Policy Generation for Wordle

Jashan Chopra, Fernando Palafox, Amit Dubey

University of Colorado Boulder

ASEN 5519 - Decision Making Under Uncertainty (Spring 2022)

I. Introduction

Wordle is a simple puzzle game where a player guesses an unknown five letter word. It was created in October 2021 by Josh Wardle for himself and friends, but released to the public. The game grew in popularity through social media, as it was easy to share one's score with visualizations automatically created by the game. Two months after release, hundreds of thousands of people were playing Wordle daily, and it was soon purchased by the New York Times [8].

Each day a random five letter word is selected. The player has six tries to try and guess the target word. When a player guesses the word, the color of the tiles representing letters change to indicate how good the guess was. If the tile turns green, the letter was both in the target word, and in the same location as the guessed word. If the tile turns yellow, the letter was in the target word, but not in the same location as it was guessed. If the tile color stays black, the letter is not in the target word.

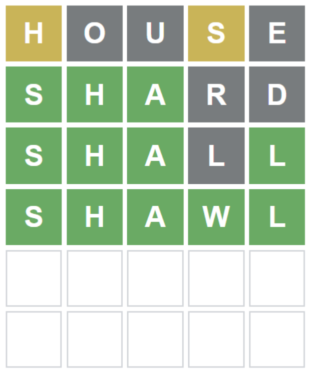


Fig. 1 A completed Wordle game

We see this in a sample game to the right. Our first guess was "HOUSE". After that guess, we needed a word with an "S" and "H". Our 'heuristic' policy for this is to try a word starting with "SH", because that's a word pattern we use often. Many questions arise from a game of Wordle. Is there an optimal first guess to maximize the information you receive? Should we always use the information, or should we guess totally different words to try and eliminate the most letters? Given a set of valid guesses and solution words, what is the minimum number of tries required on average to solve the game?

Our goal is to create a policy for finding the solution to the Wordle game in as few tries as possible. Our minimum viable policy is one that guesses a random word from the valid word list each time. Since there are 12974 possible valid words, randomly guessing the word will almost always fail to get the correct answer in the maximum amount of six attempts. A great policy should outperform the average score, which we assume to be four guesses. Our stretch goal is to analyze the generated policies and see if we can extract relevant statistics and information to answer the aforementioned questions.

II. Background and Related Work

As players of this game we took interest in how well a computer could solve it, and given the introduction of games in our course material from ASEN5519 Decision Making Under Uncertainty, it seemed well suited for a final project. Given the games recent release, there are not yet any published papers on solution methods for Wordle. However, we can draw on blog posts and videos for insight into how others are attempting to solve Wordle, and look at published papers for other games to understand the methods used to approach these types of challenges.

First, we looked at a video from Grant Sanderson, a YouTuber known as 3Blue1Brown[9]. He used a maximum entropy approach to select words based on information theory [4], where the chosen guess minimizes the residual entropy of the game. It shares the same principles as the maximum entropy IRL method discussed in class. The principles of maximum entropy are used to resolve ambiguities in choosing distributions over states and consequently actions [6] [7].

We then found a blog post by Andrew Ho, discussing a deep reinforcement learning based approach [2] to solving Wordle. Ho used an Advantage Actor Critic (A2C) approach, which we had discussed in class but not had a chance to implement ourselves [10]. This method achieved a score of around 3.9 guesses, and provided helpful information when we were constructing our POMDP definition. We ultimately decided to use the Proximal Policy Optimization (PPO) algorithm [3] due to its maturity in the Julia package we used.

The first step to finding an optimal policy to this game was to first define it as a (PO)MDP, so that we could apply different techniques to the problem and see what worked best. Although we learned about these different approaches and techniques through class and applied them through interfacing with the Julia POMDP packages [12], we cite here some of the original research papers on the techniques. Littman et al., 1995 [1] provide insight into the QMDP method we learned in class . We will employ QMDP first as it is one of the more simple POMDP solvers to understand, although we are unsure if it will work due to the large size of our state space. On the reinforcement learning side, it will be interesting to look at the work done by Deepmind on their AlphaGo project [5]. Although our work is nowhere near as advanced, we share the similarities of a large state space and lack of aleatory uncertainty. If we have the time, exploring the combination of deep RL learning with the tree search techniques we've learned in class may bear fruit, although our game does not have another player like Go.

