Exploring Algorithms for Optimal Play in Wordle

Name: Max Van Fleet, Jun Ikeda, Vasishta Tumuluri **Professor:** Griffith
Date August 9, 2024

Background and Significance

Since its inception in 2021, the word puzzle game Wordle has gained immense popularity across the world. In each game of Wordle, a five-letter word is chosen from a set of commonly known five-letter words to be the "solution" word, whose identity is unknown to the player. The player then has six attempts to guess the word, where each guess must be an element of a larger set of five-letter words. After each guess by the player, each letter of the guess is colored in a way that provides some information about the word; specifically, a letter in the guess is colored green if it appears in the solution in the same index as in the guess, yellow if it appears in the solution but in a different index than it does in the guess, and gray if it does not appear in the solution.

Owing to Wordle's global popularity, particularly following its purchase by the New York Times in spring 2022, there have been numerous attempts to develop optimal approaches to playing Wordle, which vary widely in their efficacy and computational tractability and efficiency. However, these approaches often make different assumptions about the game or use different parameters than one another; one frequent difference, for example, is conflicting interpretations of the objective of the game, namely whether players should be trying to minimize the number of guesses to correctly guess the solution, or if it only matters that we correctly guess the solution within the allotted six guesses. There are also inconsistencies between different sources on the lists of valid solutions and guesses, thus changing the parameters of the game between approaches using different sources, resulting in authors' empirical tests of their approaches not being directly comparable. Additionally, while we can discuss different approaches' time complexity theoretically, differences in actual implementations make it difficult to discuss their time efficiency in a way that would allow us to consider their comparative runtime for practical use.

In this project, we examine what we consider to be two of the most notable attempts at optimal/near-optimal play in Wordle: a combination of rank-one approximation with latent semantic indexing to find the most representative word by Bonthron, and an exact dynamic programming solution that provably solves Wordle by Bertsimas and Paskov. We additionally consider some less sophisticated approaches of our own creation, namely a random algorithm that serves as a lower bound for efficacy (essentially a "gold standard of badness") as well as two similar greedy heuristic algorithms, of which one minimizes the maximum number of possible remaining solutions at the start of the next stage, and the other minimizes the expected number of possible remaining solutions at the start of the next stage. We additionally create a modified version of Bonthron's algorithm that implements principal component analysis and cosine similarity in order to make it more effective. We compare these approaches' efficacy and efficiency on a level playing field by keeping assumptions and problem parameters consistent, and by considering their time efficiency with similar implementation specifics to the extent that is practicable. Although we are unable to implement Bertsimas and Paskov's exact solution ourselves due to computational constraints, we are interested in comparing our results from the other aforementioned approaches to their exact solution in terms of efficacy.

Methods

Problem Setup

In Wordle there are two important sets of words: the set of all words that could potentially be the solution at the start of the game (we will refer to this as S), and the set of all words that can be accepted as guesses (we will refer to this as G). Note that $S \subset G$. The Wordle dataset from the source code before its purchase by NYT in 2022, which we found on Kaggle, was used for our analysis. There are 2315 solutions and 10657 additional words for 12972 words in total i.e. $|S| = 2315$ and $|G| = 12972$. This was also the dataset used by Bertsimas and Paskov.

We use a Markov Decision Process framework to formulate Wordle. In particular, we use two similar formulations of the game that differ only in their reward functions. Our MDPs have 7 decision epochs (6 for the 6 guesses, and 1 more to observe the final outcome), meaning $T = \{1, 2, ..., 7\}$. The action space at every epoch and state will just be G, as any word in the set of permissible guesses can always be guessed. The state space is $\{0, 1\} \times P(S)$ i.e. the set of all 2-tuples where the first component is 0 or 1, and the second component is any element of the power set of S. The state at each epoch t is a 2-tuple (i_t, S_t); i_t is an indicator variable for whether we have guessed the solution correctly yet or not, meaning we start at $i_1 = 0$, and $i_t = 1$ iff we have correctly guessed the solution at epoch $t - 1$ or before. S_t is the set of words that could still potentially be the solution at epoch t. For example, when the game starts (i.e. before the first guess), we will have $S_1 = S$ so the state at epoch 1 is $(0, S)$. After each guess g_t, we use the coloring algorithm to get the coloring of the letters of g_t, then an elimination algorithm to eliminate potential solutions that are no longer possible; this latter algorithm takes as input the current set of potential solutions S_t, the current guess g_t, and the coloring of the current guess g_t, and it iterates through the remaining potential solutions s to check if it can be deduced that s must not be the solution word, meaning it computes the set:

