Nerdle

Like Wordle, but for nerds

Mayank Jain (A59024762)
Jonathan Mulyawan Woenardi (A59023890)
Jash Gautam Makhija (A59025270)
Shashi Dhanasekar (A59024717)
Yudhir Gala (A59024802)

March 15, 2024

Contents

1 Introduction

Nerdle, accessible at nerdlegame.com, stands as a captivating daily math game designed to challenge players with a penchant for logical thinking and mental arithmetic. With a limit of 6 attempts, players embark on deducing an 8-digit mathematical equation, commencing with an initial guess about the equation of the day.

Nerdle draws inspiration from the renowned word game Wordle, transforming it into a numerical realm. Crafted by Richard Mann, a data scientist based in London, Nerdle offers a web-based numeric puzzle where the objective is to unravel an 8-digit equation within a set number of attempts. Feedback after each guess guides players, indicating correct positions, misplaced symbols, and absent elements. Our project centers on solving this intricate 8-digit Nerdle puzzle.

2 How Nerdle is Played

2.1 Game Objective

The primary aim is to find the solution using as few guesses as possible, guided by feedback hints after each attempt. Correct elements are highlighted in green, and the challenge is to turn all tiles green.

2.2 Game Layout

The layout comprises a game board for entering guesses and displaying feedback, along with a keyboard for input. Nerdle equations adhere to specific rules, ensuring valid characters, correct calculations, and consideration of order of operations.

2.3 Rules of Nerdle

- Eight characters compose the equation.
- Valid characters include: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '-', '∗', '/', '='.
- The character combination must form a mathematically correct equation with one "=".
- The right side of '=' must be a number, not a calculation.
- Order of operations: '∗', '/' before '+', '-'.
- The solution excludes leading or lone zeros.
- The solution lacks a negative start number or a negative number after the equal sign, though valid guesses may include these.
- The solution may contain the same character more than once.

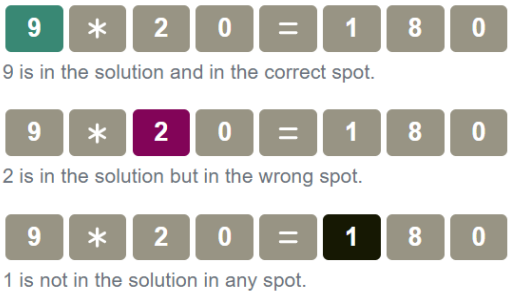

Figure 1: Tile color possibilities

2.4 Step-by-step Guide

- Initial Guess: Players begin the game by making an initial guess about the mathematical equation.
- Feedback Hints: After each guess, players receive feedback hints to guide them towards the correct solution. The hints are color-coded:
  - Green = correct character, in the correct position.
  - Purple = correct character, in the wrong position.
  - Black = incorrect character or more of the same character in the question than the answer (this is possible when there is duplicate characters in the guess or the solution).

  To be precise about handling the duplicate letters, we will provide an example:
  - Assume that there are 3 occurrences of "1" in the solution (e.g solution = "32-21=11")
  - There are 4 occurrences of "1" in the guess (e.g guess = "11+11=22")
  - The "1" which in the correct position must be colored Green.
  - Out of the other 3 occurences of "1" in the guess, only 2 must be colored Purple because there are in total only 3 ocurrences of "1" in the solution (2 if we don't count the one whose position matches).
  - Only the 2 left-most "1" will be colored Purple.
- Deductive Strategy: Players use deduction with each guess to improve their daily scores. They adjust their Nerdle strategy based on the feedback hints.

2.5 Game Walkthrough

1. In the initial attempt to solve the puzzle with the equation "48-36=12," as seen in Figure 2, feedback highlights that
   - "=" and "1" are correctly positioned (colored green),
   - "4," "-", and "6" are present but in the wrong positions (colored purple), and
   - "8," "2," and "3" are not part of the equation (colored black).
2. With this feedback, the equation is narrowed down to "_ _ _ _ _ = 1 _." Recognizing the

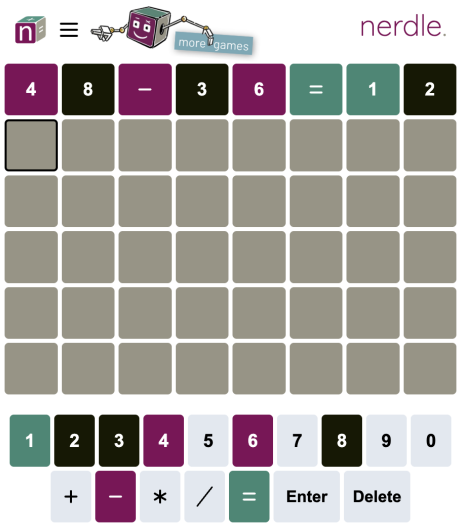

Figure 2: Initial Guess

challenge in predicting the equation solely from the initial feedback, a subsequent guess is made by excluding operators and numbers marked in black. For instance, testing the position of the operator "-", it leads to the guess "6∗4-9=15."