III. Problem Formulation

POMDP Definition

The game's structure lends itself to a Partially-Observable Markov Decision Process (POMDP). We have a target word which cannot be directly observed, and instead must be inferred through making guesses and utilizing information from those guesses. We chose to represent the state as a vector containing the the true word and the turn number (Eq. 1). The total state space is given by the permutation set of every word in the valid word list of 12,974 words, and an integer from -1 to 7 which corresponds to the turn number. Turn 0 represents an empty game board. Turns 1-6 represent guesses the player is allowed to make. Turn 7 represents a failed game, i.e: the true word was not guessed within the maximum number of guesses. Turn -1 is used to signal a terminal state, which could happen due to a failed or successful game. An initial state of the game is a tuple containing a random word from the set of valid words, and a turn counter set to 0.

$$s = [\text{target word}, \text{turn number}] \tag{1}$$

An action in the game of Wordle is a word guessed by a player. Therefore, the total action space is the set of valid words. At each turn, the player must guess a word, thus there are no other actions outside of guessing a word.

Transitions in this POMDP are given by Eq. 2 which takes in the current state s and an action a, and returns the next state s'. Given any a and s pair, s' is given by the target word of s and a turn number of 1 higher than the turn number of s. The turn number will always increase by 1 unless the turn number of s is 7 (implying we've already run of out tries) or the a and the target word of s are identical (implying a correct guess has been made). For either of these cases, the turn number of s' is instead given by -1, which signals a terminal state.

$$s'(s,a) = \begin{cases} [s(1), -1] & \text{if } a = s(1) \vee s(2) = 7 \\ [s(1), s(2)+1] & \text{if otherwise} \end{cases} \tag{2}$$

In Wordle, one cannot directly observe the target word and can only infer it based on the information returned from each guess. For this POMDP, an observation of the state was defined as the set of words that are compatible given the information gained after taking an action. Using the example game in the introduction, once we guessed "HOUSE" the game let us know that the target word contained 'S', and 'H', but did not contain 'O', 'U', or 'E'. This information was then used to eliminate words from the complete set of possible words. The set of remaining words was then returned as the observation. This means that the total observation space is the power set of the complete set of possible words. This definition of the system also meant that the observation function was deterministic for any given state and action.

The reward function represents the two goals of the game: guessing the correct word in the least amount of tries. Thus, we give a large reward (+100) for guessing the correct word. We assign a negative reward for each guess proportional to the turn number. Thus, the first guess gives a -1 reward, the second guess gives a -2 reward, etc. Failing to solve the game within the maximum six guesses returns a larger negative reward (-25).

MDP Definition

A custom MDP environment is created by constructing a wrapper around the game using the *ReinforcementLearning.jl* package [11]. The MDP state encodes the entire history of the game to maintain the Markov property. It is a 131-element vector. One element is used to track the current iteration in the game. The other elements in the vector encode if each letter is a perfect match for one of the five locations in the word, is not a match but exists in the word or does not exist in word. Thus for each letter there is an associated 5-element sub-vector corresponding to each location in the world. This accounts for the remaining 130 (26x5) elements in the state.

Fig 2 shows an example of the initial state and the state transition after the first guess. The initial state is all 1, since each letter is a valid candidate for each of the locations. After the first guess word "SNEAK", "S" is set to 2 in its position 1. All other letters are set to 0 in their position 1. "N", "E" and "K" are set 0 in all their 5 locations as these letters do not exist in the word. "A" and all the letters that have not been tried yet remain at 1 in all their locations except for location 1 which we now know is an "S".

The action space for the MDP corresponds to the 12974 valid guess words in the game. At each iteration of the game, the action consists of choosing one of these words. To save on computation each word is encoded as a 130-element one-hot vector. For each of the locations in the encoded word, we mark a letter as 1 if it exists in that location and 0 if it does not. Fig 3 shows an example of encoding the word "SHAWL".The transition between the states are completely deterministic given an action. The state transitions with a probability of 1 as described above and shown in Fig 2 for a given action. All other transition probabilities remain at 0.