> $E_t = \{s \in S_t \mid$ it can be correctly deduced from guess g_t and its coloring that s must not be the solution word$\}$

Then it sets the next state S_{t+1} as the set difference $S_{t+1} = S_t \setminus E_t$.

For defining transition probabilities, we first note that if in epoch t we take action (guess) g_t, then there are up to $|S_t|$ states we could possibly transition to, as each of the remaining potential solutions could produce no more than one different coloring of g_t, and each coloring could correspond to no more than one state. We assume that at the start of the game the solutions word is chosen uniformly at random from S, so the transition probabilities are defined as follows:

$$P(S_{t+1} = K) = (\text{number of potential solutions in } S_t \text{ that would result in } S_{t+1} = K)/|S_t|$$

Finally, the reward function is where our two formulations differ; the first, which will be referred to as "guess-indifferent," has all rewards 0 for the first six epochs and reward i_7 at epoch 7, meaning we have reward 1 if the solution is guessed within the allotted guesses and 0 otherwise. The second "guess-biased" formulation, for epoch $t < 7$, current state (i_t, S_t), and action g_t, has reward $r_t((i_t, S_t), g_t) = i_t - 1$, and for t = 7 it has $r_7(i_7, S_7) = -\alpha$ if $i_7 = 0$ and $r_7(i_7, S_7) = 0$ if $i_7 = 1$, where $\alpha \geq 7$. Intuitively, this means that in the guess-indifferent formulation we only care if we correctly guess the solution within the allotted six guesses or not, whereas in the guess-biased formulation we also care about the number of guesses needed to do so, as we have a cost of 1 for each guess and an additional cost of α if we do not guess the solution at all.

We now describe the approaches to playing Wordle that we consider in this project. Ordinarily, a game formulated as a Markov Decision Process like this (with a slightly modified reward function) would be solved using back-propagation/backwards induction to find the optimal strategy. However, this approach is not computationally tractable with any computational means currently available, as it would require, for each possible ending state s_e of the game, enumerating all possible successively earlier states that could lead to the occurrence of s_e. Bertsimas and Paskov evade this issue by setting up the optimality equations (Bellman equations) for this dynamic programming and solving them directly using recursion in order to derive an exact and provably optimal strategy to play Wordle, but as they note in their paper, even this is quite computationally heavy. Thus, in this project we consider approaches to Wordle that do not have any such guarantee of optimality but are far less computationally expensive, as they optimize heuristics for approximately optimal play at each epoch rather than directly optimizing the reward function as an exact solution would.

Random and Greedy Algorithms

We begin with our naive "gold standard of badness" random algorithm. This algorithm is as simple as it can be while still playing the game in a somewhat intelligent way; at each stage t, it simply chooses a guess g_t uniformly at random from the list of remaining potential solutions S_t. As we assume each element of S_t is equally likely to be the solution, this algorithm could be called greedy in some sense because at each epoch t it maximizes the probability of our guess g_t being correct (as every element $s \in S_t$ has probability $1/|S_t|$ of being the solution), and it does so in the simplest way possible. This algorithm has time complexity $\mathcal{O}(|S|)$, as at each epoch t the algorithm does two things: compute S_t using the elimination algorithm with parameters S_{t-1}, g_{t-1}, and the coloring of g_{t-1}, in order to observe the current state (time complexity linear in the cardinality of S_{t-1}), then choose a random guess g_t from S_t (constant time complexity). Thus its runtime is linear in the cardinality of S.