3. This refined equation becomes "6_ _ _ 9= 1_," and leveraging feedback from the previous step, the correct equation is deduced as "6-1+9=14." The player's strategic approach, demonstrated in the accompanying figure 3, results in successfully unraveling the puzzle in just 3 attempts.

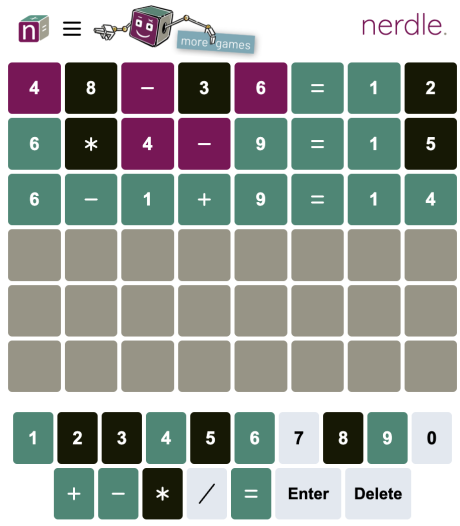

Figure 3: Solved Nerdle example

3 Terminology

In this project, our goal is to tackle various computational problems associated with the Nerdle game. These challenges will enable us to simulate the game and conduct a comprehensive analysis of the various strategies one can use to solve the game most effectively.

In this section, we establish the terminology crucial for tackling computational challenges in the Nerdle game.

1. Characters C are defined as the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, \times, \div, =\}$. These characters form the basic character set for mathematical equations.

2. An equation q is represented as a sequence $c_1, c_2, c_3, ...c_k$, where each c_i is an element from the character set C. The set Q consists of equations q.

3. The length of an equation q is denoted as k, representing the number of characters in the equation.

4. A hint set H is defined as { `EXACT_MATCH`, `WRONG_POSITION`, `WRONG_CHARACTER` }. Henceforth, we shall use `EM` for `EXACT_MATCH`, `WP` for `WRONG_POSITION`, and `WC` for `WRONG_CHARACTER` for brevity.

5. A guess g is represented as $[(c_1, h_1), (c_2, h_2), ...(c_k, h_k)]$, where each c_i is an element from C, and each h_i is a hint from H. The set G consists of guesses g. For any guess $g = [(c_1, h_1), (c_2, h_2), ...(c_k, h_k)]$, $[c_1, c_2, ..., c_k]$ is called the equation part, while $[h_1, h_2, ..., h_k]$ is called the feedback part.

4 Computing the list of valid mathematical equations for the solution

We want to generate a list of all possible mathematical equations. The solution will be selected from this list.

Input

1. Character set C

Output

1. List of equations Q_{all} where each equation q is $c_1, c_2, ...c_k$ where $c_i \in C$.

Constraints

1. The equations must adhere to the rules of a valid mathematical equation.
   (a) Each equation should have a single '='.
   (b) No leading zeroes.
   (c) No consecutive operators.
   (d) No operators at the start of the equation and in the RHS of the equation.
2. In addition to the equation following mathematical rules, we also impose constraints arising from the behaviour of the game.
   (a) The right-hand side (RHS) of the equation is limited to containing 1 to 3 digits. This restriction aims to control the complexity of the expressions.
   (b) Equations are constrained to exclude signed numerical values, ensuring that all numbers, including the starting one, are non-signed.

Objective

1. Exhaustive search across the entire equation space.

Algorithm

The algorithm uses complete search along with pruning and mathematical evaluation to generate the set of valid equations of length 8. We have broken down the algorithm into multiple functions for ease of understanding:

- `generate_valid_equations`: Generates all possible valid equations of length 8. Refer [1].
- `generate_expressions`: Generates all possible expressions of given length. Refer [2].
- `evaluate`: Parses arithmetic expressions and computes the result. Refer [3].
- `has_leading_zeroes`: Checks for leading zeroes in the expression. Refer [4].
- `operate`: Performs a mathematical operation between two operands and operator. Refer [4].

Proof of Correctness

To prove our algorithm is correct, we need to prove the list of equations it returns contains all the equations returned by a correct algorithm and does not contain any equations which are not returned by the correct algorithm.

We will first proof that `generate_expressions` returns all expressions using the characters. Using proof by induction,

Base Case: For length 1, the function returns a list of single-digit expressions, which are valid as they consist of characters from the character set and do not contain consecutive characters.