For every action the policy takes it receives a reward of -1. There are no positive rewards and the objective is to minimize the total negative reward. This matches the objective of the game to guess the target word in a minimum number of iterations. We do not give intermediate rewards or perform exquisite reward tuning or shaping. Since Wordle is a finite duration, short-horizon game, we set the discount factor $\gamma = 1$

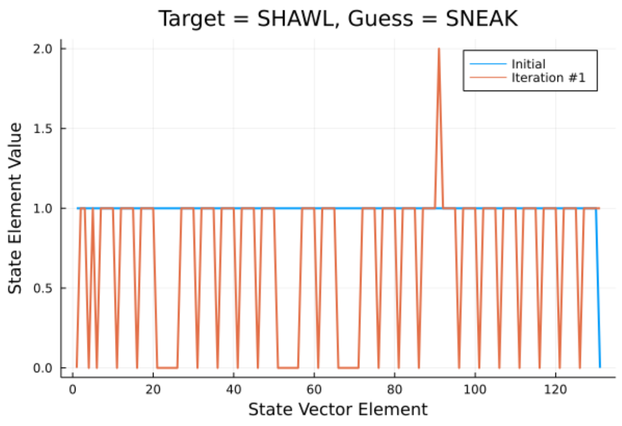
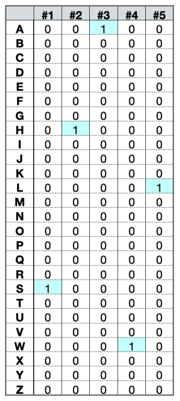


Fig. 2 MDP state vector and transition



	#1	#2	#3	#4	#5
A	0	0	1	0	0
B	0	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	0	0	0
F	0	0	0	0	0
G	0	0	0	0	0
H	0	1	0	0	0
I	0	0	0	0	0
J	0	0	0	0	0
K	0	0	0	0	0
L	0	0	0	0	1
M	0	0	0	0	0
N	0	0	0	0	0
O	0	0	0	0	0
P	0	0	0	0	0
Q	0	0	0	0	0
R	0	0	0	0	0
S	1	0	0	0	0
T	0	0	0	0	0
U	0	0	0	0	0
V	0	0	0	0	0
W	0	0	0	1	0
X	0	0	0	0	0
Y	0	0	0	0	0
Z	0	0	0	0	0

Fig. 3 One-hot encoding of the word "SHAWL"

IV. Solution Approach

The team took a few different approaches toward solving the Wordle problem. First, we implemented a simple heuristic policy. Since there's no uncertainty in the transition dynamics, a heuristic policy works quite well. This heuristic policy also serves as comparison for policies developed through other approaches, like QMDP or Reinforcement Learning. We expect our developed policies to do better than the random policy, but comparing to our heuristic policy allows us to see if the types of solvers we've utilized in class work well for a problem of the same nature as Wordle.

Heuristics

We implemented two heuristics. Our first heuristic policy is a history-based policy that guesses words randomly but utilizes information from prior guesses to eliminate words that are guaranteed not to be the answer. Each turn we choose a random word from that list, which for the first turn is the same as the total word list. Using the POMDP defined observation we construct a list that contains only valid words for the current WordleGame. On the next turn, the random word is then selected from this pruned list.

Our second heuristic policy is based on maximum entropy prioritization. Entropy is the information theoretic measure of uncertainty. The policy takes actions that result in distribution over states with maximum entropy. The information we get about the target word from each state after a guess is inversely proportional to the state's probability (Eq. 3). The average amount of information considering all possible states and their probabilities is entropy (Eq. 4).

With any guess the game can return 1 of 243 states depending on the target. This heuristic chooses the guess at each iteration that will have the most uniform distribution over the states considering all the target words that are still in consideration. This results in minimizing the residual entropy, that is, the largest possible elimination of words at each step. In Fig. 4 and Fig. 5 we show distribution over states for a guess that maximizes entropy and a guess that minimizes entropy. As expected the guess that maximizes entropy has more uniform distribution over states.

Fig. 6 shows the starting distribution of entropy for all valid guesses in the game. We can see that our best starting guess will have an entropy of about 6.3.

$$I(X) = log_b(\frac{1}{P(X)}) \rightarrow I(X) = -log_b P(X) \tag{3}$$

$$H(X) = E[I[X]] = E[-log_b P(X)] \rightarrow H(X) = -\sum_{i=1}^{n} P(x_i) log_b P(x_i) \tag{4}$$