We then consider two more complex greedy algorithms. The first is a MinMax algorithm that, at each stage t, attempts to minimize the maximum size of the potential solutions set at the start of the next stage t+1, and the other (which we call the MinEx algorithm) attempts to minimize the expected size of the potential solutions set at the start of the next stage t+1. However, to do either of these exactly is quite a computationally heavy task; at each stage t, these would require considering each of the $|G|$ potential guesses, conditioning on each of the $|S_t|$ solutions, computing the coloring of each guess with each solution, and computing the associated S_{t+1} that would result from each one of these $|G||S_t|$ possibilities. As this is far too computationally heavy for simple greedy algorithms that only minimize heuristics, we instead take a randomized approach to these algorithms; at each stage t, we consider $|S_t|/100$ random guesses from S_t (not G) and condition each of these on $|S_t|/100$ potential solutions from S_t.

Both of these greedy algorithms have time complexity $\mathcal{O}(|S|^3)$, as they must each do two things at each epoch t: compute S_t using the elimination algorithm as described above (constant time complexity), then compute a guess g_t. For this latter step, these algorithms must, for each of $|S_t|/100$ random candidate guesses g_{tc}, condition on $|S_t|/100$ random remaining solutions s_{tr}, use the elimination algorithm to compute the S_{t+1} that would follow if we guess g_{tc} and s_{tr} is indeed the solution, then find the candidate guess with the smallest maximum/average cardinality of the computed sets S_{t+1}. As noted above, the elimination algorithm is $\mathcal{O}(|S|)$, and it must be run for each guess-solution pair, of which there are $(|S_t|/100)^2$, resulting in a time complexity of $\mathcal{O}(|S|^3)$. The last step finds the maximum/average cardinality of S_{t+1} for

each of $(|S_t|/100)^2$ guess-solution pairs, which is only $\mathcal{O}(|S|^2)$, so it does not increase the time complexity of the algorithms, so both of these algorithms have cubic time complexity $\mathcal{O}(|S|^3)$. We note that both algorithms would still have cubic time complexity $\mathcal{O}(|G||S|^2)$ even if we considered every guess from G and conditioned on every solution in S_t at each epoch t, but for practical use their actual runtime is much shorter with restricting guesses to S_t and using the described random sampling.

Rank-One Approximation and Latent Semantic Indexing

We then consider Michael Bonthron's approach of using matrices to represent the set of solutions (called the solution space) and the set of actions (called the action space), then conducting latent semantic indexing on dominant eigenvectors to use as guesses. Bonthron proposes converting each word into a unique column vector of length 130, where components $26(k-1)+1$ through $26k$ represent the k^{th} letter of the word, and the j^{th} of those 26 components is 1 if the k^{th} letter of the word is the j^{th} letter of the alphabet and 0 otherwise. Each column vector is concatenated into a matrix of dimension $130 \times n$ where n is the number of words in a given space. For example, if we were dealing with two letter words, the two-letter word AZ would be represented as the following vector in $\mathbb{R}^{52}$:

$$AZ := \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

Finding the Best Word

In the way described above, we represent the set of possible guesses G now as a matrix G, and the set of potential solutions S as a matrix S. In his paper, Bonthron assumes those two sets (and thus the matrices) to be the same, meaning that in his investigation every word that can be accepted as a guess could also be the solution at the start of the game. Thus, he proceeds to find the column vector that "best represents" G. To do this, he finds the dominant eigenvalue (the largest magnitude eigenvalue) λ_{dom} associated with GG^T and then finds the dominant eigenvector u associated with that eigenvalue. Why does this work? Consider the singular value decomposition $G = U\Sigma V^T$, where Σ is a diagonal matrix whose diagonal entries are the singular values of G in descending order. The closest rank-one approximation to G is $u_1\sigma_1{v_1}^T$, where $\sigma_1 = \Sigma_{1,1}$ and u_1 and v_1 are the associated left and right singular vectors. Therefore, u_1 can be considered the column vector that"best represents" G. σ_1 is the largest singular value of G, so ${\sigma_1}^2$ must be the largest magnitude eigenvalue λ_{dom} of GG^T, with associated eigenvector u_1, as $GG^T = U\Sigma^2U^T$ is an eigendecomposition. Thus, the dominant eigenvector u of GG^T is exactly the left singular vector of G associated with singular value σ_1, which is also the column vector u_1 that best represents G. Because we have $S \subset G$, and we want our guesses to be good representations of the solution space S, not the action space G, we deviate from Bonthron's approach by calculating this dominant eigenvector for the matrix SS^T rather than GG^T.

