strength over Human Play

A key property of the optimal policy is that it finds the absolute best guess to make. As such, the next guess is derived from a consideration of all possible actions and the information that each of these tells. In many situations, this intelligent, global search can make the difference between winning the game with certainty or losing. For instance, consider the situation depicted in Figure 6a in which a human guesses the word LIGHT, revealing four green tiles in the final four characters. They then proceed to attempt finding the solution by changing only the first character — a strategy which ultimately loses them the game. While this greedy, local-search behavior is intuitive and represents a first-instinct to many, it can clearly be problematic as winning can be left to luck. On the other hand, in Figure 6b we depict the optimal strategy given that LIGHT was the initial guess and revealed four green tiles. As we can see from the Figure, instead of greedily searching through possible solutions, the optimal strategy makes a second guess which is not a possible solution but rather aims to ascertain the first letter from an array of potential solutions. That is, by guessing FERMS second, it tests whether FIGHT, EIGHT, RIGHT, MIGHT, and SIGHT are the hidden word, and is able to determine that EIGHT is the hidden word, winning the game with just 3 moves.

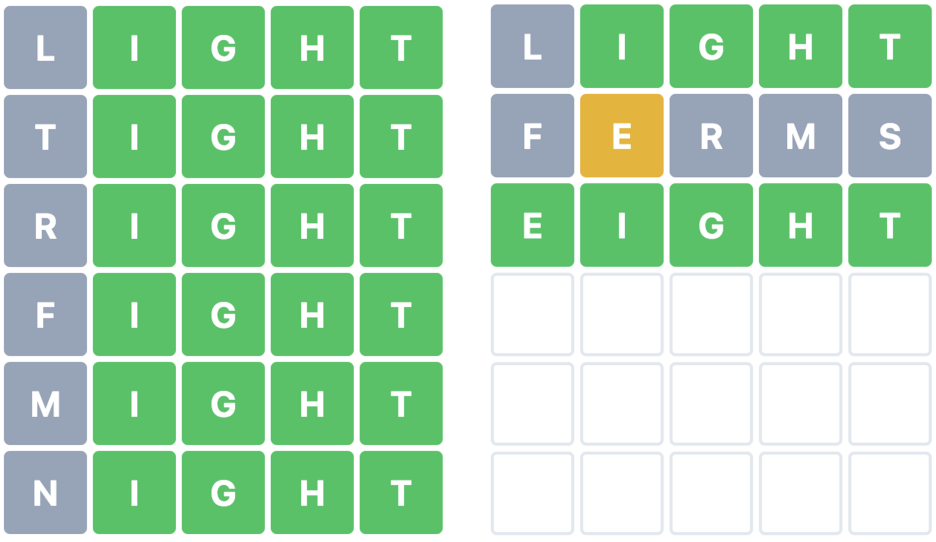


Figure 6 Figure comparing the resulting Wordle board of a possible human-played game and of the optimal policy.

Ultimately, this demonstrates the clear edge that following an optimal policy gives. And combining this with the interpretable depiction of an optimal strategy leaves the readers room to observe and employ interesting strategies to the game of Wordle, again highlighting the benefit of adding interpretability to policies generated in Dynamic Programming and Reinforcement Learning problems.

We end this section by presenting two additional examples that display the optimal policy's strength. Namely, situations in which the player is starved of information because of the appearance of many black tiles are notoriously difficult — these often occur when the hidden word has repeated and uncommon characters. However, as Figure 7 illustrates, the optimal policy is able to intelligently recover from these situations. Namely, it guesses characters that most frequently appear in words, thus cutting away a large number of potential solutions with each guess. When it ultimately gets a non-black tile (such as the yellow Y in the left-side of Figure 7, or the yellow U on the right-side), it is able to quickly determine the final solution from the small number of solutions still viable.