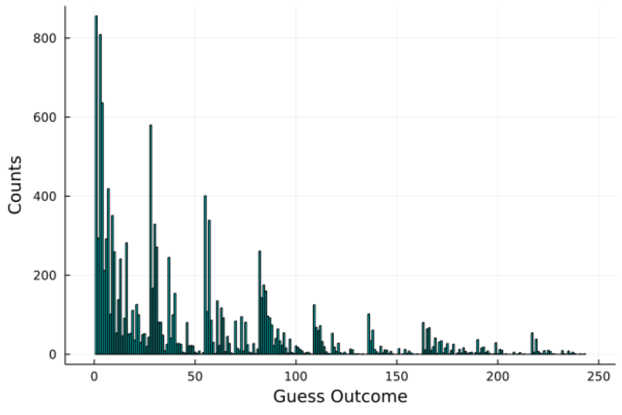


Fig. 4 Maximum entropy guess

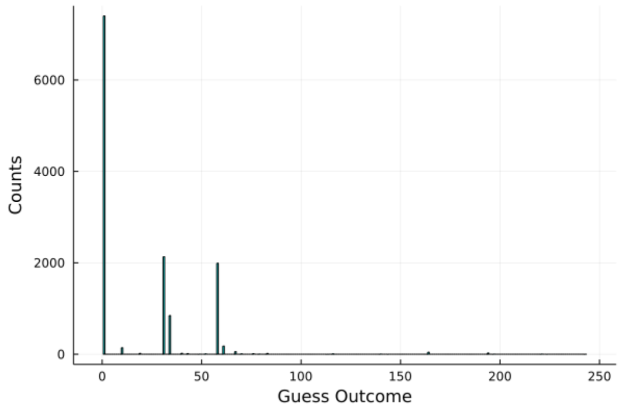


Fig. 5 Minimum entropy guess

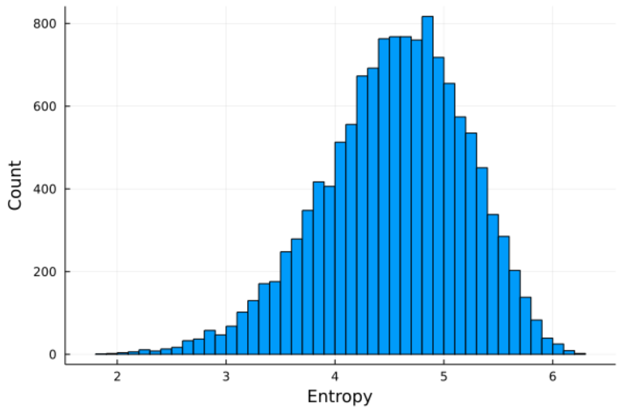


Fig. 6 Starting word entropy

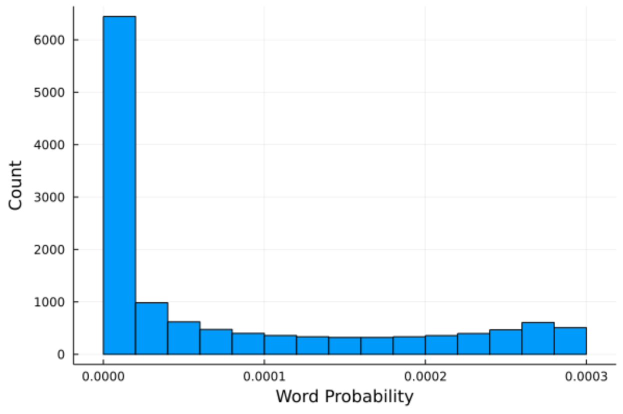


Fig. 7 Word occurrence probability

The target words in Wordle are commonly occurring words from the English language. We use this information to improve our policy and lower the average number of required iterations. We see from Fig. 7 that about half the valid guess words have very low probability of occurrence. We know that these words can't be valid targets. The remaining half have roughly uniform probability of occurrence though the probability of any given word occurring is still considerably low. The word probability is normalized after every guess such that the occurrence probability of words that have not been eliminated sums to 1. The probability associated with each word increases after every guess as we are elimination a significant number of words at each iteration and then normalizing. We set up a hypothesis test to determine if the current word with the highest probability of occurrence is the target word (H_0) (Eq. 5). Fig. 8 shows that words with normalized probability of occurrence of 0.40 or greater are more likely to be the target word. Fig. 9 shows that this probability threshold is crossed after the second iteration on average.

$$\frac{p_0(\mathbf{x})}{p_1(\mathbf{x})} \underset{H_0}{\overset{H_1}{\lessgtr}} 1 \tag{5}$$

