Implementing and Evaluating Entropy and Simulation based Algorithms for Wordle using Deterministic POMDP Modeling

Bachelor's thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence Research Group
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Tanja Schindler

Floyd Peiszan
floyd.peiszan@unibas.ch
20-057-428

13.01.2025

Acknowledgments

First and foremost, I would like to thank my supervisor, Tanja Schindler, for accompanying me throughout the whole thesis, helping me with any subject whenever I needed it, and guiding me in the right direction. I want to further thank Florian Pommerening for helping me set up the sciCORE environment to run my experiments. Finally, I extend my gratitude to Prof. Malte Helmert for giving me the opportunity to work on my Bachelor's thesis in the AI research group of the University of Basel.

Use of AI Tools

For the report on my bachelor's thesis, I used Grammarly[1] as an automated tool for spelling checks, synonym suggestions, and grammatical corrections regarding punctuation. No new content was created by an AI tool. I have reviewed all the suggestions and take full responsibility for the result.

[1] Available at: https://app.grammarly.com/.

Abstract

This thesis aims to develop algorithms to optimize strategies for solving Wordle, a word guessing game in which the player has to find a hidden five-letter word in at most six guesses by evaluating feedback on guessed words. We formalize Wordle with a DET-POMDP model as appropriate for environments with uncertainty. We implemented three algorithms built on top of each other to evaluate their accuracy. Our presented entropy based heuristic solves a given Wordle game in 3.43 guesses on average, which is within 1% of the optimal score, and forms the foundation for more advanced simulation based methods. On the hand, this includes rollout methodologies that simulate the heuristic, seeking to improve the average heuristic scores. Our experiments show that these methodologies only result in marginal improvements at the cost of significantly higher computing costs. On the other hand, the application of UCT guiding for Monte Carlo Tree Search allows us to produce a theoretically optimal solution with vast limitations on practical usage.

Table of Contents

1 Introduction

After Josh Wardle released the word guessing game Wordle in October 2021, it quickly gained widespread attention for its simple, yet engaging gameplay. The goal is to guess a hidden five-letter word. As seen in Figure 1.1, the player has six guesses to come to the right conclusion. On each guess, the game returns feedback that reveals color-coded hints about the solution word. Green tiles indicate correct letters in the correct positions, yellow tiles describe correct letters in the wrong position, and gray tiles mark incorrect letters.

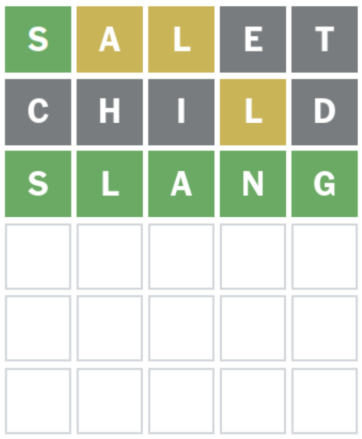


Figure 1.1: Example of a game of Wordle: It took the player three guesses to arrive at the solution word 'SLANG'. Source: New York Times Wordle[2].

Wordle has an easy mode and a hard mode. The above-discussed rules apply to the easy mode. In hard mode, only words using all hints from the previous feedback are valid guesses: If a letter is marked as correct, each subsequent guess has to contain this letter in the correct position. In the example of Figure 1.1 each guess needs to start with the letter 'S' after the feedback for the first guess is received. Likewise, a letter marked as yellow has to appear in

[2] Available at: https://www.nytimes.com/games/wordle/index.html [11 January 2025]

each subsequent guess. Additionally, all incorrect letters can no longer be used. The guess 'CHILD' would therefore not be allowed in hard mode because it does not have an 'S' as the first letter and also does not contain an 'A'. In this thesis, we will focus on the easy mode, but all algorithms can be adapted to hard mode.

There are predefined lists of words that are considered valid guesses or possible solutions. These lists are managed by the New York Times and get slightly modified over time. The lists can be accessed using the developer tools of an internet browser on the Webpage[3] where the game is accessible to the public. As of December 2024, these lists contain 14855 guess and 2309 solution words. To guarantee comparability, this thesis will use legacy word lists used by many researchers that contain 12972 guess words and 2315 solution words. Naturally, each possible solution word is also contained in the guess word list.

The simple nature of the game sparks research interest, as it is not easy to solve computationally. Although it is straightforward to understand and play, there is a huge amount of possible game trees. Various approaches have been applied to solve the game. Researchers applied different heuristic evaluations [2], [4], [8], and some tried to solve the game exactly using dynamic programming [1]. Some of this work will be discussed in Chapter 3.

This thesis aims to derive a theoretical model of Wordle and to design and implement an algorithm to solve the game. In this context, solving refers to minimizing the number of guesses on average and identifying the best opening guess. For each Wordle instance, the algorithm has to produce a policy to find the solution where policy means a sequence of guesses to the Wordle game.

The main computational challenges lie in the large search space due to the high number of possible word combinations. This is asking for an efficient algorithm and possibly some precomputation. Another challenge is handling the uncertainty of each guess. There is no trivial way of measuring how good a guess is before guessing it, as feedback is only received after the decision has been made. This makes the game of Wordle only partially observable. Accordingly, decisions on which guess to choose next must be made on the spot, and it is impossible to compute a policy before the game. To create a mathematical model we use deterministic partially observable Markov decision processes. This allows us to apply algorithms from literature with known characteristics. More about this will be discussed in Chapter 2.

To derive an algorithm, we use a maximum information gain heuristic, that aims to reduce the amount of uncertainty after a guess as much as possible. We will then use this heuristic as a base for different rollout approaches that aim to reduce the number of guesses needed to find the solution word by repeatedly simulating the heuristic. We will present a straightforward rollout algorithm that evaluates potential guess words by simulating the heuristic across all possible solution words. Additionally, we will introduce a rollout algorithm leveraging the so-called UCT formula to explore all possible game scenarios, given sufficient rollouts, to achieve an optimal score. This formula helps guide the algorithm to guarantee the exploration of all game tree nodes while focussing on promising branches. Detailed explanations of these algorithms will be provided in Chapter 4.

[3] Available at: https://www.nytimes.com/games/wordle/index.html [11 January 2025]

2 Modeling Wordle

In this section, we provide a formal framework that allows us to describe the algorithms developed for Wordle. First, we define key terms to discuss the algorithms more precisely. We then present a widely known mathematical model for decision problems under uncertainty and adapt it specifically for Wordle. This formal model will enable us to use algorithms from the literature.

2.1 Key Concepts and Terminology in Wordle Modeling

In this section, we define key terms and concepts that are fundamental to understanding the modeling of Wordle. These definitions establish a common language for the subsequent analysis.

Definition 1. ***Solution****: In the game of Wordle, the* **solution** *refers to the hidden word that the player aims to guess. It is the correct word chosen from the solution word list, and the objective of the game is to identify this word by making valid guesses and interpreting the feedback provided after each guess.*

Definition 2. ***Guess****: In the game of Wordle, a* **guess** *refers to a word that the player is allowed to guess at any given stage of the game. In easy mode, all words from the guess word list are valid guesses at any time. In hard mode, only words that align with the feedback from previous guesses are allowed.*

Definition 3. ***Policy****: In the context of Wordle, a* **policy** *refers to the sequence or chain of guesses that an algorithm chooses in order to identify the solution word. The policy determines the algorithm's progression from one guess to the next, based on the feedback received. An* ***optimal policy*** *is a* **policy** *that yields the best possible average score across all Wordle games.*

With this, the goal of an algorithm is to produce a policy that minimizes the amount of guess attempts necessary to find the solution.

2.2 Deterministic Partially Observable Markov Decision Processes

In order to formalize the game of Wordle, we need a sequential decision-making model. The model should allow the generation of policies for the decision-making problem. A deterministic model is not sufficient, since the solution in Wordle is initially unknown. Deterministic Partially Observable Markov Decision Processes (DET-POMDPs) are a more expressive formalism by allowing to model problems where not all information is available from the beginning. DET-POMDPs were first introduced by Littman [7] in his doctoral thesis. They are mathematical models to solve decision-making problems where the environment is only partially observable. Here, uncertainty derives from incomplete initial knowledge rather than randomness. In the case of Wordle, the only unknown variable is the solution since we can only know it after receiving feedback sufficient to rule out all the other words from the solution list.

2.2.1 Formal model of DET-POMDPs

A general model of DET-POMDPS can be found in Bonet [3]. We will use their definition as a base to build a mathematical model of Wordle.