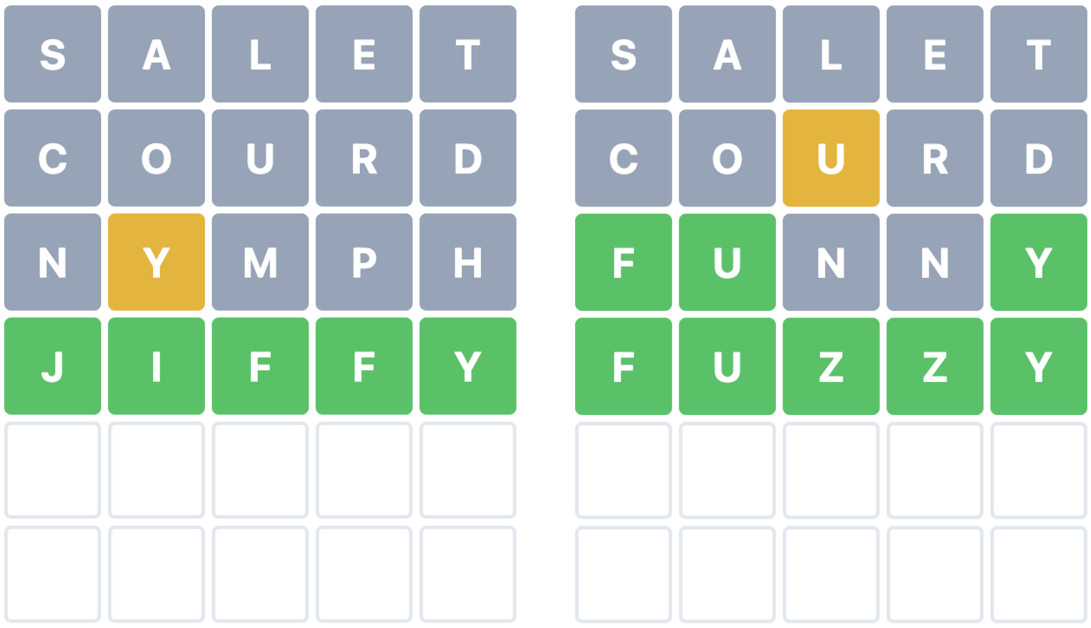


Figure 7 Figure portraying the optimal policy under difficult situations.

3.5. On the Difficulty of Heuristics and Reinforcement Learning to Uncover the Optimal Policy

In this section, we present a series of experiments aimed to help the reader understand why prior heuristics and applications of Reinforcement Learning to Wordle struggled to uncover an optimal policy, or even reach consensus on an optimal first-word. To preface this section, as discussed in the Literature section many prior attempts to solve Wordle have claimed to find "best" first words, yet many of these papers and works do not even converge to the same answer.

One common feature of these prior works — such as [Katz and Conlen (2022)], [Glaiel (2021)], or [Anderson and Meyer (2022)] — is that they directly seek future states that are "good," in the sense of requiring a minimal number of moves to win. Indeed, as discussed in Section 2.2, in this work we also seek the same objective, but do so in a certifiably exact manner — whereas prior work has no certification because of the use of heuristics. As we illustrate via experiments shortly, this becomes problematic when combined with the large action space of Wordle, which admits objective values (see (1)) that are very fine-grained. Namely, small errors in estimating the objective value causes sub-optimalities in the policy at later time steps, and these errors compound as the policy and state-values are back-propagated.

To illustrate this point, we performed an experiment in which we add noise to the value function V of states at $t=3$, and attempt to recover the optimal policy given that SALET is the first word guessed. Specifically, for a given state at $t=3, s_3$, we perturb $V_3(s_3)$ by up to $x\%$ of its value, and seek to compute the optimal policy under these noisy estimates of the value function. We note that this experiment seeks to emulate prior work, as they use approximate methods (such as 2-step greedy heuristics) to compute their policy.

Figure 8 presents the results of this experiment, which we re-run 25 times to account for randomness in the perturbations. On the y-axis, we have the expected number of moves needed to win Wordle under the policy uncovered, and on the x-axis we have the percent perturbation of the value function at $t=3$. For the reader's reference, we plot a horizontal red-line at value 3.42 to emphasize that an optimal policy for Wordle wins the game in 3.42 guesses on average. Finally, the blue line plots the median expected number of moves to win of the uncovered optimal policy, and the blue shaded region around this plots the maximum and minimum expected moves to win across the 25 simulations.

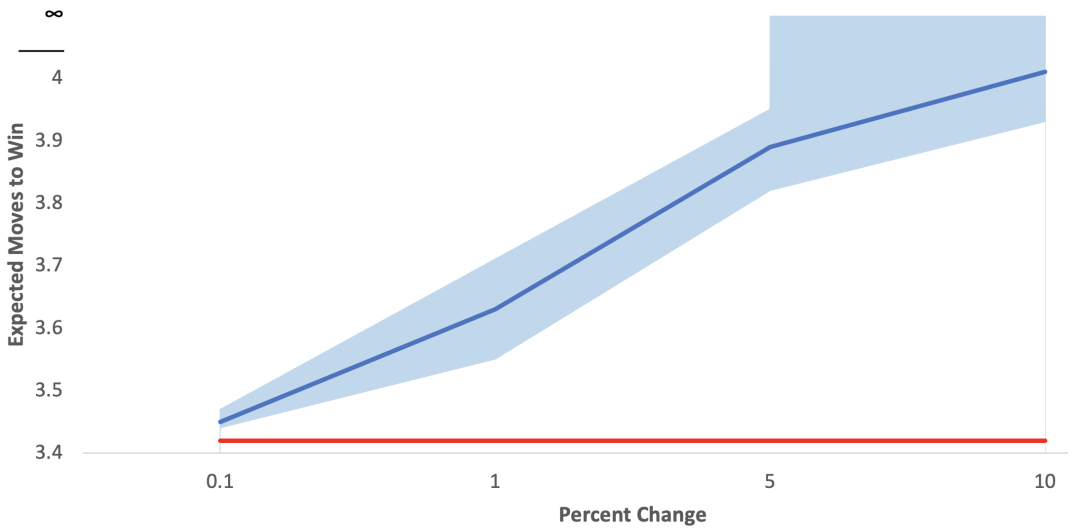


Figure 8 Graph illustrating the effect of noisy estimates of the value function on the uncovered optimal policy.