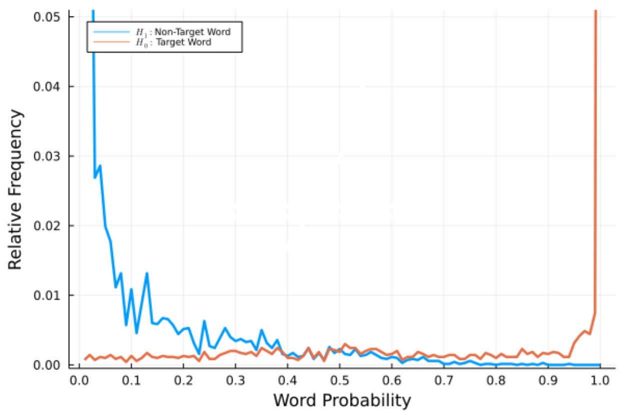


Fig. 8 Target-word hypothesis test

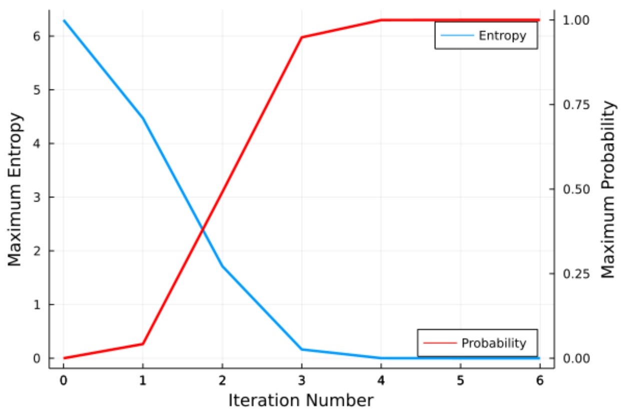


Fig. 9 Entropy and probability vs. iteration

QMDP

Given the POMDP description, the team used a QMDP-based solver to solve the game. The team implemented a simple belief updater which takes in an observation (a set of words), and updates the corresponding belief probabilities to the reciprocal of the observation's cardinality. This is essentially a uniform distribution over the words contained in the observation. QMDP is based on value iteration which relies on the iterative computation of a value function. Traditionally, this computation requires a transition matrix, however constructing a transition matrix for the action space (12974 actions and the complete state space (9×12974 word-turn tuples) resulted in a matrix too large to be handled by our personal computers. Fortunately, the team was able to exploit the structure of the problem (deterministic and simple transitions) to modify the value iteration algorithm so that it would not require any exorbitantly-sized matrices, and instead only required a single 12974×12974 matrix - still large, but manageable. The exact QMDP implementation can be found in the appendix.

Reinforcement Learning

We used actor-critic based Proximal Policy Optimization (PPO) algorithm to find the optimal policy for the Wordle MDP. PPO is a policy-gradient technique that finds distributions over the action space that maximize the average return. The primary reason to choose PPO was that it can support multi-dimensional and continuous action spaces. Our architecture (Fig 10), known as Wordle-NET, takes a 131-element state vector. There are two, 256-wide, fully-connected dense inner layers. The network outputs a 130-element action vector that represents the one-hot encoded words from the valid guess word list. We used the Adam optimizer for our stochastic gradient step to update network weights.

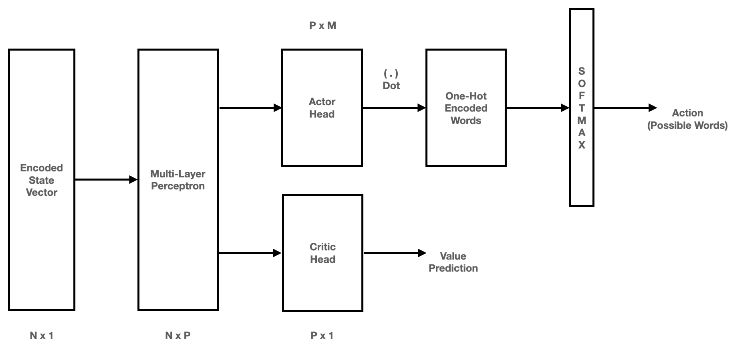


Fig. 10 Wordle-NET architecture

V. Results & Quantification of Success

Perfect Policy & Evaluation Metrics

Our evaluation function takes in the Wordle POMDP, a function that returns a word for guessing, and the number of evaluation trials. For each trial, we create a WordleGame struct with a random word from the full word list. We then loop six times, the number of turns Wordle provides. Each turn we call the provided function to construct a guess depending on the state of the WordleGame, and we also increment a counter of the game score. If the guess is equivalent to the game's target word, then we update a counter of games won and break the loop. The game score is summed over each evaluation trial and then averaged at the end for a final score. Our evaluation function returns this final score and the number of games won. Thus, this final score is essentially a measure of how many turns we took to solve Wordle on average. The higher the final score, the worse the performance, like golf. A perfect policy is one that always guesses the correct word. This policy will have a 100% win rate, and a final score of 1.0.