Definition 4. *A formal definition of a DET-POMDP as found in Bonet [3].*

- ***State Space:*** *A finite state space $S = \{1, \ldots, n\}$.*
- ***Actions:*** *Finite sets of applicable actions $A_i \subseteq A$ for each state $i \in S$.*
- ***Observations:*** *A finite set of observations O.*
- ***Initial State:*** *An initial subset of states (belief state) $b_0 \subseteq S$, or alternatively an initial distribution of states $b_0 \in \Delta S$.*
- ***Goal States:*** *A subset $T \subseteq S$ representing the goal states.*
- ***Transition Function:*** *A deterministic transition function $f(i, a) \in S$, for $i \in S, a \in A_i$, specifying the result of applying action a to state i.*
- ***Observation Function:*** *A deterministic observation function $o(i, a)$, for $i \in S, a \in A$, specifying the observation received when entering state i after applying action a.*
- ***Costs:*** *Positive costs $c(i, a)$, for $i \in S, a \in A_i$, specifying the cost of applying action a in state i.*

This definition incorporates the uncertainty from incomplete initial knowledge in the initial belief state b_0. This is necessary as all future states would otherwise be known as the transition and observation functions are deterministic.

2.3 Wordle modeled as a DET-POMDP

To model Wordle, we need to adjust the above definition of DET-POMDPs. The hidden information is of course the solution word. The hidden information will be incorporated into the observation function. This function is only deterministic within a single game of Wordle rather than across all instances of the game as it changes behavior according to the hidden word. We can only ever know the true value of the observation function after taking an action.

Definition 5. *The Wordle game can formally be described as follows:*

- ***State:*** *A state $s \in S$ is defined as the set of words from the solution list that is still consistent with the feedback from all previous guesses. The state $s_i \in S$ represents the state of the game after i guesses with $0 \leq i \leq 6$, A goal word $g \in s_i$ is called a possible solution word at state s_i.*
- ***Action:*** *An action $a \in A$ is a valid guess word from the dictionary. The action a_{s_i} is the action applied in the state s_i and leads to the state s_{i+1} when applied in the transition function.*
- ***Observation:*** *An observation $o_i \in O$ is defined as the feedback Wordle returns on an action $a_{s_i} \in A$:*
$$O = \{(f_1, f_2, f_3, f_4, f_5) \mid f_x \in \{g, y, b\}\}$$
where g: green, y: yellow, and b: grey.
- ***Initial Belief State:*** *The initial belief state $b_0 = s_0$ is defined as the set of all possible words from the solution word list.*
- ***Goal State:*** *A goal state is defined as a state $s_i \in S$ that contains only one word, and this word is the last action a_{i-1}.*
- ***Transition Function:*** *The transition function $f(s_i, a_{s_i})$ removes each word from state s_i that is not consistent with the observation (feedback) received when taking action a_{s_i}.*
- ***Observation Function:*** *The observation function $o(s_i, a_{s_i})$ generates observations (feedback) for a guess based on the true solution word.*
- ***Costs:*** *The costs are defined as $c(s_i, a_{s_i}) = 1$ for each action $a_{s_i} \in A$.*

This definition simplifies the process of tracking the game. At any stage, there is direct access to the set of possible solution words. This set is used by multiple algorithms discussed in Chapter 4. To maintain good readability we will use the term ‘guess (word)’ interchangeably with the term ‘action’ and refer to the goal words $g \in s_i$ as ‘possible solution (words)’.

3 Related Work

In this section, we will present work from different authors from the literature. We illustrate a variety of algorithms that aim to find a policy for Wordle without conducting an exhaustive search of the whole game tree as well as one approach that yields an optimal policy. In Chapter 3.3 we explore related problems and elaborate on the application of the used strategies for Wordle.

3.1 Heuristic Algorithms

In the context of solving Wordle, many approaches have been applied. Numerous researchers employ heuristic methodologies to estimate the best policy. A heuristic is a problem-solving approach that uses practical, rule-of-thumb strategies or shortcuts to find solutions quickly, often at the cost of completeness or optimality. It would be beyond the scope of this thesis to discuss them all. This section will focus on one heuristic that yielded good results in the past and explore an interesting way of improving the performance of heuristic algorithms.

3.1.1 Entropy based Algorithms

Some authors use entropy to make the next guess in Wordle. In this context, entropy measures the information a guess can reveal about the solution. A higher entropy guess can potentially eliminate more possibilities, helping to narrow down the solution space. In 2022 Sanderson [8] released a video on his YouTube channel '3Blue1Brown' where he discusses an information theoretic approach to Wordle. His algorithm computes the entropy for a guess word by looking at the feedback the guess word could generate across all solution word candidates. The next guess is chosen based on a combination of the highest entropy value and relative word frequencies in the English language. The word frequencies are used to determine whether a guess can be a solution as his approach does not use the list of possible solutions and treats each guess word as a solution candidate. Even without incorporating this list, the algorithm gets an average score of 3.601 with 'CRANE' as the opening guess. By utilizing the predefined answer list and computing the entropy value two guesses in advance, the algorithm gets an average score of 3.433 with 'SALET' as the opening guess.

A simple one-step entropy-based approach will form the basis of our implementation and serve as a guideline for more advanced methodologies.

3.1.2 Online Rollout

To improve the performance of heuristic algorithms, Bhambri et al. [2] introduced an online rollout approach. At any stage of the game, they choose the next guess as follows. They take a base heuristic and simulate the game for every action/goal pair. The next guess is the action word that gets the lowest average score. They were able to improve the performance of the maximum information gain heuristic from 3.6108 to 3.4345 with ‘SALET’ as the opening guess. This is within 0.4% of the optimal score shown by Bertsimas and Paskov [1] discussed in the next section. We implemented a similar approach but could not get comparably good results using this methodology. More details will be provided in Chapters 4 and 6.

3.2 Exact Algorithms

Solving Wordle using an exact algorithm poses an immense challenge. The main obstacle is the size of the full game tree with the number of possible states being $2^{2315} \sim 10^{697}$, which makes it unfeasible to compute. This section briefly discusses a clever approach to get an optimal solution.

3.2.1 Dynamic programming approach

In 2022, Bertsimas and Paskov [1] have found an optimal policy for Wordle. They precompute the best action for each possible state by simulating all state/action pairs. They were able to achieve this by using recursive algorithms and a lot of optimizations. Some shortcuts were used for specific states and state space sizes as well as pruning sub-optimal branches early by introducing a lower bound on the amount of guesses for a specific guess word at an arbitrary state. Despite these improvements, they claim that their efficient parallelized C++ implementation still took several days to solve on a 64-core computer. To maintain readability they use decision trees to represent the precomputed actions. Their conclusion shows that ‘SALET’ is the optimal opening guess with an average score of 3.42177 and at most 5 tries to guess the correct answer. They made their algorithm available through a Webpage[4]. We will use their findings to compare our algorithms to the scores that an optimal policy would find.

3.3 Related problems

There are a lot of computational problems similar to Wordle. An example played in real life would be Mastermind. It is a code-breaking game where one player creates a secret sequence of colored pegs, and the other player tries to guess it within a limited number of

[4] Available at: https://www.wordleopt.com [11 January 2025]

attempts. After each guess, the code-maker provides feedback to indicate the correctness of colors. There are countless strategies to solve Mastermind, however, the search space is significantly lower than for Wordle making it hard to adapt Mastermind strategies.
Another decision-making problem under uncertainty is the Canadian traveler's problem. It is a pathfinding problem where a traveler seeks the shortest route on a graph with uncertain edge conditions, such as snow-blocked roads. The traveler discovers the state of edges only upon arrival at a node. Eyerich et al. [5] discuss this problem and adapt a Monte Carlo search algorithm using an Upper Confidence Bound for trees (UCT) mechanism. This is a simulation based search method that selects successor nodes (the next decision) based on a trade-off between the exploration of rarely visited paths and the exploitation of parts that have produced good outcomes in the past. Balancing using UCT allows for optimality in the limit. In practice, Eyerich et al. could not get great scores by only relying on the UCT formula but were able to optimize it with heuristic evaluations to guide early simulations to more promising parts of the game tree, achieving scores better than most policies. We will adapt basic UCT mechanisms to Wordle and derive an algorithm that theoretically will produce an optimal policy given enough resources.

4 Algorithmic concepts of solving Wordle

This section will discuss the implemented algorithmic approaches to produce a policy for Wordle, aiming to optimize guess efficiency and solution accuracy. The maximum information gain heuristic drives to estimate the next best guess and forms the basis for further approaches. The basic rollout approach tries to improve upon a heuristic evaluation by using it to simulate many games to assess the next guess. The UCT approach aims to provide a way of computing an optimal policy given enough resources while also exploiting the heuristic.

4.1 Maximum information gain heuristic

A good guess in Wordle is inherently one that eliminates the most possible solution words. In this section, we focus on a way of measuring how well a guess would perform at any stage of the game. To do so in an information-theoretic manner, we need to quantify the amount of information a guess would return. Intuitively the more information a guess provides, the more we can narrow down the space of possible solution words. The amount of information can be evaluated using entropy. The concept of entropy was first introduced by Shannon [9] in 1948.