Interestingly, from the graph we see that as the perturbations become bigger, the deviation of the resultant optimal policy from the true optimal policy grows. So much so, in fact, that if we perturb the value function around 10%, some of the simulations result in a policy that does not win with certainty. Additionally, we see that even if the perturbations are as small as 0.1%, we still cannot recover the true policy, as the lowest point of the shaded region is still greater than the expected value of the true optimal policy. Finally, it is important to point out that this degradation in the policy occurs in just 3 time-steps from first guess – that is, that the error propagation's are significant for just 3 guesses from the start. Ultimately, then, this experiment again highlights and reveals why approximate methods – such as prior applications of reinforcement learning – struggle to uncover an optimal policy.

4. Conclusion

In this paper, we derive a novel, exact Dynamic Programming-based model for the game of Wordle, and present an algorithm that tractably solves the game and, to the best of our knowledge, produces the first certifiably optimal policy for the game. The crux of the algorithm comes from a series of theoretical and computational observations from the Bellman Equation solving the game, which can be found in Section 2.2.

Additionally, we use Optimal Classification Trees with Hyperplanes to display the optimal policies in a compact yet human-readable format. We emphasize the utility of this approach to portraying the policies, as standard look-up tables for the game grow dramatically in size for policies after the second guess.

Finally, we perform experiments to illustrate why prior, approximate approaches to solving the game have struggled to uncover an optimal policy. Ultimately, we find that a large sensitivity to approximating state-values contributes to difficulty in uncovering a true optimal policy for the game.

References

Anderson BJ, Meyer JG (2022) Finding the optimal human strategy for wordle using maximum correct letter probabilities and reinforcement learning. URL http://dx.doi.org/10.48550/ARXIV.2202.00557.

Bertsekas DP (2005) *Dynamic Programming and Optimal Control*, volume I (Belmont, MA, USA: Athena Scientific), 3rd edition.

Bertsimas D, Dunn J (2017) Optimal classification trees. *Machine Learning* 106(7):1039–1082.

Bertsimas D, Dunn J (2019) *Machine Learning Under a Modern Optimization Lens* (Dynamic Ideas press).

Bertsimas D, Paskov A (2022) World-class interpretable poker. *Machine Learning* (111):3063–3083, URL http://dx.doi.org/10.1007/s10994-022-06179-8.

Bonthron M (2022) Rank one approximation as a strategy for wordle. URL http://dx.doi.org/10.48550/ARXIV.2204.06324.

Breiman L, Friedman JH, Olshen RA, Stone CJ (1984) *Classification and Regression Trees* (Wadsworth and Brooks).

Glaiel T (2021) The mathematically optimal first guess in wordle.

Katz J, Conlen M (2022) Introducing wordlebot, the upshot's daily wordle companion.

Liu CL (2022) Using wordle for learning to design and compare strategies. URL http://dx.doi.org/10.48550/ARXIV.2205.11225.

Lokshtanov D, Subercaseaux B (2022) Wordle is np-hard. URL http://dx.doi.org/10.48550/ARXIV.2203.16713.

Malik A (2022) Popular puzzle game wordle is being purchased by the new york times. URL https://tcrn.ch/3HletQK.

Algorithm 2 Compute $V_t^*(s_t)$

Input: Integer t, State s_t, Dictionary[(Set,Integer)→Float] V_{memory}
Output: Float $V_t^*(s_t)$
if $t = 6$ or ($t = 5$ and $|s_t| > 1$) **then** ▷ Optimization 1
    return ∞
else if $t = 5$ **then**
    return 1
else if $|s_t| = 1$ **then**
    return 1
else if $|s_t| = 2$ **then** ▷ Optimization 2
    return 1.5
else if $(s_t, t) \in V_{memory}$ **then**
    return $V_{memory}[(s_t, t)]$
end if
state_value $\leftarrow \infty$
for action $\in (s_t \cup (A - s_t))$ **do** ▷ Optimization 5
    temp, next_states $\leftarrow 1$, Get_Transition_Info(s_t, action) ▷ Optimization 3
    if next_states.size $= 1$ and $s_t \in$ next_states **then**
        continue ▷ Skip action, no info revealed
    end if
    for $s_{t+1} \in$ next_states **do**
        temp $\leftarrow$ temp $+ \frac{2|s_{t+1}|-1}{|s_{t+1}|}$ ▷ Optimization 4
    end for
    for $s_{t+1} \in$ next_states **do**
        if temp $\geq$ state_value **then**
            break
        else if $s_{t+1} =$ {action} **then**
            continue
        end if
        temp $\leftarrow$ temp $+ V_{t+1}^*(s_{t+1})$
    end for
    if temp $<$ state_value **then**
        state_value $\leftarrow$ temp
    end if
end for
return state_value