Random Policy

The random policy is straightforward, for every turn of the WordleGame we guess a word randomly from the entire word list. Since the word list is quite large, this policy almost always gets the worst possible score: 7.0.

Heuristics

The word-elimination heuristic policy is fairly poor, managing to win just about 80% of the 1,000 random games played, but with a high score. There are a few reasons why this heuristic does poorly. First, the initial guess is typically bad, and not something that a human would use. For example, the word "PIZZA" may be selected as a random first guess, but this word will not eliminate many words because two of the letters are 'Z', which few words in the list contain. Second, there are many Wordle games with a target word that contains many common letters, for example a word like "CRANE." Even after five or more guesses, there may still be a fairly sizeable list leftover of words that this heuristic will randomly choose from. Third, this evaluation method and heuristic have no knowledge of the actual words that would be used to construct Wordle games. The actual word list used to generate games is a smaller subset of the overall word list, and thus humans are generally inclined to pick words they know, which in effect are more likely to be words chosen for games. However, our evaluation method creates random games, and this heuristic is just as likely to pick a more well known word as it is to pick something like "ZYMIC" or "ABACI". We see that if we run the evaluation with the smaller set of solution words, we get an improved final score.

Maximum entropy heuristic performs much better with an average score of 3.63 and 100% win rate. Fig 12 shows the distribution over number of iterations needed over the test data set. A game won with this heuristic is shown in Fig 11.