Definition 6. ***Entropy****: In information theory,* ***entropy*** *measures the amount of uncertainty or randomness in a system associated with a random variable. It is typically measured in bits (logarithm base 2).*

In a simple case, where each outcome is equally likely, an entropy value of 3 bits would correspond to a situation with $2^3 = 8$ outcomes. The logarithmic nature of entropy allows for simple addition or subtraction. Starting with this entropy value of 3 and $2^3 = 8$ equally likely outcomes, reducing the entropy by 1 would halve the number of outcomes, leaving $2^2 = 4$. Reducing the entropy by 2 would halve the outcomes again to $2^1 = 2$.
It is generally assumed, that the solution word for Wordle is chosen uniformly across all possible solution words, allowing us to apply entropy to measure the uncertainty at any stage of the game. With a list of 2315 possible solution words, the initial entropy of a wordle game is $\log_2 2315 \approx 11.18$.

The maximum information gain heuristic uses the average entropy reduction a guess word would yield across all possible solution words. A higher average entropy reduction will result in a greater reduction of the possible solution space size. If we guess a word associated with an average entropy reduction of 4, the entropy of the game would reduce on average by 4 leaving $11.18-4=7.18$. After such a guess we have on average $2^{7.18} \approx 145$ possible solution words left. This corresponds to cutting the solution space of 2315 in half 4 times.
In the case of Wordle, mathematically the entropy reduction is calculated as follows:

$$H(a_i) = -\sum_i P(o_i)\log_2 P(o_i) \tag{4.1}$$

Where $H(a_i)$ is the average entropy reduction of action a_i and $P(o_i)$ is the probability of a possible observation o_i. A possible observation is the distinct feedback a guess receives on a specific solution word. The probability $P(o_i)$ is the ratio of the count of o_i obtained across all still possible solution words and the number of those still possible solution words.

$$P(o_i) = \frac{|o_i|}{|s_i|} \tag{4.2}$$

where $|o_i|$ is the number of occurrences of o_i and $|s_i|$ is the size of the current possible solution space.
The maximum information gain heuristic aims to maximize the entropy reduction across all guess words. This ensures that on average the next guess is as informative as possible. We apply two different forms of tie breaking when there are two guess words with the same average entropy reduction. The first version sorts words alphabetically. We call this version MIG_A. The second version prioritizes guess words that are also possible solution words before reverting to alphabetical sorting. We will call this version MIG_B. In practice, the latter version performs much better. Detailed evaluations will be discussed in Chapter 6.

4.1.1 Implementation

This section will discuss some of the main challenges of implementing the maximum information gain heuristic. We will give an overview of data structures and algorithmic concepts. The programming language of choice is Java to balance memory handling simplicity and performance.
To decide the next guess, the entropy reduction needs to be calculated for every possible guess word in the action space A. Note that in hard mode the action space will be reduced after each guess. In the following, we will describe how the entropy reduction for a single guess word is calculated. To apply equation (4.1) we need to obtain the probability distributions $P(o_i)$ of the observations (feedbacks) across all possible solution words. This can only be done by counting the occurrences of each distinct observation. The pseudo-code for Algorithm 1 visualizes this.
To avoid expensive String computations the words are represented by their integer indices in the word lists. Since there can only be three different characters in the feedback, it is encoded as an integer in base-3. The feedbacks are computed once before the main algorithm, and each later call is just an array lookup.
In line 2 Algorithm 1 initializes an array to store the counts of each distinct feedback. Since

Algorithm 1 Count Feedbacks for a guess word

```
1: Procedure CountFeedbacks(guessWord, solutionList)
2: feedbackCounts ← array of size 243 initialized to 0
3: for solution ∈ solutionList do
4:     feedback ← GetFeedback(guessWord, solution)
5:     feedbackCount[feedback] ← feedbackCount[feedback] + 1
6: end for
7: Return feedbackCounts
8: End Procedure
```

the word length is 5 the base 3 encoding allows for the small size of $3^5 = 243$. The distinct feedbacks are counted and stored in the array by the for-loop in lines 3 to 5. Algorithm 2 uses the counted feedbacks to calculate the entropy reduction of the guess word according to equation (4.1).

Algorithm 2 Compute Entropy reduction for a guess word with counted Feedbacks

```
1: Procedure ComputeEntropy(feedbackCounts)
2: entropy ← 0.0
3: inverseNumSolutions ← 1.0/legth of feedbackCounts
4: for $i$ ← 0 to length of feedbackCounts − 1 do
5:     if feedbackCounts[$i$] = 0 then
6:         continue
7:     end if
8:     probability ← feedbackCounts[$i$] × inverseNumSolutions
9:     entropy ← entropy − (probability × log(probability))
10: end for
11: Return entropy
12: End Procedure
```

It calculates the sum in lines 4 to 9. To slightly speed up the computation, the inverse of the amount of possible solution words is calculated once in line 3. Now the probability can be computed using the multiplication in line 8 rather than having to do a more expensive division. As equation (4.1) specifies the negative sum, the algorithm directly subtracts the single values in line 9.
At any stage of the game, the maximum information gain heuristic aims to maximize the average entropy reduction across all guess words in the action space A. This requires running the above algorithms for each possible action. Note that precomputing the average entropy reduction for each guess word is not feasible, as it depends on the specific set of possible solutions. With 2315 solution words, there are 2^{2315} potential subsets.
The maximum information gain heuristic is used as the basis for the following approaches and is evaluated exhaustively often. To achieve maximum speed, we parallelize the computation for the guess word associated with the highest entropy reduction. As this value needs to be calculated for each guess word individually, we can split up the list of all guess words into smaller sublists and compute the best guess word for each of these sublists parallel.

4.2 Rollout

While the maximum information gain heuristic performs well in general (see Chapter 6), there is potential for further optimization of its decision-making process to improve the average score achieved over multiple games. For this purpose, we introduce a rollout approach that builds on the existing heuristic by simulating its application multiple times across different scenarios. Bhambri et al. [2] use a similar procedure where they were able to improve the average scores of different heuristics to get within one percent of the optimal score as shown by Bertsimas and Paskov [1].

With the rollout approach, we no longer solely rely on a heuristic evaluation to choose the next guess. Instead, we select the next guess according to the average Q Factor calculated for each potential guess word. The average Q Factor describes the average score, a guess would yield. We describe how the average Q Factor for one guess word is calculated. The algorithm performs a sequence of rollouts. A rollout is carried out for each still possible solution word, conditioned on this word being the true solution to the game. A rollout is computed by guessing the current guess word and then simulating the heuristic until the solution word is reached. The average Q Factor for the guess word a at the state s_i is denoted as $Q(a_{s_i})$ and calculated as follows:

$$Q(a_{s_i}) = \frac{1}{|s_i|} \sum_{g \in s_i} h(a_{s_i}, s_i, g) \tag{4.3}$$

Where $|s_i|$ is the amount of still possible solution words and $h(a_{s_i}, s_i, g)$ is the number of guesses required by the heuristic to find the solution word g from the state s_i when guessing a_{s_i}.

The guess a_{s_i} with the lowest average Q Factor will be selected for the game. According to the simulations with the heuristic, this guess yields the best average score across all guess words. In case two guess words return the same average Q Factor the guess with the higher entropy reduction will be chosen. If the entropy reductions are also equal, a guess that is a possible solution will be prioritized. If this does not lead to tie breaking we revert to alphabetical sorting. In easy mode, we are allowed to choose any guess from the guess list containing 12972 words. To extensively reduce the computation time, each rollout will only consider n of the most promising guesses, determined by their heuristic evaluation of entropy reduction. The performance of the rollout will be discussed in Chapter 6.

4.2.1 Implementation

We will now discuss in some detail how the rollout is implemented. For simplicity, we omit the process of choosing the n most promising guesses. Algorithm 3 describes the structure of the implementation.

As we only need to find the guess with the lowest average Q Factor, it is sufficient to keep track of a single guess and associated average Q Factor. We use the variables in lines 2 and 3 for this purpose. The for-loop from line 4 computes the average score of a single guess word by simulating the game in lines 6 and 7 for every possible solution word. For the simulation, we assume the solution variable to be the true solution word to the game. The game simulation in line 7 is achieved by repeatedly guessing according to the heuristic

Algorithm 3 Find Next Guess using Rollout

```
1: Procedure FindNextGuessRollout(guessWords, solutionWords)
2: bestGuess ← $-1$
3: bestQFactor ← $\infty$
4: for guess in guessWords do
5:     score ← 0
6:     for solution in solutionWords do
7:         score ← score + SimulateGame(guess, solution)
8:     end for
9:     QFactor ← score/length of solutionWords
10:    if QFactor < bestQFactor then
11:        bestQFactor ← QFactor
12:        bestGuess ← guess
13:    end if
14: end for
15: Return bestGuess
16: End Procedure
```

and reevaluating the list of still possible solution words. Lines 10 to 12 are responsible for comparing the score of the current guess to the score of the best guess found so far. For simplicity Algorithm 3 does not do the sophisticated tie breaking discussed earlier.