Once the most representative column vector u is calculated, latent semantic indexing is used to find the closest possible guess word. In particular, we use cosine similarity as follows to find the word in our action space G to the dominant eigenvector u of SS^T:

$$g* = \arg\min_g\{\theta = \arccos(\frac{u^T * g}{||g|| * ||u||})\}$$

The word $g*$ with the smallest θ is the closest possible guess word to the most representative column vector u of the solution space S.

Therefore, at each epoch t, we start by converting S_t into a matrix and computing u_t, the dominant eigenvector of $S_t S_t^T$ and the first singular vector of S_t. Then we choose our guess g_t* as the word associated with the smallest value of θ as done above, using $u = u_t$, and if there is a tie we choose the tying word that comes first in alphabetical order. Then at the start of the next epoch $t+1$, we use the information learned from the guess g_t and its coloring to reduce the solution space from S_t to S_{t+1} before finding the dominant eigenvector for the matrix $S_{t+1}{S_{t+1}}^T$ and its closest element of the action space.

This algorithm has cubic time complexity $\mathcal{O}(|S|^3)$, because at each epoch t it must do the following: convert S_t into a matrix (linear time $\mathcal{O}(|S_t|)$), find the dominant eigenvector u of $S_t {S_t}^T$ (cubic time $\mathcal{O}(|S_t|^3)$), use cosine similarity to find $g* \in G$ closest to u (linear time $\mathcal{O}(|S_t|)$), and finally run the elimination algorithm (linear time $\mathcal{O}(|S_t|)$). Thus the highest time complexity operation runs in $\mathcal{O}(|S_t|^3)$ time, so $\mathcal{O}(|S|^3)$ is the overall time complexity of this algorithm.

Principal Component Analysis and 90 - Cosine Similarity

This approach is very similar to the above approach with rank-one approximation and latent semantic indexing. Once again we begin by creating a matrix G and matrix S in the same manner as above.

Finding the Best Word

At each epoch t, we perform singular value decomposition of the matrix S_t to find the best rank-one approximations for S_t, where $R_k = u_k \sigma_k v_k^T$ is the k^{th} best rank-one approximation of S_t. Based on the plots in Image 1 in the Appendix, showing percent variance explained by each principal component over several different trials, we found that the percent variance explained seems to drop off significantly after the third best rank-one approximation; thus, at each epoch t, we decided to only consider the best three rank-one approximations for each S_t, as further approximations would seem to have a much smaller chance at being good fits for the best word. We then calculate the proportions of variance explained by each of the first three rank-one approximations R_1, R_2, and R_3 and label these proportions p_1, p_2 and p_3 respectively. Then we select the first column vectors of R_1, R_2 and R_3, which we call r_1, r_2, and r_3, and find the closest words from our action space G as follows, where g_{t_k} is the closest guess word to r_k:

$$g_{t_k} = \arg\min_g \{\theta_k = \arccos(\frac{r_k^T * g}{||g|| * ||r_k||})\} \; \forall k \in \{1, 2, 3\}$$

We store these minimum angles as θ_1, θ_2 and θ_3, and then create three scores to evaluate "guess quality" by calculating $s_k = p_k(90 - \theta_k)$. As p_k is the proportion of variance of variance explained by the k^{th} singular value decomposition, where a higher value means r_k is more representative of S_t, and $(90 - \theta_k)$ is a measure of how close the closest guess g_{t_k} is to r_k, where a higher value means that it is closer, this score s_k serves as a way to balance our consideration of both of these important factors. Thus, we choose the word g_{t_k} with the highest score s_k to be our guess, then make the guess and use our elimination algorithm to eliminate any words that are no longer possible solutions before moving onto the next epoch.

The only differences between this algorithm and the previous one are that this one considers the first column of each of the best three rank-one approximations of S_t rather than just the first left singular vector of S_t, and that this one finds and compares the guesses maximizing cosine similarity for all three of those vectors then multiplies it by a percent variance explained metric to find the best guess rather than just guessing the word closest to the dominant eigenvector. This process still has cubic time complexity $\mathcal{O}(|S|^3)$, since finding the singular value decomposition of matrix S_t is $\mathcal{O}(|S_t|^3)$, but because it essentially runs the bulk of the previous algorithm three times at each epoch, it should be noted that this algorithm should run slightly less than three times slower than the previous algorithm.