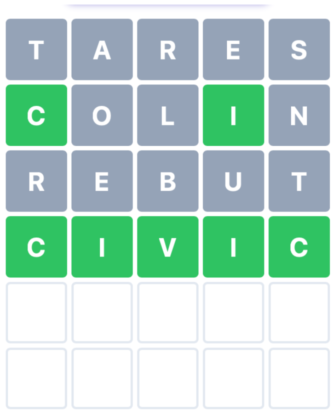


Fig. 11 Game illustration with maximum entropy heuristic

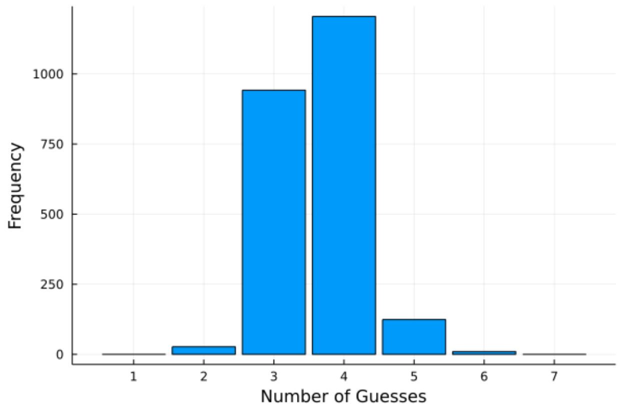


Fig. 12 Iterations needed to solve the game

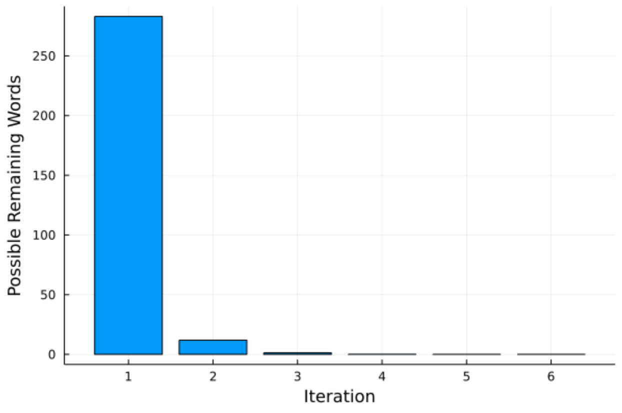


Fig. 13 Number of possible solution words

The word "REBUT", which would be an unlikely choice for a human player based on the given game state, was used based on its entropy to eliminate the maximum number of possible words. Thus, it can be advantageous to guess words that definitely can't be the target word to eliminate possibilities. Table 1 shows the game scores with different starting words ranked according to their entropy. We can see that the starting words with higher entropy result in desired lower average scores. Though choosing the word with absolutely the highest entropy is not critical. Different starting words with entropy above 6.1 result in similar scores. Fig 13 shows that we can reduce the number of potential candidates from 12974 to approximately 275 after just the first guess. This is because we have gained approximately 6.3 bits of information by choosing the high entropy starting words. As described earlier, in addition to entropy we also consider the word probability information. Table 2 shows that if the occurrence probability of a word increases to more than 0.40, it is better to use that as the next guess instead of using the word with maximum entropy.

Starting Word	Entropy	Game Score	Win %
TARES	6.29	3.66	100.0
LARES	6.21	3.66	100.0
RATES	6.19	3.63	100.0
RALES	6.17	3.65	100.0
TEARS	6.16	3.63	100.0
YUKKY	2.02	4.23	100.0
IMMIX	2.02	4.23	100.0
FUFFY	1.96	4.31	99.9
XYLYL	1.95	4.31	99.9
QAJAQ	1.80	4.32	99.9
Random Starts		3.85	99.9

Table 1 Different starting strategies

Probability Threshold	Game Score	Win %
0.05	3.72	98.5
0.15	3.64	99.4
0.25	3.61	99.7
0.35	3.63	99.9
0.45	3.66	100.0
0.55	3.72	100.0
0.65	3.77	100.0
0.75	3.80	100.0
0.85	3.85	100.0
0.95	3.90	100.0

Table 2 Target-word hypothesis test based on occurrence probability

Reinforcement Learning

To train Wordle-NET, we started with a simplified problem. Due to the fixed structure of the game it was not possible to start with shorter words. Instead we started with a shorter list of words that could be valid targets and guesses. Fig 14 shows the learning curve with a shortened list of 100 words. With a shorter list of words the network did learn to select actions that could complete the game in less than 6 iterations. The best average score was approximately 5.3. Though we also see that if we continue to train the network, the network weights start to diverge and the performance suffers. This shows that we need to employ early-stopping in our training strategy and periodically save the weights. The test should also be performed with the best saved policy. Though there was some success with the smaller list of words, the network could not be trained to work with the full list of 12974 words. It was observed that the policy would get stuck in a local minima and the network would simply cycle through a few select words as the output. A full grid-search over the hyperparameters turned out to be prohibitively expensive computationally. The final performance over the full word list was comparable to a random policy.

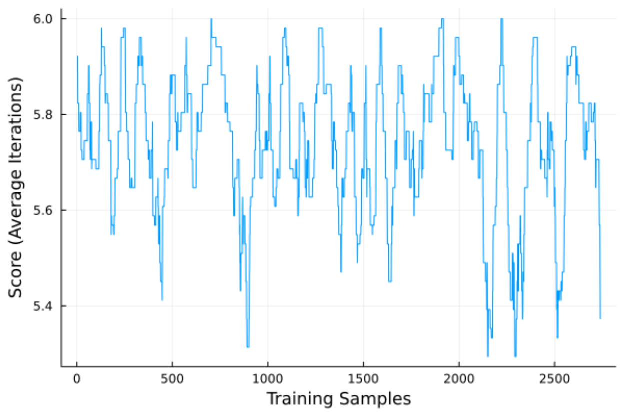


Fig. 14 Learning curve with PPO (reduced actions)

QMDP

Results for the QMDP solver show limited performance with a mean score of 4 but a win rate of only 41%. However, improved results were seen if the set of valid words was reduced to a smaller number. For example, a set size of 200 resulted in a mean score of 3.5 with a 82% win percentage. Large data sets also proved to be prohibitively expensive in terms of computation time - even with the efficiency modifications made to the QMDP algorithm. Perhaps issues arise from the QMDP assumption of full observability after the first step, which in this case may result in a value function which is overly optimistic. Another source of error may be the uniform distribution over the words in an observation returned by the belief updater. This may work well for small word sets since the chances of selecting the correct word are better, but may not be ideal for large word sets. One main issue with QMDP was that it would frequently continue to select the same word even after it had been shown to be incorrect.