4.3 Monte Carlo Tree Search with UCT guiding

The algorithms discussed so far do not have any guarantees on the quality of their policies. The last approach we explore will at least in theory solve this issue. In the previous rollout approach, the rollouts are independent of each other. With the method discussed now, each subsequent rollout depends on the outcome of the previous rollouts. We will no longer perform one rollout for each guess and possible solution word pair. A rollout for the Upper Confidence Bound (UCT) algorithm chooses a random possible solution word and simulates the game while taking into account the knowledge of guesses that performed well in previous rollouts. The number of rollouts N is a parameter of the algorithm, meaning for each decision to be made the algorithm will perform N rollouts.
The UCT algorithm was first introduced by Kocsis and Szepesvári [6] in 2006 to guide Monte Carlo planning. According to their findings, this algorithm converges to the optimal cost function (and therefore also to the optimal policy) when the number of computed rollouts converge to ∞. Monte Carlo planning refers to making decisions under uncertainty by combining the idea of random sampling (Monte Carlo methods) with planning algorithms to estimate the best course of action. It is a simulation-based process that estimates the likely outcomes of different decisions rather than relying on exact models or exhaustive search. It aims to balance exploration (trying new or uncertain actions) and exploitation (choosing actions that appear most promising based on current knowledge), i.e. the previous rollouts. A common technique for Monte Carlo planning is the use of Monte Carlo Tree Search (MCTS). The algorithm iteratively builds a search tree by repeatedly simulating random playouts. Nodes are evaluated based on the results of these simulations to select the best decision. The UCT formula (4.4) is used to balance the exploration of rarely visited paths

and the exploitation of parts that have produced good outcomes in the past. It is designed in a way to visit less promising nodes increasingly rarely over time. However, given enough rollouts, it will visit each node arbitrarily often. To describe how the next guess during a rollout is found we need to define the following:

Definition 7. ***Guess sequence:*** *A* ***guess sequence*** $\sigma = \langle s, s_1, ..., s_i \rangle$ *is the sequence of belief states that describes a partial rollout starting at s.* $R^k(\sigma)$ *is the number of rollouts among the first k rollouts that start with guess sequence* σ*.* $C^k(\sigma)$ *is the average cost (score) to complete these* $R^k(\sigma)$ *rollouts.*

The guess sequences allow us to describe a partial rollout at any stage. The number of rollouts among the first k rollouts with the average cost is necessary for the UCT formula (4.4) to decide on the next guess $a \in A$ to add to the guess sequence in that rollout. Let ρ be an unfinished guess sequence. To pick the next guess we extend ρ with $\rho_i = \langle \rho, s_i \rangle$ where s_i is the state added to the guess sequence ρ when choosing action a. The UCT algorithm maximizes the UCT formula to find the best next guess:

$$UCT(a) = B\frac{log R^k(\rho)}{R^k(\rho_i)} - cost(\rho, \rho_i) - C^k(\rho_i) \tag{4.4}$$

where $cost(\rho, \rho_i)$ is the travel cost from ρ to ρ_i. In the case of Worlde this is always 1 and therefore not considered in the actual implementation. B is a bias parameter to balance exploration and exploitation. Kocsis and Szepesvári [6] suggest choosing this parameter in a way that it grows linearly with the optimal cost. The higher the bias parameter, the more the algorithm focuses on exploration and trying less explored branches. As the optimal cost is unavailable we use the average cost of the previous k rollouts. We further divide this parameter by 5 to encourage exploitation. If $R^k(\rho_i) = 0$ the value of the formula is considered to be ∞, so that each successor gets visited once in the beginning phase.
After the specified amount of rollouts is computed, the UCT algorithm selects the next guess as the one that yielded the best score across all rollouts. Tie breaking is done as for the standard rollout. First, we consider the heuristic evaluation of entropy reduction. Then we prefer guess words that are also possible solution words. Only if this cannot make a decision we revert to alphabetical tie breaking. For Wordle, we have 12972 possible successors for each node as in easy mode it is allowed to try each guess word at any stage of the game. As a simulation mostly requires more than one guess, this translates into a huge amount of rollouts the algorithm needs to perform in order to just visit each node once. Aiming to find a good policy, the UCT algorithm needs to visit each node multiple times to get to a meaningful conclusion of its value. To reduce computation time for the experiments we make the same adaptation as for the standard rollout. We will only consider the n most promising successors for the root node of the MCTS. This comes at the cost of optimality in the limit but makes for a good comparison to the standard rollout. As the algorithm needs to perform a lot more rollouts than we can accomplish in our experiments we would not have achieved an optimal policy either way.
We experimented with optimizations to guide earlier rollouts to more promising parts of the search tree. Eyerich et al. [5] use a heuristic evaluation to assess costs for unvisited nodes. This allows to select new nodes based on an estimated cost and visit low-cost nodes first. In

the case of Wordle, the MIG heuristics are not applicable in such a manner as this would increase computing costs exponentially. The issue with the MIG heuristics is that they do not compute an expected score for a guess word but a value of average entropy reduction. Therefore it is not sufficient to evaluate the heuristic only on the current guess word but it is necessary to simulate a whole game using the heuristic resulting in significantly more heuristic evaluations. Instead, we make use of the fact that we only consider the n most promising successors. To improve the end game strategies of the UCT algorithm we add the possible solution words to these successors when at most 10 are left. This ensures that the algorithm will consider these words as guesses in early rollouts.

4.3.1 Implementation

To build the Monte Carlo Seach Tree we use a recursive implementation. The tree is made up of nodes that each contain the guess that leads to this node (another way of storing the state as in σ), the average cost this node is associated with so far, a visit counter, and the visited successors. We omit the details of how an unvisited successor to a node is found. Each time, the UCT algorithm is called, it will perform N rollouts as specified by a parameter to the algorithm. The first step is to create the root node for the MCTS. Only in this node, we use the optimization where we only consider the n most promising successor guesses according to the heuristic evaluation. This allows for comparability to the standard rollout approach. To incorporate the cases where $R^k(\rho_i) = 0$, we differentiate between unvisited and visited successors. When the algorithm is called it performs the specified amount of rollouts. For each rollout, a random solution is selected among all possible solutions. A rollout is visualized in Algorithm 4 where we start by visiting the root node.

Algorithm 4 Visit a node of the Monte Carlo search tree

```
1: Procedure visitNode(node, solution, SolutionWords, bias)
2: if length of solutionWords = 1 then
3:     return 1.0
4: end if
5: successor ← node.getUnvisitedSuccessor()
6: if successor = null then
7:     next ← applyTreePolicy(node, bias)
8:     utility ← visitTreeNode(next, solution, solutionWords, bias)
9: else
10:    utility ← simulateGame(successor.guess, solution)
11:    successor.avgCost ← utility
12:    node.visitedSuccessors.add(successor)
13: end if
14: node.visitCounter++
15: node.avgCost ← node.avgCost + ((utility - node.avgCost)/node.visitCounter)
16: return utility
17: End Procedure
```

This algorithm recursively visits nodes (that represent a state after a guess) until the solution to this rollout is found. Each visited node will update its average score according to the new rollout. In line 2 we check if we have already found the solution and return the cost of

1 for guessing this solution. If we have not found the solution yet, we check for unvisited successors of the current node. This allows for the integration of never-tried successors as specified by the UCT formula being ∞ in the case where $R^k(\rho_i) = 0$. Lines 10 to 12 visualize the simulation of a game using the heuristic for nodes never visited before. The simulation is skipped at this point as it is just guessing the word chosen by the heuristic and adapting the possible solution space accordingly until the solution to the rollout is found. If there is no unvisited successor left to the current node, we will apply the UCT formula (4.4) in line 7. The applyTreePolicy function selects the next successor based on the UCT formula. The algorithm then visits this successor to estimate the costs of choosing it as the next guess. Lines 14 and 15 refresh the variables of the current node by incrementing the visit counter and adapting the average Score according to the new rollout.
After the specified amount of rollouts is computed, the UCT algorithm selects the next guess by taking the successor of the root node with the lowest average score. As the comparison of the successor nodes and the tie breaking logic are trivial, they are skipped in this section.

5 Variations of Wordle

In this section, we explore several notable Wordle variations and discuss how the algorithms introduced earlier could be adapted to handle the change of rules incurred by each variant. By considering the hypothetical behavior of these algorithms under varying conditions, we aim to explore their flexibility and outline their theoretical limitations.

5.1 Hard Mode

In hard mode, the rules about allowed guess words change. It is no longer allowed to use each guess word at any stage of the game. After feedback on a guess is received, all available hints have to be used in each subsequent guess. If a letter is marked as green, it needs to be used in all guess words at the correct position. Likewise, a yellow marked letter needs to always be included from the moment on.

To apply these rules to our algorithms we need to add a functionality to filter the guess word list according to the the received feedback. As our implementation computes the feedbacks once upon initialization and later solely relies on integer values to identify words, we need to reintroduce the String representation of the words to be able to consider the feedback for individual letters. The overhead of this will be marginal as the list of words is reasonably small and the filtering has to be performed at most five times per game. The expensive part of computing the entropy reduction values can still be done using the integer representation and will be less expensive as there are fewer guess words to consider.