Inductive Step: Assuming the algorithm holds true for expressions of length t, we need to show that for $n = t + 1$, `generate_expressions(t+1)` generates all valid expressions of length $t + 1$. Consider an expression of length $(t+1)$ as two sub-expressions of lengths 1 and t. The expression of length 1 acts as the starting character. Since `generate_expressions(t)` returns all valid expressions of length t, by combining these with the starting character, we ensure all possible expressions of length $(t + 1)$ are generated. Combining two expressions, each representing sequences of characters, results in another sequence of characters. This satisfies the constraint of syntactic validity, as each character in the expression remains separated by operators, ensuring no consecutive operators are present.

Thus, the `generate_expressions`'s correctness is maintained as it effectively explores all valid combinations of expressions while ensuring syntactic correctness at each step.

The outer loop of the algorithm iterates over expressions generated by `generate_expressions` of length 4 to 6. For each generated expression, it checks several conditions:

- It ensures that the expression doesn't start with an operator.
- It checks for leading zeroes and skips such expressions.
- It evaluates the expression and verifies that the result is non-negative.
- It formats the expression as an equation and adds it to the list of valid equations only if its length is 8 characters.

Total length of equation must be 8. Since RHS can be 1-3 digits long, and knowing the equal symbol takes up 1 character, it leaves the LHS of the equation to be 4-6 characters long. By the inductive hypothesis, we know that `generate_expressions` correctly generates all valid expressions of length up to t. Therefore, by iterating over expressions of length 4-6, and applying the aforementioned conditions, the algorithm ensures that only valid equations of length 8 are included in the list of valid equations.

Thus, we have shown that the `generate_valid_equations` algorithm correctly generates valid arithmetic equations of length 8.

Time Complexity

A naive implementation would take $O(15^8)$ time complexity, as each position can have 15 characters. However, in our optimized algorithm, we utilize the constraints to prune the search space early on. The '=' character can either be at the 5^{th}, 6^{th} position or the 7^{th} position. Also, we only generate the LHS of the equation and evaluate the right-hand side; therefore, the outer loop ranges from 4 to 6 since the RHS can be either 1 to 3 digits long.

Time complexity for each of the functions is explained briefly below:

- `generate_valid_equations`: $O(3 \cdot |generate\_expressions(k)|) \cdot O(k)$ (nested loops, followed

Algorithm 1 Generate Valid Arithmetic Equations

```
1: procedure GENERATE_VALID_EQUATIONS
2:     valid_equations ← []
3:     for length in [4, 5, 6] do
4:         for expression in generate_expressions(length) do
5:             if not expression[0] isdigit() then
6:                 continue                    ▷ Skip expressions starting with an operator
7:             end if
8:
9:             if has_leading_zeroes(expression) then
10:                continue                    ▷ Skip expressions having leading zeroes
11:            end if
12:
13:            result ← evaluate(expression)
14:            if result < 0 then
15:                continue                    ▷ Skip expressions with negative RHS
16:            end if
17:
18:            formatted_expression ← expression + " = " + str(result)
19:            if len(formatted_expression) = 8 then        ▷ Length of the string must be 8
20:                valid_equations.append(formatted_expression)
21:            end if
22:        end for
23:    end for
24:    return valid_equations
25: end procedure
```

Algorithm 2 Generate Expressions of Fixed Length

```
cache ← {}
digits ← "0123456789"
operators ← " − +/ ∗ "
starting_characters ← digits + operators

procedure GENERATE_EXPRESSIONS(length)
    if length = 1 then
        return [digit for digit in digits]
    end if
    if length in cache then
        return cache[length]
    end if
    expressions ← []
    sub_expressions ← generate_expressions(length − 1)
    for start_char in starting_characters do
        for sub_expr in sub_expressions do
            if start_char in operators and sub_expr[0] in operators then
                continue                                  ▷ Rule out consecutive operators
            else
                expr ← start_char + sub_expr    ▷ Combine starting character and sub-expression
            end if
            expressions.append(expr)
        end for
    end for
    cache[length] ← expressions                          ▷ Cache the generated expression
    return cache[length]
end procedure
```