Consolidated Results

The evaluation results of each method described above for policy generation is given in Table 3. We evaluated 1,000 trials for each of the policies except the cheating policy, since the outcome of that policy is deterministic.

Policy	Number of Trials	Final Average Score	Win Percentage
Perfect	10	1.0	100%
Random	1000	6.999	0.1%
Elimination Heuristic	1000	5.662	80.7%
Reinforcement Learning	1000	6.999	0.1%
QMDP	1000	3.701*	38.4%
Maximum Entropy Heuristic	1000	3.625	100%

* QMDP score removes lost games, which is why the score is so low even though the win percentage is low. This was done to show that when QMDP works it achieves a good score.

Table 3 Evaluation games chosen from solution set

VI. Conclusion & Future Work

We saw promising results from our work to solve Wordle. We were able to meet our minimum viable product of effectively defining Wordle as a (PO)MDP and implementing basic random solvers for these environments. We then satisfied our main approach by constructing a theoretically principled heuristic policy that achieved a 100% win rate and beat our target of being able to get an average score of less than 4. We were also able meet our stretch goals by evaluating the results from this policy and answering the questions we had posed earlier about optimal approaches to play Wordle. These answers can guide human players in their own play strategies as well.

Our QMDP and reinforcement learning generated policies did not achieve strong performance over the complete data set, mainly due to the policies continuing to select words from a very small subset. However, as we saw in class, developing a heuristic policy with knowledge of the problem can often perform better than many of the methods used in class. We also encountered an interesting debate with changing QMDP so much. Since we had to modify the belief updater, and the evaluation method, at what point does that basically become a heuristic itself?

There were improvements we could have made to achieve better or similar performance to the maximum entropy heuristic. We could have used an online solver to combat performance related issues we saw in QMDP, for example, a sampling-based solver like DESPOT may have been able to better handle the massive state and transition matrices. To improve the reinforcement learning method, we could have experimented with different policy-gradient and value iteration algorithms and performed a more extensive grid search over hyperparameters.

VII. Contributions & Release

Jashan Chopra handled creation of the baseline code and project organization. Specifically, Jashan worked on evaluation wrappers for the various policies, the creation of the perfect, random, and elimination heuristic policies, and contributed to initial layout of the Wordle POMDP. Fernando Palafox worked on refinement of the Wordle POMDP and the QMDP solver. Specifically, he refined the definition of the observations, transitions, and implemented a belief updater. Additionally, he adapted the QMDP baseline code to efficiently work with the large state space. Amit Dubey worked on the reinforcement learning approach, coded the maximum entropy heuristic and performed the associated experiments for tuning parameters and generating analysis. All three members worked on the final report equally.

The authors grant permission for this report to be posted publicly. The final code is also publicly available at https://github.com/JashanChopra/WordleSolver.

References

[1] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling, “Learning policies for partially observable environments: Scaling up,” in Proceedings of the Twelfth International Conference on Machine Learning, 1995, pp. 362–370.

[2] V. Mnih et al., “Human-level control through deep reinforcement learning.,” Nature, vol. 518, no. 7540, pp. 529–533, 2015.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms.,” CoRR, vol. abs/1707.06347, 2017.

[4] C.E. Shannon, "A Mathematical Theory of Communication", Bell System Technical J., vol. 27, pp. 379-423, July and Oct. 1948.

[5] D. Silver et al., “Mastering the Game of Go with Deep Neural Networks and Tree Search,” Nature, vol. 529, no. 7587, pp. 484–489, 2016.

[6] B. D. Ziebart, A. L. Maas, J. A. Bagnell, and A. K. Dey, “Maximum Entropy Inverse Reinforcement Learning.,” in AAAI, 2008, pp. 1433–1438.

[7] E. T. Jaynes, “Information Theory and Statistical Mechanics,” Phys. Rev., vol. 106, no. 4, pp. 620–630, 1957.

[8] Wordle is joining the New York Times games. (2022, January 31). The New York Times Company. https://www.nytco.com/press/wordle-new-york-times-games/

[9] Solving Wordle using information theory. 3Blue1Brown. https://www.3blue1brown.com/lessons/wordle

[10] W&B. (2022, May 1). Weights & Biases. https://wandb.ai/andrewkho/wordle-solver/reports/Solving-Wordle-with-Reinforcement-Learning–VmlldzoxNTUzOTc4

[11] https://juliareinforcementlearning.org

[12] https://github.com/JuliaPOMDP/POMDPs.jl