Time constraints did not allow for implementing the guess word filtering meaning we cannot discuss results regarding the performance of our algorithms in hard mode.

5.2 Word length

Another presumably less intriguing variation of Wordle is the length of the words being used. This is easily achieved by swapping the word lists for lists with words of different lengths and ensuring the array where we store the occurrences of individual feedback for the entropy calculations is large enough. While the variation of the word lists is less interesting for a human player, it has a great impact on the efficiency of our algorithms. Using longer words

significantly increases the amount of possible feedback and, as a consequence, increases the computation costs for the maximum information gain heuristic. With 5-letter words, there are $3^5 = 243$ distinct feedbacks to consider. By adding just one more letter to the words there are already $3^6 = 729$ distinct feedbacks increasing the complexity of the entropy calculations. The runtimes discussed in Chapter 6.2.3 suggest that the algorithms relying solely on the heuristic evaluation can handle words a couple of letters longer. However, it would be too expensive for the other approaches.
A compelling fact about the word length is that the feedback becomes more informative as the word length increases. Intuitively, with longer words, there is simply more space to mark correct or incorrect letters allowing for more distinct feedback on letter positions.

5.3 Word amount

Adding more words to the lists of allowed guess or solution words results in more expensive computations as more words have to be considered. Increasing the vocabulary size linearly increases the computation time as each word is treated individually. Based on the runtimes discussed in Chapter 6.2.3 the heuristic algorithms can handle word lists significantly larger while being able to compute a policy in a reasonable amount of time. The rollout approaches should be able to handle a reasonable increase in vocabulary size. Implementation-wise we would not need to modify anything. As the word lists are kept in memory, too large word lists could cause problems.

6 Results

This section will discuss the experiments conducted using the different algorithmic approaches to assess their performance. The goal is to evaluate algorithm performance in solving Wordle puzzles based on specific metrics. The main focus will lie on the accuracy of the policies produced, we will also comment on efficiency and resource utilization. We use our findings to determine the best opening guess and, additionally, we will compare our results to existing approaches.

6.1 Experimental Setup

The game of Wordle is managed by the New York Times which is constantly updating the word lists used for the game over time. In our experiments, we use older word lists widely used among researchers. This allows accurate comparison to different approaches. The word lists used contain 12,972 5-letter words that are considered valid guesses. 2315 of these words can be the solution to a Worlde game.

The accuracy of an algorithm is measured by simulating all 2315 games and computing the average score. This average score can be compared to the score of the optimal policy found by Bertsimas and Paskov [1]. We determine the accuracy of each of our algorithms for different fixed opening guess words including words that performed well in the experiments of other researchers as well as the optimal opening words chosen by our algorithms. These words are 'SOARE' for the MIG heuristics, 'CRATE' for both the standard rollout approach and UCT with MIG_B as well as 'TRACE' for UCT with MIG_A.

All experiments are conducted on the same controlled hardware. Calculations were performed at sciCORE[5] scientific computing center at the University of Basel, specifically the infai_1 computing nodes.

[5] Available at: http://scicore.unibas.ch/

6.1.1 Maximum Information Gain Heuristic

The experiments on the MIG heuristic are carried out with the parallelized implementation. The resources for each run with a fixed opening guess word are 8 CPU cores and 3GB of memory. This is a lot more memory than needed but ensures there are no bottlenecks regarding swap. The amount of CPU cores was chosen experimentally to yield the best runtime, balancing speed gained through parallel execution and computation spent on managing it. The parallelization decreased the runtime by around three to four times depending on the hardware used and has no effect on the average score computed.

6.1.2 Rollout

The experiments on the rollout approach are carried out with the same resources as for the MIG heuristic. We use the parallelized implementation of the heuristic. Experiments were administered using both versions of the heuristic, that is the version with alphabetical tie breaking, denoted with the suffix A, and the more sophisticated tie breaking logic indicated with the suffix B. For the rollout, we conducted experiments with $n = 10$ and $n = 100$ best-performing guesses.

6.1.3 UCT

The UCT algorithm has a significantly higher runtime than both our other approaches. To save on resources we only experimented with 'SALET' as a fixed opening guess. We compare $N = 50{,}000$ and $N = 100{,}000$ rollouts. Again, the experiments were performed with the same resources and we distinguish between the two versions of the MIG heuristic as mentioned above. The optimized version that adds possible solutions to the root successors to help the endgame performance will be denoted with an O suffix.

6.2 Evaluation of the algorithms

To evaluate each algorithm's performance, we will discuss accuracy and runtime. To maintain readability, we split this section and only focus on one metric at a time. An algorithm's accuracy reflects how good the achieved score is, while the runtime expresses efficiency. Additionally, we will comment on which opening word allows to achieve the best overall score.

6.2.1 Accuracy

The most important metric we want to compare is the accuracy of the algorithms. Table 6.1 contains the average scores obtained by the two versions of the MIG heuristic as well as the results from the Rollout algorithm. The best score for each opening guess word is marked in bold characters while the best score for each algorithm is marked in italic characters. The column 'Optimal' refers to the score that an optimal policy would achieve. We source these values from the scores compiled in [2]. Figure 6.1 further visualizes the average scores achieved.

Table 6.1: Average Scores for the Maximum Information Gain Heuristic and Rollout algorithms as well as the score of an optimal policy

Opening Guess	MIG_A	MIG_B	Rollout_10_A	Rollout_10_B	Rollout_100_A	Optimal
salet	3.6112	*3.4333*	3.5641	3.5866	*3.5387*	*3.4212*
carle	3.6104	**3.4410**	3.5767	3.5952	3.5490	3.4285
train	3.6186	**3.4553**	3.5866	3.5970	3.5654	3.4436
raise	3.6389	**3.4713**	3.6043	3.6233	3.5793	3.4618
clout	3.6976	**3.5235**	3.6613	3.6773	3.6354	-
soare	3.6380	**3.4648**	3.6013	3.6242	3.5784	-
salut	3.6644	**3.4898**	3.6324	3.6475	3.5983	-
lants	3.6803	**3.5343**	3.6803	3.6976	3.6549	-
reast	*3.6009*	**3.4380**	*3.5620*	*3.5797*	3.5421	3.4225
crate	3.6177	**3.4328**	3.5758	3.5948	3.5456	3.4238
trace	3.6069	**3.4354**	3.5646	3.5819	3.5374	3.4238
slate	3.6147	**3.4393**	3.5650	3.5840	3.5378	3.4246
trape	3.6320	**3.4600**	3.5918	3.6099	3.5624	3.4454
slane	3.6255	**3.4397**	3.5857	3.6052	3.5564	3.4311
prate	3.6333	**3.4536**	3.5905	3.6121	3.5603	3.4376
crane	3.6073	**3.4393**	3.5702	3.5844	3.5456	3.4255

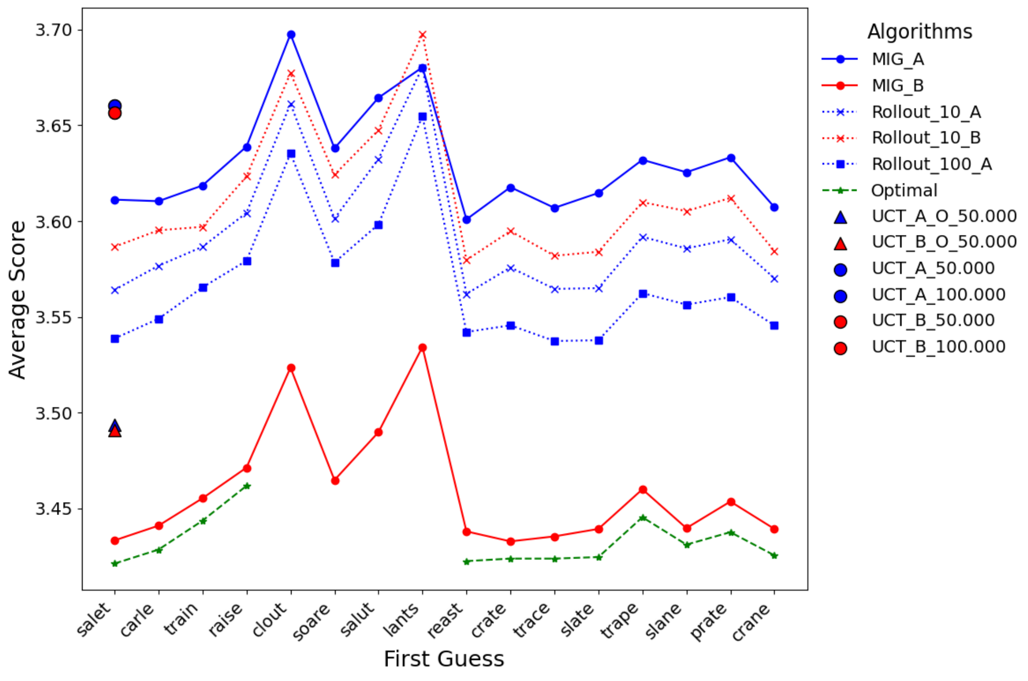


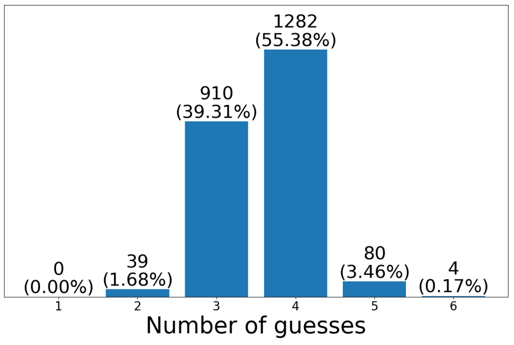
Figure 6.1: Comparison of the average scores achieved by MIG, rollout and UCT algorithms