Algorithm 3 Evaluate an expression

```
1: procedure EVALUATE(expression)        ▷ Parse arithmetic expressions and compute the result
2:     numbers ← []
3:     operators ← []
4:     current_number ← 0                                  ▷ Variable to store current number
5:     for char in expression do
6:         if char.isdigit() then
7:             current_number ← current_number × 10 + int(char)          ▷ Construct number
8:         else
9:             if current_number then
10:                numbers.append(current_number)
11:                current_number ← 0
12:            end if
13:            operators.append(char)
14:        end if
15:    end for
16:
17:    if current_number then
18:        numbers.append(current_number)                ▷ Add last constructed number
19:    end if
20:
21:    if len(operators) = 0 then
22:        return numbers[0]                          ▷ No operators, return the only number
23:    end if
24:
25:    if len(operators) = 1 then
26:        return operate(numbers[0], numbers[1], operators[0])          ▷ Only one operator
27:    end if
28:
29:    if operators[1] in "*/"" and operators[0] in "+-" then
30:        return operate(numbers[0], operate(numbers[1], numbers[2], operators[1]), operators[0])
31:    else
32:        return operate(operate(numbers[0], numbers[1], operators[0]), numbers[2], operators[1])
33:    end if
34: end procedure
```

Algorithm 4 Utilities

```
1: procedure HAS_LEADING_ZEROES(expression)                ▷ Check for leading zeroes
2:     for idx, char in enumerate(expression) do
3:         if idx = 0 and char = "0" then
4:             return True
5:         else if char = "0" and expression[idx − 1] in ["+", "−", "/", "∗", "="] then
6:             return True
7:         end if
8:     end for
9: end procedure
10:
11: procedure OPERATE(a, b, operator)                      ▷ Perform Mathematical Operation
12:     if operator = "∗" then
13:         return a ∗ b
14:     else if operator = "+" then
15:         return a + b
16:     else if operator = "−" then
17:         return a − b
18:     else if operator = "/" then
19:         return a/b
20:     end if
21: end procedure
```

by operations for each iteration in nested loop) which simplifies to $O(3 \cdot 10 \cdot 14^6 \cdot 6)$ for $max(k) = 6$. The actual search space is much smaller since we do not allow consecutive operators and no leading zeroes, but we leave the exact calculation for later owing to the additional complexity.

- `generate_expressions`: $O(10 \cdot |C|^{\text{length}-1})$. Recurrence relation is given by $T(\text{length}) = T(\text{length}-1)+O(10 \cdot |C|^{\text{length}-1})$. After unfolding all terms, the constant term is the dominant term and hence, also the final time complexity.
- `has_leading_zeroes`: $O(k)$ where k is the length of the expression.
- `evaluate`: $O(k)$ where k is the length of the expression.
- `operate`: $O(1)$

Thus, the final time complexity of generating all valid possible equations is of the order of $O(10 \cdot 14^k \cdot k)$ where k is the length of LHS. A tighter bound can be established using permutations and combinations to understand the total number of possible expressions.

5 Evaluating the guess equation

In this problem, our focus is on evaluating the equation guessed by the player and return the hints corresponding to each character of the guessed equation by comparing it with the correct equation.

Input

1. The current guessed equation q and let $q = c_1, c_2, ..c_k$ denotes the characters of the guessed equation q where $c_i \in C$.
2. The answer equation.

Output

1. Updated Guess $g = [(c_1, h_1), (c_2, h_2), ...(c_k, h_k)]$ where $c_i \in C$, $h_i \in H$.

Constraints

1. Length of answer equation and length of guessed equation q must be equal to k.
2. For guess $g = [(c_1, h_1), (c_2, h_2), ...(c_k, h_k)]$, k is equal to the length of q.
3. $\forall (c_i, h_i) \in g$, the i^{th} character of $q = c_i$.
4. $\forall (c_i, h_i) \in g$, $h_i \in H$.

Objective

To find the hints corresponding to each character of the guessed equation by comparing it with the correct equation based on the rules explained in Section 2.4.

Algorithm

The algorithm evaluates a guessed equation for the game Nerdle based on a given answer. It returns an updated guess with hints (`EM`, `WP`, or `WC`) for each character in the guess.

The algorithm initializes an empty list upd_guess to store the updated guess. The algorithm counts the occurrences of each character in the answer and stores it in the dictionary ans_count. This dictionary will be useful when the algorithm is evaluating the guessed equation for determining the case of either `WP` or `WC` for a character in the guessed equation.

Then, the algorithm iterates through each character in the guess equation. If the character in the guess matches the corresponding character in the answer, it reduces the count of that character in ans_count. This indicates that the character has been correctly guessed.

Then for the evaluation of the guess, the algorithm iterates through each character in the guess equation again.For each character, if the character matches the corresponding character in the answer, it appends a tuple (character, `EM`) to upd_guess, indicating an exact match. If the character does not match the corresponding character in the answer and its count in ans_count is zero, it means that this character cannot be part of the answer. So, it appends a tuple (char, `WC`) to upd_guess, indicating a wrong character. If the character does not match the corresponding character in the answer and its count in ans_count is not zero, it means that this character appears in the answer but in the wrong position. So, it appends a tuple (char, `WP`) to upd_guess, indicating a wrong position. It also reduces the count of that character in ans_count. Finally, it returns the upd_guess, which contains the updated guess with hints for each character.