Because we will compare specific execution times later, we note here that all of these algorithms were implemented in Python for testing, most likely with some inefficiencies that slow them down somewhat but do not drastically increase their runtime. All of the code for these implementations can be found on GitHub at Link 1 in the Appendix.

Results

We begin with our "gold standard of badness" random algorithm. We ran this algorithm once with each element of the potential solutions set, and it had an average runtime of 0.008 seconds per game of Wordle. It correctly guessed the solution in 2253 of these 2315 trials, for a success rate of 0.973, meaning that the experimental average reward of this algorithm in the guess-indifferent formulation would be 0.973. In the games where it guessed the solution correctly, it took an average of 4.08 guesses to do so, and we see that it failed to correctly guess the solution 62 out of 2315 times, meaning that in the guess-biased formulation its experimental average reward would be $-(4.08 + \frac{62}{2315}\alpha) \leq -4.27$ as we have $\alpha \geq 7$.

We now move onto the non-deterministic greedy algorithms that, at each decision epoch, attempted to minimize the heuristics for optimality that are maximum and expected cardinality of the remaining solution set. The MinEx algorithm correctly guessed the solution in 2271 out of 2315 trials, taking an average of 3.93 guesses in the games in which it was successful. This gives it average experimental rewards per game of 0.981 in the guess-indifferent formulation and $-(3.93 + \frac{44}{2315}\alpha) \leq -4.06$ in the guess-biased formulation. The MinMax algorithm saw similar results, with average experimental rewards of 0.984 in the guess-indifferent formulation and $-(3.92 + \frac{38}{2315}\alpha) \leq -4.03$ in the guess-biased formulation. The average execution times per game were 3.45 seconds for both the MinEx and MinMax algorithms.

Before proceeding to our other results, we remark here that it is important to consider the results of these three approaches that we have tested so far in the context that they are non-deterministic algorithms. Even if we fix a given solution word at the start of the game, these algorithms can play the game differently in different trials; thus, it is not possible to experimentally find these approaches' true mean rewards just by running one trial of the game for each solution word, and the results that we have observed are experimental and have no guarantee of being exactly correct.

Next we examine the results of our modified implementation of Bonthron's rank-one approximation and latent semantic indexing algorithm. Because this is a deterministic algorithm, it will make the same first guess in every game, and in initial testing we found that this guess is SAINE. Thus, in our code, rather than computing the dominant eigenvector of SS^T every time the game is played, we hard-coded this algorithm to always guess SAINE as the first guess

to cut down on computation time. After this first guess, the algorithm proceeds as described in the Methods section for all other guesses. With this small modification, the algorithm had an average execution time of 0.996 seconds per game in our trials. In terms of performance, it correctly guessed the solution in 1784 out of 2315 trials, giving it an experimental average reward of 0.771 in the guess-indifferent formulation, and in those 1784 successful trials it took an average of 3.72 attempts to guess the solution, resulting in an experimental average reward of $-(3.72 + \frac{531}{2315}\alpha) \leq -5.33$ in the guess-biased formulation.

Finally, we review our results with our modified version of Bonthron's approach implementing guess-scoring with principal component analysis and 90 - cosine similarity. This deterministic approach also always chooses SAINE as the first guess, so we once again hard-coded this first guess in every game and do the full computation for the remaining guesses. This algorithm guessed the solution correctly in 2290 out of our 2315 trials, so it had an experimental guess-indifferent reward of 0.989. In the successful trials it took an average of 3.79 attempts to guess the solution, giving it an experimental guess-biased reward of $-(3.79 + \frac{25}{2315}\alpha) \leq -3.87$.

We note that these past two algorithms are deterministic, and we have assumed in our Markov Decision Process formulation that, in each game, a word from the solutions set is chosen uniformly at random. Thus, simply running the game once with each element of S being the solution and taking the arithmetic means of the rewards in our trials gives the true mean rewards for these algorithms. Therefore, the rewards stated above for these two approaches are not just experimental approximations but in fact exactly correct.