The *MIG_A* heuristic performs well achieving scores within around 5% of the optimal score. The performance of the heuristic can be significantly improved by introducing better tie breaking in the case of equal average entropy reductions for multiple guess words. This is evident as *MIG_B* scores within 0.35% of the optimal policy. The rollout approach that aims to improve upon the performance of a heuristic marginally improves the score of *MIG_A*. Nonetheless, the improvement is not nearly as visible as found by Bhambri et al. [2] who were able to improve the score to be within 0.4% of optimality. Furthermore, the rollout approach even worsens the score in the case of the *MIG_B* heuristic. The scores of the rollout approach using both versions of the heuristic are similar but the rollout with the better heuristic consistently yields slightly worse results. The similarity of the results indicates that the rollout will score roughly the same regardless of the scores the heuristic would achieve on its own. We assume the reason for this is that the rollout approach does not need a well-scoring heuristic, but rather one that preserves the value ordering of the optimal policy as good as possible. Other findings seem to confirm this hypothesis. Bhambri et al. [2] tested a rollout algorithm with a couple of different heuristics, and their rollout approach returns similar results regardless of the scores of the heuristic used. It is important to note that all of the used heuristics score well and will therefore have similar properties regarding the value ordering, providing the rollout with a guideline. Our results further show that using $n = 100$ instead of $n = 10$ of the most promising next guesses yields slightly better results, as expected.
Time and resource constraints only allowed us to test the accuracy of the UCT algorithm for the fixed opening guess 'SALET'. The results are found in table 6.2. A similar pattern

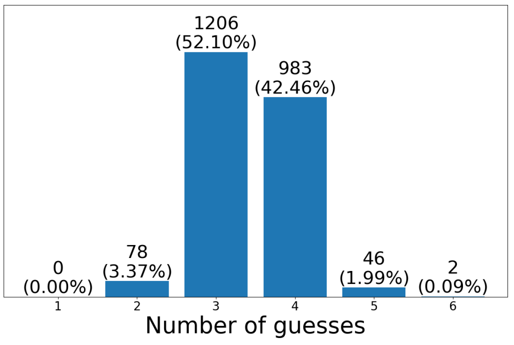
Table 6.2: Average scores for the UCT algorithm

Number of rollouts	UCT_A	UCT_B	UCT_A_O	UCT_B_O
50,000	3.6605	3.6566	3.4937	3.4907
100,000	3.6605	3.6566	-	-

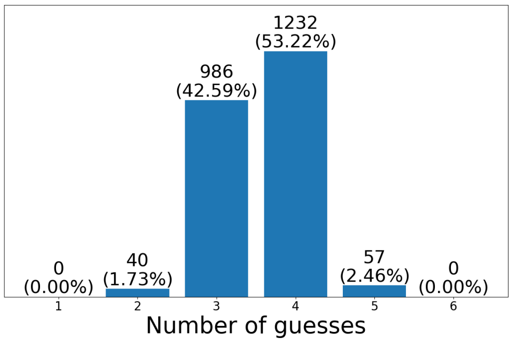
as for the standard rollout approach can be observed. UCT also yields slightly worse results when presented with a higher-quality heuristic. This leads to the same conclusion, that the accuracy does not strictly depend on the heuristic used and the heuristics rather serve as a guideline for the rollout algorithms. With the selected amount of rollouts, UCT performs slightly worse than any other rollout approach tested. This is likely due to the fact that, in order to converge to a policy of good quality, the UCT algorithm needs a number of rollouts that is orders of magnitudes higher than what we used. An interesting observation lies in the fact that UCT yields the same score regardless of using $N = 50{,}000$ or $N = 100{,}000$ rollouts. We explain this with the theory that the amount of rollouts is not nearly enough to test each possible successor multiple times across different possible solutions. Early rollouts are highly influenced by the heuristic and the random selection of solutions for each rollout. We also tested $N = 5000$ rollouts. This resulted in a slightly better score than with more rollouts. In this case, it is impossible to try each policy even once and therefore the algorithm score aligns more with the heuristic baseline. The slightly optimized version *UCT_O* where we add the possible solutions to the root successors in the end game significantly improves the



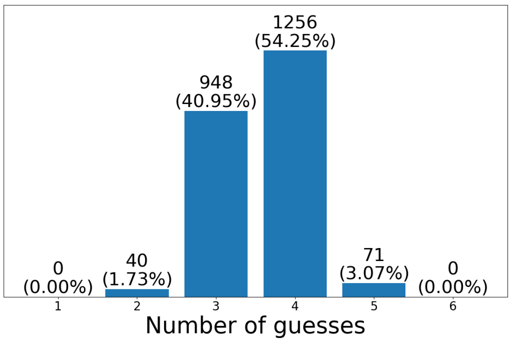
(a) MIG_A



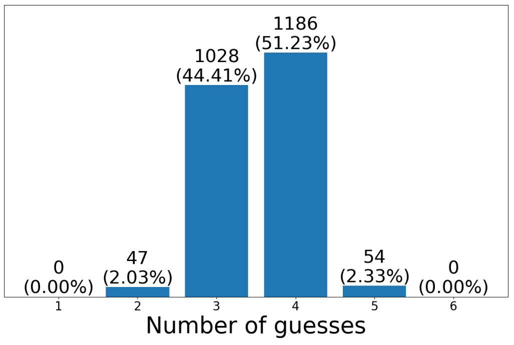
(b) MIG_B



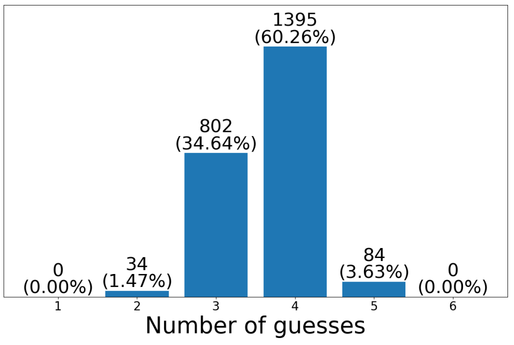
(c) Rollout_10_A



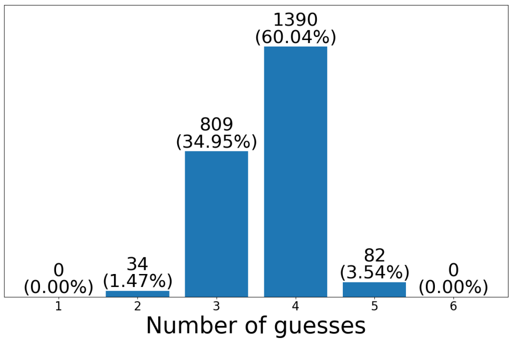
(d) Rollout_10_B



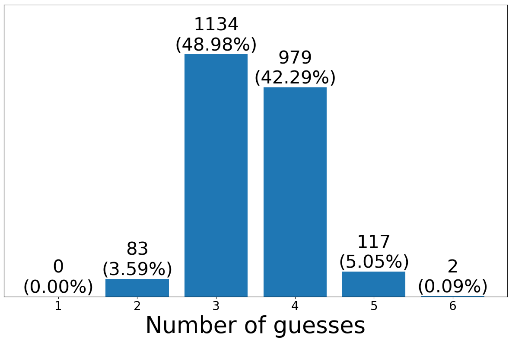
(e) Rollout_100_A



(f) UCT_A



(g) UCT_B



(h) UCT_B_O

Figure 6.2: Wordle games solved in at most n guesses for different algorithms. Subfigures (a)–(h) correspond to MIG_A, MIG_B, Rollout_10_A, Rollout_10_B, Rollout_100_A, UCT_A, UCT_B, and UCT_B_O, respectively.

score of the UCT algorithm by allowing earlier rollouts to consider possible solution words. Despite only affecting the endgame, we were able to significantly reduce the amounts of average required guesses from 3.6605 to 3.4937 using *MIG_A* and 3.6566 to 3.4907 using *MIG_B*. This is better than all of our other rollout approaches but still comes short of *MIG_B*. Time constraints did not allow us to apply a similar optimization to the other rollout approach. We can only assume this would lead to a similar effect here.
All implemented algorithms can find the solution to each game in at most six guesses as specified by the rules of Wordle. With 'SALET' as the opening guess both versions of the MIG heuristic sometimes require six guesses for a few games while all other algorithms can solve each Wordle game in at most five guesses with the same opening word. Figure 6.2 illustrates the amount of Wordle games solved in n guesses highlighting most games are solved in three or four guesses. Our best performing algorithms *MIG_B* and *UCT_O* are most likely to finish a game in three guesses while the other algorithms tend to need four.