Algorithm 5 Nerdle Evaluation Algorithm

```
1: function NERDLE_EVAL(ans, guess_eq)
2:     upd_guess ← empty list
3:     ans_count ← {}
4:     for c in ans do
5:         ans_count[c] ← ans_count[c] + 1      ▷ Count occurrences of each character in the answer
6:     end for
7:     for i in range(len(guess_eq)) do
8:         if guess_eq[i] == ans[i] then
9:             ans_count[char] ← ans_count[char] − 1          ▷ Update count for exact matches
10:        end if
11:    end for
12:    for i in range(len(guess_eq)) do
13:        if guess_eq[i] == ans[i] then
14:            upd_guess.append((char, EM))                              ▷ Exact Match
15:        else if ans_count[char] == 0 then
16:            upd_guess.append((char, WC))                              ▷ Wrong Character
17:        else
18:            upd_guess.append((char, WP))                              ▷ Wrong Position
19:            ans_count[char] ← ans_count[char] − 1          ▷ Update count for wrong positions
20:        end if
21:    end for
22:    return upd_guess
23: end function
```

Correctness

For the evaluation of the guess, as the algorithm iterates through each character in the guess, it correctly handles the case of `EM` for every exact match with the corresponding character in the answer. According to the rules of the game Nerdle discussed before, if a particular character in the guess exists in the answer, then the number of occurrences of that character with the hint as either `EM` or `WP` should not exceed the number of occurrences of that character in the answer. The algorithm initially decrements the count of every character in the answer where there is an exact match with the guess before the assignment of hints to respective characters. By doing so, the algorithm takes into account the number of occurrences of that character with the hint as `EM`, ensuring the correct handling of the count of every character in the answer. Since the algorithm is iterating from leftmost character to rightmost character of the guessed equation, if the number of occurrences of a particular character in the guessed equation exceeds the number of occurrences

of that character in the answer, the algorithm ensures that starting from the leftmost position of that character in the guessed equation wherever there is not an exact match, it correctly handles the case of `WP` before the case of `WC` as discussed in Section 2.4. In the case of `WP`, for a particular character in the guess, the algorithm decrements the number of occurrences of that character in the answer, thereby ensuring that the number of occurrences of that character with the hint as either `EM` or `WP` does not exceed the number of occurrences of that character in the answer. Therefore, the algorithm correctly handles the cases of `WP` and `WC`.

Time Complexity

The time complexity of the provided nerdle evaluation algorithm can be analysed stepwise-

- Converting the answer string to a list and creating a dictionary to store the count of each character in the answer takes $O(k)$ time, where k is the length of the equation.
- In the first loop, it iterates through each character in the guess and checks if it matches the corresponding character in the answer. This takes $O(k)$ time.
- The second loop also iterates through each character in the guess and performs constant-time operations such as comparisons and dictionary lookups. This loop also takes $O(k)$ time.
- Constructing and returning the upd_guess list takes $O(k)$ time since it involves iterating through the equation that was guessed.

Overall, the time complexity of the algorithm is $O(k)$, where k is the length of the equation and is constant. As we consider $k = 8$ to be a fixed constant in this project, the time complexity of this computation problem can be considered as $O(1)$.

6 Computing a set of valid mathematical equations given a guess

As players, our task is to provide a mathematical equation as a guess. In this problem, our focus is on generating a list of mathematical equations that remain possible as the solution, given all previous constraints. We may also consider a "harder" version of the game, where the next guess must incorporate all existing clues that the player has already acquired.

Input

1. List of current valid equations Q such that $Q \subseteq Q_{all}$.
2. Latest guess $g = [(c_1, h_1), (c_2, h_2), ...(c_k, h_k)]$ where $c_i \in C$, $h_i \in H$.

Output

1. Updated list of equations Q'.

Constraints

For each equation in Q', there are several constraints between the latest guess and the equation based on the rules of the game.

There are two constraints related to the position of a character in the output equation, depending on whether a hint is `EM` or not:

- A character with a hint of `EM` implies that there should be the same character in the exact same position in the equation.
- A character with a hint not equal to `EM` implies that the same character should not be in the exact same position in the equation.

There are two additional constraints that bound the number of occurrences of a particular character in the output equation based on the existence of a `WC` hint:

- For a particular character c, if there is no hint of `WC`, then the number of occurrences of c in the equation must be more than or equal to those in the guess.
- For a particular character c, if there is a hint of `WC`, then the number of occurrences of c in

the equation must be equal to those with a hint not equal to `WC` in the guess.