For comparison, Bertsimas and Paskov's dynamic programming solution guarantees a reward of 1 in the guess-indifferent formulation, and has an average reward of -3.421 in the guess-biased formulation. Although we lack the computational resources to implement their methods to exactly solve Wordle ourselves, Bertsimas and Paskov note in their paper that it took days to execute solving the game using an efficient C++ implementation of their algorithm parallelized across a 64-core computer, and therefore as a lower bound we estimate that it would take at least several weeks with a somewhat inefficient Python implementation running on a college student's laptop, as would be the case if we attempted this.

Table 1 in the Appendix displays these results in tabular form.

Conclusions

In this project we compared attempts at near-optimal play in Wordle that varied widely in their approaches at finding effective guesses at each step. Surprisingly, a very simplistic random algorithm that simply chose a random potential solution at each stage performed much better than expected, correctly guessing the solution in 97.3% of trials. The greedy algorithms had slightly better performance, but they do run over 400 times slower than the random algorithm, which makes it somewhat questionable whether they should really be favored over the random algorithm for practical use, although the runtimes are still fairly reasonable.

Our implementation of Bonthron's approach actually performed even worse than our random algorithm, both in terms of runtime and performance. We suspect that the main issue that caused this is that, as we noticed in our testing, this algorithm often gets stuck on a certain guess i.e. even after making a guess g_t and adjusting the solution space for epoch $t+1$ accordingly, the closest word to the dominant eigenvector of $S_{t+1}S_{t+1}^T$ is sometimes still the same

word, meaning we get $g_{t+1} = g_t$. Because guessing the same word again will not result in any further changes to the solution space, we end up with $S_{t+2} = S_{t+1}$, and thus $g_{t+2} = g_{t+1}$ and so on, where this approach will continue to guess this same word for the rest of the game, meaning that in all cases where we see this phenomenon, this approach will fail to guess the solution.

Our modified implementation of Bonthron's algorithm with principal component analysis for guess-scoring performed the best. We believe this is because it largely addressed the issue with our original implementation of Bonthron's approach, and it also generally took into account the fact that the closest guess to the first singular vector might actually still be quite far away from that vector, and it may be more representative of the solution matrix to choose the closest guess to the second or third singular vector because that guess may be closer to the vector itself. As this algorithm saw the best reward in both of our formulations of the game, and also had quite a reasonable runtime, less than 230 times slower than the random algorithm, we recommend using our modified Bonthron approach for the best results.

Future attempts at effective and efficient play in Wordle could try more modifications to the Bonthron approach, such as forcing the algorithm to guess a word that is actually in the solution space for the last one or two attempts, because occasionally we observed that even on the last guess these approaches would guess a word not in S, which has no chance of being correct and thus worsens the expected reward. Additionally, although we did not explore this direction in our project, future attempts could also try an approach implementing principles of information theory to use the first few guesses to amass the most information possible about the solution word before using the remaining guesses to try to correctly guess the solution.

References

Bertsimas, D., & Paskov, A. (2022). An Exact and Interpretable Solution to Wordle.

Bonthron, M. (2022). Rank One Approximation as a Strategy for Wordle.

Cruise, B. (2022, January 14). Wordle valid words. Kaggle.

Pratusevich, M. 1. (2022, February 12). A Python Wordle Clone. Practice python. https://www.practicepython.org/blog/2022/02/12/wordle.html

Tracy, M. (2022, January 31). The New York Times Buys Wordle. *The New York Times*.