6.2.2 Best opening guess

The tested algorithms perform comparably well regarding the opening guess choice with small differences in the ranking. The optimal policy as found by Bertsimas and Paskov [1] gets the best score with the opening guess of 'SALET'. Our two most accurate algorithms tested with multiple opening guess words *MIG_B* and *Rollout_100_A* also get the best score on this opening guess. All other approaches perform better with the opening guess 'REAST' which is the second best for the optimal policy. Thus, our results reflect the optimal policy quite well regarding the choice of the opening guess. We were able to confirm 'SALET' being the best option when desiring the best possible score while words like 'REAST', 'CRATE', 'CRANE', or 'TRACE' are further great options.

6.2.3 Runtime Requirements

Table 6.3 shows the time requirements for both heuristic versions as well as the rollout approach. The times denoted in the table are the average times to find the solution to a distinct Wordle game. All of the measured times are obtained by using the parallelized version of the heuristic. Figure 6.3 further visualizes the time differences by utilizing a logarithmic scale to the y-axis.
The plot in Figure 6.3 clearly shows that the rollout approach exponentially increases the runtime of the heuristics. It takes the parallelized implementation of the heuristic a couple of milliseconds to complete a distinct Wordle game. The rollout with $n = 10$ most promising next guesses takes a couple of seconds for each game. In practice, this runtime is still feasible. Increasing n linearly slows the runtime as expected. With double the amount of guesses to consider, double the amount of games has to be simulated. In practice using a high n value is not viable as it significantly increases the runtime while only marginally boosting the average score.
Across all algorithms, we can see a connection between the average score and the time taken. A worse score leads to a higher runtime. This is also as expected as a worse score correlates with worse guesses. A worse guess reduces the amount of possible solutions less than a good

Table 6.3: Average runtimes for the Maximum Information Gain Heuristic and Rollout algorithms to complete a distinct Wordle game measured in ms

Opening Guess	MIG_A (ms)	MIG_B (ms)	Rollout_10_A (ms)	Rollout_10_B (ms)	Rollout_100_A (ms)
salet	5.84	5.74	1507.53	1183.66	14486.84
carle	6.32	7.20	1244.55	1529.84	16290.10
train	6.61	6.27	1838.25	1485.88	15690.97
raise	5.85	6.73	990.71	1219.78	13128.42
clout	8.44	7.60	3763.18	2989.56	30749.71
soare	6.37	6.59	1223.43	1275.22	13870.86
salut	7.66	6.47	2676.67	2073.36	22281.57
lants	6.92	8.35	3118.62	3243.33	34920.17
reast	6.53	6.56	1468.83	1496.09	16096.90
crate	6.92	6.18	1548.09	1238.82	13077.92
trace	7.25	6.96	1575.95	1574.52	16676.68
slate	5.93	5.66	1461.38	1182.59	12624.15
trape	7.02	7.26	1408.99	1776.83	18915.19
slane	6.69	5.90	1533.87	1250.32	13196.11
prate	7.00	7.19	1385.20	1742.94	18664.90
crane	7.10	6.22	1659.33	1326.16	14139.39

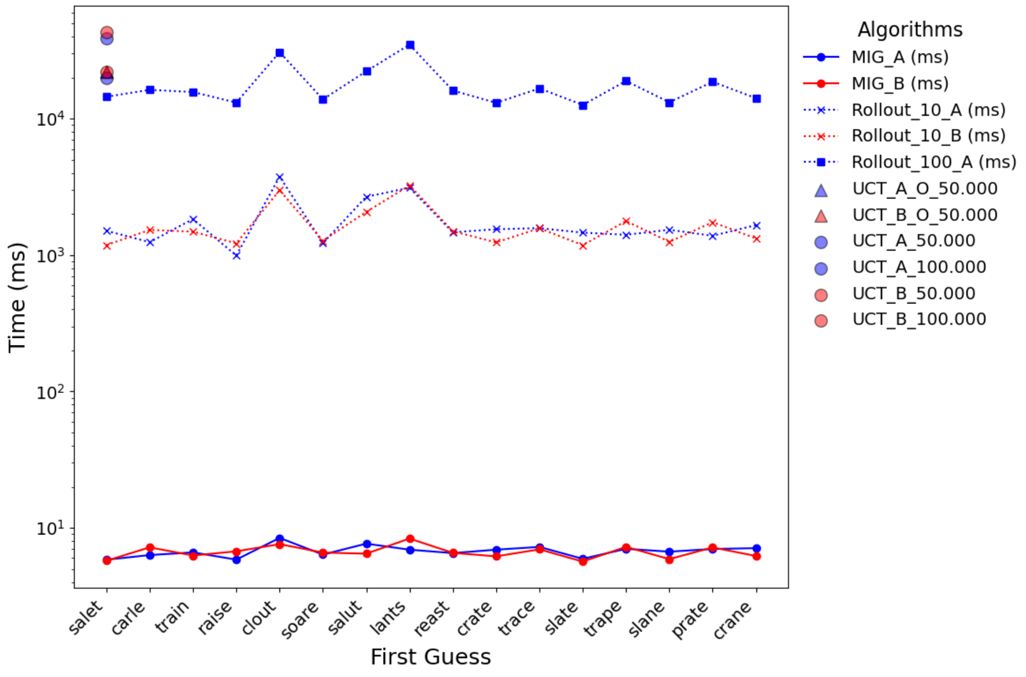


Figure 6.3: Comparison of the average runtimes for MIG, rollout and UCT algorithms to solve a distinct Worlde game using a log scale measured in ms

guess would, resulting in more cases that have to be computed by the heuristic.
The versions of the MIG heuristic share a comparable runtime with small differences in speed depending on the opening guess word. This can be explained by the tie breaking mechanism. While the more advanced tie breaking in MIG_B takes more time to compute, it reduces the amount of guesses taken and, with this, the runtime. Depending on how often the tie breaking has to be called MIG_A may be faster or slower than MIG_B.
In our experiments, the UCT approach has to perform a lot more simulations than the rollout algorithm. Table 6.4 denotes the average runtimes for the UCT algorithm. With $n = 10$ most promising successors the UCT algorithm is orders of magnitude slower than the standard rollout, even though we do not use an excessive amount of rollouts regarding UCT measures. UCT also incurs overhead for creating and managing the Monte Carlo Search Tree. As the accuracy of UCT with the tested amount of rollouts is significantly worse than the other approaches it is not feasible to use in practice.

Table 6.4: Average runtimes for the UCT algorithm with different amounts of rollouts to solve a distinct Worlde game with the opening guess 'SALET' measured in ms

Number of rollouts	UCT_A (ms)	UCT_B (ms)	UCT_A_O (ms)	UCT_B_O (ms)
50.000	19819.81	20545.45	22094.34	22055.80
100.000	38858.05	43131.73	-	-

6.3 Comparison to existing approaches

In this section, we will use results from the literature to assess the performance of our approaches. This allows us to classify the value of the developed algorithms. We compare the scores and efficiency from our heuristic evaluations to similar entropy-based attempts. Our rollout approach is compared to another rollout version aswell as to a rollout implementation for the Canadian Traveller's problem. We cannot compare our UCT implementation to another UCT-based Wordle solver as we do not know about similar implemented strategies. To achieve some form of assessment, we compare the UCT implementation with a UCT based solver for the Canadian Traveller's problem.