Putting it mathematically,

1. $Q' \subseteq Q_{all}$
2. $\forall (c_i, h_i) \in g$ if $h_i = \texttt{EM}$ then $\forall q \in Q'$, the i^{th} character of $q = c_i$.
3. $\forall (c_i, h_i) \in g$ if $h_i \neq \texttt{EM}$ then $\forall q \in Q'$, the i^{th} character of $q \neq c_i$.
4. $\forall c \in C$ if $\nexists (c_i, h_i) \in g, c_i = c, h_i = \texttt{WC}$ then $\forall q \in Q'$, $|\{(c_i, h_i) \in g, c_i = c\}| \leq$ number of characters $c \in q$.
5. $\forall c \in C$ if $\exists (c_i, h_i) \in g, c_i = c, h_i = \texttt{WC}$ then $\forall q \in Q'$, $|\{(c_i, h_i) \in g, c_i = c, h_i \in \{\texttt{EM, WP}\}\}| =$ Number of characters $c \in q$.

Objective

1. Find all equations from $Q \subseteq Q_{all}$ satisfying the constraints.

Algorithm

This algorithm is designed to update a list of equations based on the latest guess. The goal is to remove equations from the current valid list of equations that do not satisfy constraints with respect to the latest guess. An empty list 'to-remove' is initialized. This list will store equations that need to be removed. The algorithm loops through each equation in the list of current valid equations Q. For each equation, initially the algorithm loops through each character in the equation and checks two constraints are being followed against the hint h_i and character c_i from the latest guess for that index position i .If h_i is not `EM` and c_i matches the character in the equation, or if h_i is `EM` but c_i does not match the character in the equation, the equation is added to 'to-remove', and the loop breaks. This step ensures that equations are removed if they contain characters that don't match with the latest guess, considering the `EM` condition.

The algorithm, for each character c in the set of characters C , checks two additional constraints that bound the number of occurrences of c in the latest guess based on the existence of a WC hint. If there are no hints as `WC` for the character c in the latest guess g, and the count of c in the equation

is less than the count of c in the latest guess g, the equation is added to 'to-remove', and the loop breaks. If there exists a hint `WC` for the character c in the latest guess g, the algorithm calculates the count of characters with hints as `EM` or `WP` associated with c in g. If the count of c in the equation does not match this count, the equation is added to 'to-remove', and the loop breaks.

After identifying equations to be removed, the algorithm updates Q by taking the set difference between Q and 'to-remove'. The algorithm returns the updated set Q', which contains equations that satisfy the conditions with respect to the latest guess g.

Algorithm 6 Computing Valid Equations

Input: Q (List of current valid equations), g the latest guess
Output: Updated Q'
Initialize an empty list *to_remove*
for each equation in Q **do**
    for each (i, char) in enumerate(equation) **do**
        $(c_i, h_i) \leftarrow g[i]$
        if $h_i \neq$ `EM` and $c_i =$ char **then**
            Append equation to *to_remove*
            break
        else if $h_i =$ `EM` and $c_i \neq$ char **then**
            Append equation to *to_remove*
            break
        end if
    end for
    for each c in C **do**
        if all$((c_i, h_i) \neq (c,$ `WC`$)$ for (c_i, h_i) in $g)$ **then**
            if sum(1 for char in equation if char $== c) < sum(1$ for(c_i, h_i) in g if $c_i = c)$ **then**
                Append equation to *to_remove*
                break
            end if
        else if any$((c_i == c$ and $h_i ==$ `WC`$)$ for c_i, h_i in $g)$ **then**
            $wc\_count \leftarrow sum(1$ for (c_i, h_i) in g if $c_i = c$ and $h_i \in \{$`EM`, `WP`$\})$
            if sum(1 for char in equation if char $== c) \neq wc\_count$ **then**
                Append equation to *to_remove*
                break
            end if
        end if
    end for
end for
$Q' \leftarrow$ set.difference(Q, to_remove)
return Q'

Correctness

This algorithm is intended to update a set of equations, Q, after a new guess is made by a player. The objective of the algorithm is to remove any equation from Q that cannot be used as the target equation because of the hints from an evaluation result. To prove the correctness of the algorithm, we need to show that:

1. The algorithm correctly handles each of the three possible hints (`EM`, `WP`, and `WC`) and removes all equations from Q that are inconsistent with the evaluation result.
2. $Q' \subseteq Q_{all}$

To prove the first point, we can analyze the hint cases:

1. For the `EM` case, we can observe that if an equation does not contain the given character in the specified position, it cannot be the target equation. Therefore, removing such equations from Q is correct and does not remove any valid equations.
2. Based on the presence of a `WC` hint, there are two additional constraints regarding the occurrence of a specific character in the output equation:
    - If there is no `WC` hint for a particular character, then the number of times the character appears in the equation must be greater than or equal to the number of times it appears in the guess, or else the equation cannot be the target equation.
    - If there is a `WC` hint for a particular character, then the number of times the character appears in the equation must be equal to the number of times it appears with a hint other than `WC` in the guess, or else the equation cannot be the target equation.
3. Therefore, removing such equations from Q is correct and does not remove any valid equations.