Appendix

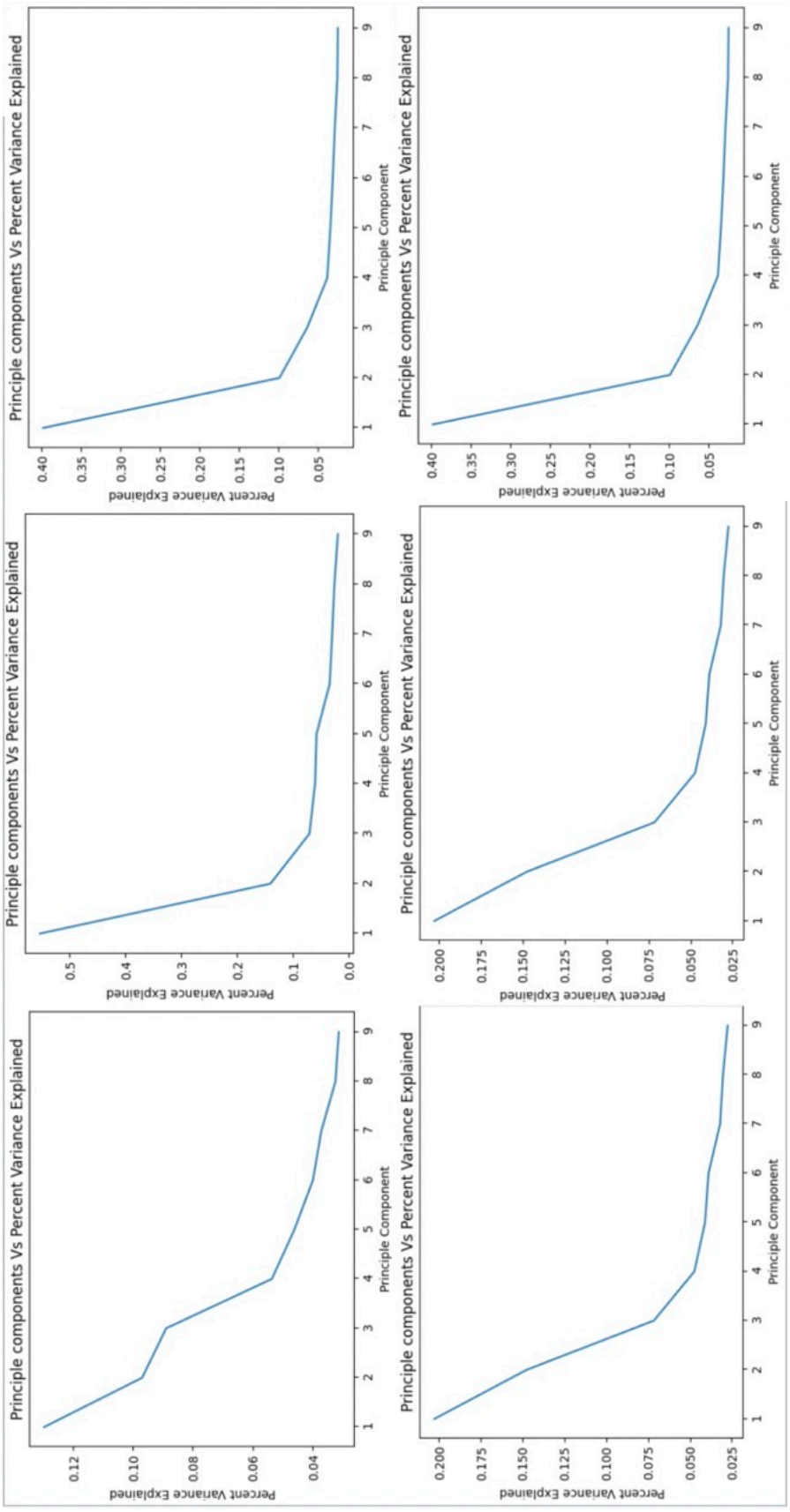

Image 1: Different Iterations of Percent Variance Explained by Each Principal Component

GitHub - vtumulur - Wordle Project

Link 1

Method	Success Rate	Avg. No. of Attempts	Avg. Guess Biased Reward	Avg. Game Run Time
Random	.973	4.08	$-(4.08 + \frac{62}{2315}\alpha)$	0.008 sec
MinMax	.984	3.92	$-(3.92 + \frac{38}{2315}\alpha)$	3.45 sec
MinEx	.981	3.93	$-(3.93 + \frac{44}{2315}\alpha)$	3.45 sec
Rank-One Approximation and Latent Semantic Indexing	.771	3.72	$-(3.72 + \frac{531}{2315}\alpha)$	0.938 sec
Principal Component Analysis and 90 - Cosine Similarity	.989	3.79	$-(3.79 + \frac{25}{2315}\alpha)$	1.78 sec

Table 1: Performance and Runtime