6.3.1 Maximum information gain heuristic

We compare our implementations of MIG_A and MIG_B with the adoptions of Sanderson [8] and Bhambri et al. [2]. Sanderson does not take advantage of the predefined list of possible solution words but uses relative word frequencies of the English language to evaluate the chance of a guess word being a potential solution. This improves his average score from 4.124 without considering word frequencies to 3.601 with 'CRANE' as the opening guess. However, these improvements lead the algorithm to fail six games previously completed, meaning it cannot find a solution in at most six tries. Still, this version can achieve a slightly better average score than the 3.607 MIG_A obtained for the same opening word. This reflects a great accomplishment by Sanderson as his methods of incorporating relative word frequencies yield great policies without knowing the true possible solution words. A

big disadvantage of his algorithm is some form of inconsistency as it is not able to solve all games. In terms of efficiency Sanderson's YouTube video hints at a computation speed of around 80 iterations per second. We do not know the hardware on which his experiments were performed so this serves only as a rough estimate. Our single-threaded versions of the MIG heuristic run at around 60 iterations per second while the parallelized versions run at around 180 iterations per second. Without further knowledge of hardware, we conclude that the efficiency is at a comparable level.
Sanderson has implemented another version of the maximum information gain heuristic where the set of possible solution words is used. Additionally, this approach computes the average entropy reduction two steps in advance instead of just one. This allows it to get an average score of 3.433 with 'SALET' as the opening guess. Overlooking some rounding inaccuracies, this is the same score that our MIG_B receives. We do not know the tie breaking logic applied in Sanderson's implementations. Still, our MIG_B heuristic achieves scores only obtained by a more complex two-step-look ahead approach. We also do not know about the efficiency of this version and can only assume it to be lower as a two-step computation is significantly more expensive.
The maximum information gain heuristic implemented by Bhambri et al. is a simple one-step-look-ahead version that makes use of the set of possible solution words. The score with the opening guess 'SALET' of 3.6108 slightly differs from MIG_A's 3.6112. Again, we do not know about the tie breaking logic of their approach. The similarity of the scores suggests a tie breaking according to the word order in the lists. We have no knowledge of the time efficiency of their approach.

6.3.2 Rollout

We compare our rollout implementation with the one found by Bhambri et al. [2]. To the best of our understanding, both approaches follow the same logic. In their experiments, they also use a list of $n = 10$ most promising guesses based on the heuristic evaluation. Still, they were able to achieve a significantly better improvement by applying the rollout methodology to the maximum information gain heuristic. They were able to improve the score with multiple opening guesses to be within 0.4% of the optimal score. To the best of our knowledge, we cannot find a reason for these inconsistencies as both approaches rely on the same methods for evaluating potential guess words. Our implementation was only able to slightly improve upon the performance of the worse-performing MIG_A heuristic while decreasing the score of the MIG_B heuristic. Using $n = 100$ most promising guesses led to a slight improvement of the performance, aligning with the findings by Bhambri et al.. In terms of efficiency, they claim the runtime of a single Wordle game to be in the order of a few seconds, matching our implementation.
Eyerich et al. [5] use a similar rollout approach for the Canadian traveler's problem. As there are no predefined solutions to this problem as in Wordle, their methods do a specified amount of rollouts on random problem instances. This could slightly improve the performance of a simple algorithm comparable to our heuristics. These improvements are in the same order of magnitude as in our approach while the rollouts sometimes worsen the score.

6.3.3 UCT

To the best of our knowledge, there is no comparable Wordle-solving algorithm leveraging the UCT formula for Monte Carlo Tree search. To get some form of comparison we take a look at Eyerich et al.'s [5] implementation of UCT for the Canadian traveler's problem. They use a simple UCT version that only relies on the UCT formula and an optimized version that leverages heuristic evaluations to guide early rollouts to promising parts of the search tree. The former version is comparable to our UCT implementation as we also solely rely on the UCT formula to guide the search. The latter version can be compared to our optimized UCT version in some ways. Our optimizations only apply to the end game and only to the case where we limit ourselves to the n most promising root successors, while their optimizations apply to the whole problem. They found the simple UCT version to score consistently worse than other approaches as the number of rollouts computed is largely too small to locate promising parts. Our results reflect the same behavior of worsening the score when using not enough rollouts. Their optimized version can score better than other tested approaches indicating a strong benefit to leveraging a heuristic to guide UCT. Despite only limited optimizations, we were able to also improve significantly on the performance of UCT with a limited amount of rollouts.
In terms of time efficiency, their UCT implementation is slightly faster than their simpler rollout versions. However, these versions do the same amount of rollouts as their UCT implementation while our simpler rollouts do far less, meaning we cannot compare the runtimes in a meaningful manner.

7 Conclusion

In this thesis, we designed and implemented several related algorithms to produce policies for the game Wordle. We derived a theoretical model of the game that allows us to apply algorithms from literature. The basis for all our algorithms lies in information theory, especially the concept of entropy. The heuristic of maximum information gain provides an intuitive way to estimate the quality of a potential guess word by evaluating the possible size reduction of the search space. We use this heuristic as a basis for various rollout approaches aiming to enhance the quality of the found policy through repeated simulation of possible games. The first rollout approach assesses a guess word by simulating the heuristic for every solution word still possible at the current stage of the game and chooses the one that yields the lowest average score. The UCT approach simulates a specified amount of random playouts while remembering paths taken previously. This enables the exploitation of paths that are known to produce good results. The UCT formula ensures that worse paths are chosen increasingly rarely over time while still being visited infinitely often given an unlimited amount of rollouts. With this, in the limit, the UCT algorithm can guarantee the exploration of all possibilities and therefore an optimal policy.

Our experiments show that relying solely on the maximum information gain heuristic already results in policies of good quality with MIG_B scoring within 0.35% of optimality. The implementation of the first rollout approach however could not get the expected results delivering only marginal improvements upon the simpler heuristic MIG_A while worsening the scores of MIG_B. We assume the reason for this is that the rollout algorithm does not require a well scoring heuristic but rather one that preserves the value ordering of the optimal policy. The UCT algorithm also scored worse than the maximum information gain heuristics for the tested amount of rollouts. With some light modifications, we could exploit the knowledge about the possible solution words to improve the end game strategies of the algorithm in the case where we only consider n of the best guesses which allowed for significant improvements to the average score of the policy. These findings align with evaluations of UCT for the Canadian traveler's problem by Eyerich et al. [5]

Our findings suggest that a heuristic algorithm can perform extremely well in practice. While not guaranteeing any properties regarding the quality of the found solution, in the special case of Wordle, they are a powerful way of finding excellent policies.

8 Future Work

While this thesis provided valuable insights into the characteristics of Wordle solving algorithms, it also highlights several opportunities for future exploration. In this section, we will highlight suggestions for how to extend and improve the research.

In Chapter 5 we described potential variations of the game and how the presented algorithms can be adapted to these new environments. Certainly, the most interesting variation is the hard mode. Adding the functionality to filter the list of potential guess words would enable experimental evaluations on how the algorithms would perform under stricter rules regarding the guess word selection.

Implementing a heuristic relying on different reasoning would enable further investigation of the behavior of the first implemented rollout approach. Our experiments showed that improvements to the score of the maximum information gain heuristic do not correspond to a better score of the rollout algorithm. Using a completely different heuristic methodology could verify our hypothesis that the algorithm does not depend on the heuristic used but rather needs some form of guidance.

We were not able to implement the improvements for the UCT algorithm discussed by Eyerich et al. [5] as this requires an estimate of the score that a guess word would achieve at any stage of the game. Our heuristic implementation has to simulate the whole game and, with it, entropy calculations for a vast number of guess words to estimate the score of a single guess word, rendering its application too complex for score estimates. The main problem is that the heuristic does not evaluate guess words according to their expected score but by the average entropy reduction. As the calculation for the entropy reduction for a single guess word is fast, new research could find a way to convert the entropy reduction value of a guess word to an expected score while keeping the stage of the game in mind. At a later stage of the game, the average entropy reduction of a guess word will be lower than at an earlier stage, as there are fewer possible solution words. Using these score estimates would allow the application of significant improvements to the UCT algorithm by guiding early rollouts to promising parts of the search tree.

Bibliography

[1] Dimitris Bertsimas and Alex Paskov. An Exact Solution to Wordle. *Operations Research*, 0(0), 2024. doi: 10.1287/opre.2022.0434. URL https://doi.org/10.1287/opre.2022.0434.

[2] Siddhant Bhambri, Amrita Bhattacharjee, and Dimitri P. Bertsekas. Playing Wordle Using an Online Rollout Algorithm for Deterministic POMDPs. In *IEEE Conference on Games, CoG 2023, Boston, MA, USA, August 21-24, 2023*, pages 1–4. IEEE, 2023. doi: 10.1109/COG57401.2023.10333228. URL https://doi.org/10.1109/CoG57401.2023.10333228.

[3] Blai Bonet. Deterministic POMDPs revisited. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 59–66, Arlington, Virginia, USA, 2009. AUAI Press.

[4] Yen-Chi Chen, Hao-En Kuan, Yen-Shun Lu, Tzu-Chun Chen, and I-Chen Wu. Entropy-Based Two-Phase Optimization Algorithm for Solving Wordle-like Games. In *2022 International Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 24–29, 2022. doi: 10.1109/TAAI57707.2022.00014.

[5] Patrick Eyerich, Thomas Keller, and Malte Helmert. High-Quality Policies for the Canadian Traveler's Problem. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pages 51–58, 2010.

[6] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[7] Michael Lederman Littman. *Algorithms for sequential decision-making.* Brown University, 1996.

[8] Grant Sanderson. Solving Wordle using information theory, 2022. URL: https://www.youtube.com/watch?v=v68zYyaEmEA [Accessed: 24.11.2024].

[9] Claude E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27(3):379–423, July 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x. URL https://doi.org/10.1002/j.1538-7305.1948.tb01338.x.