Finally since Q was already a subset of Q_{all} and we are only removing equations from it, hence the updated set Q' is still a subset of Q_{all}. In conclusion, the algorithm above is correct and removes all equations from Q that are inconsistent with the hints provided by the evaluation result, while leaving only the equations that are consistent with the clues and form the possible valid set of

equations for the game.

Time Complexity

The time complexity depends on the equation length and the size of the set of current valid equations Q. The equation length is always constant and is equal to k. Suppose the size of Q is m. To analyze the time complexity of the provided algorithm, let's break it down:

1. Initialization: Initializing an empty list to_remove takes constant time, $O(1)$.
2. Checking Equations:
    a) For each equation in Q, the algorithm iterates through each character, requiring $O(k)$ time, where k is the length of the equation. Inside this loop:
        - Retrieving the corresponding character-hint pair from the latest guess g takes constant time, $O(1)$.
        - The condition checks and operations inside the loop have constant time complexity, $O(1)$.
    b) For each equation in Q, the algorithm iterates over each character in the set of characters C, requiring $O(|C|)$ iterations. For each character c, the algorithm checks conditions based on the latest guess g. For each condition check, it requires iterating over elements in g, which has a size of $O(k)$. The condition checks and operations inside the loop have constant time complexity, $O(1)$.
3. Constructing the set to_remove involves iterating over the equations in Q and performing checks for each equation. This operation has a time complexity of $O(m \times (|C| \times k + k))$. Since k and $|C|$ are constants, we get $O(m)$.
4. Taking the set difference between Q and to_remove to get the updated set Q' has a time complexity of $O(|Q|) = O(m)$.

Hence, the overall time complexity of the algorithm is $O(m) + O(m)$, i.e., $O(m)$.

7 Data Structure to maintain the game state

In the previous computational problems, we have presented several algorithms necessary for simulating the Nerdle game. In this computational problem, our aim is to create a single data structure called $GameStateManager$ which maintains every important states of the game as it proceeds. The data structure should call the previous computational problems as subroutines as they implement the logic of the game.

There are two benefits of introducing this data structure:

- It can be used by other computational problems. For example, in the next computational problem, we are going to search for the best equation that the player can make using a variant of minimax algorithm. The implementation can utilize the data structure for optimized implementation and clear abstraction.
- It can be used for experimentation and game simulation. We will use this data structure when we are collecting data.

7.1 Design

We model the Nerdle game as a two-player game, where `User` plays agains `Computer`. The `User` makes a "move" by guessing an *Equation* and the computer makes a "move" by providing the *Feedback* for the last *Equation* that the `User` provided. The `User` and `Computer` must makes a move alternately.

The data structure exposes certain methods to keep track of all the the *Equations* and *Feedbacks* that the `User` and `Computer` has provided. It also provided certain methods to get further information about the game (e.g. the list of equations that are possible as the solution). A complete list of all the methods can be find in Table 1.

We can also consider a state machine with two states which represents whether it is the `User`'s or `Computer`'s turn to move. Some of the methods will modify the state while the other methods are only applicable for a specific state. For more explanation, see the Figure 4 that represents the state machine diagram.

Method	**Input**	**Output**	**Description**
get_possible_equations	None	List[Equation]	Return all equations that can still possibly be the hidden equation given all guesses that have been made.
submit_equation	Equation	None	User makes a move by submitting an equation as a guess. It is now the computer's turn to make a move by submitting a feedback.
revert_equation	None	None	Revert previous user's move.
get_possible_feedbacks	None	List[Feedback]	When it is the computer's turn to move, return a list of all possible feedback that the computer can give.
get_reduce_equations_size	Feedback	int	Given that the computer makes a move by submitting a particular feedback, return the size of the reduced list of possible equations.
submit_feedback	Feedback	None	Computer makes a move by submitting a feedback to the last guess's equation. It is now the user's turn to make a move by submitting the next equation.
revert_feedback	None	None	Revert the previous feedback given by the computer

Table 1: Methods supported by GameStateManager

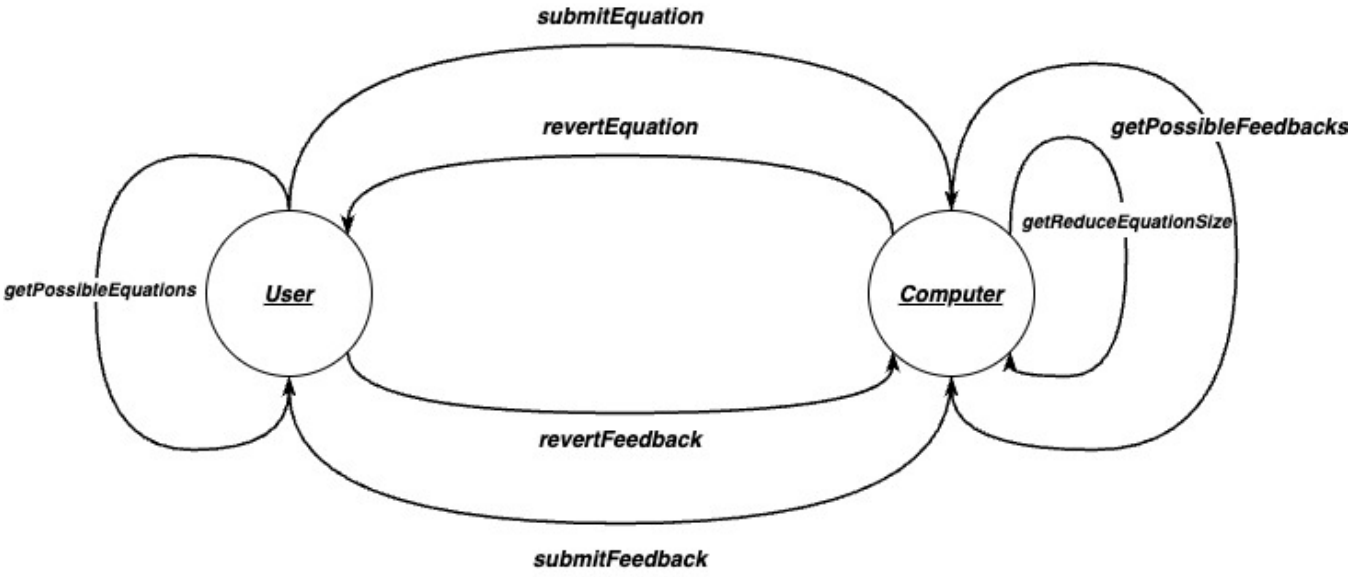

Figure 4: State transition for each methods in GameStateManager

In this report, we will assume that methods will be called accordingly based on whether it is the `User` or the `Computer` turn to move. For example, consecutive alternate calls of *submitEquation* and *submitFeedback* is a valid sequence of method calls.

7.2 Naive Implementation

The naive implementation is a very intuitive implementation that manages all the game states and repeatedly calls the `EvaluateGuess` and `FilterEquationsByGuesses` to implement the required logic.

Implementation

In the naive implementation, we will keep several lists as our game states:

- *guess_equations* is the list of equations that the `User` submits as the game progresses. The last element of the list should be the last guess the `User` made.
- $feedbacks$ is the list of feedbacks that the `Computer` submits. The feedback on index i should correspond to the guess' equation on index i.
- *possible_equations_versions* keep tracks of possible equations that can still be possible given the existing guesses. The first element is exactly the original list of all equations in the game.

As the `User` makes more guesses, this list of equations get smaller and *possible_equations_versions* keep track of all versions of this list.

The implementation utilizes previous computational problems as subroutines:

- `EvaluateGuess` (Section 5): given a guess and equation, return the appropriate feedback.
- `FilterEquationsByGuesses` (Section 6): given a list of possible equations and guesses, filter the list for equations that satisfy the constraints based on the gueses.

The implementation of each method is described in the Algorithm 7.

Proof of Correctness

To prove the correctness, we will establish certain invariants that the algorithm should hold.

When it is the `User`'s turn to move:

- The *guess_equations* and $feedbacks$ have the same length and holds the previous guesses.
- The *possible_equations_versions* has one more element than *guess_equations*.
- The last element of *possible_equations_versions* is the filtered list of possible equations that satisfy all existing guesses made before.

When it is the $Computer's$ turn too move:

- The *guess_equations* has one more element than $feedbacks$. The last element of *guess_equations* is the last equation that the `User` has submitted while the other holds the previous guesses.
- The *possible_equations_versions* is unchanged since before the `User` submit the last equation.

The methods below change the game turn from `User` to `Computer` or otherwise. They are also modifying the internal lists of $GameStateManager$. We will show that they preserve the invariants as defined below.

- `SUBMIT_EQUATION` changes the turn from `User` to `Computer`. It append a new element to the *guess_equations*, which is the latest equation that the `User` has submitted. Based on the previous invariant on the `User`'s turn, the *guess_equations* and $feedbacks$ have the same length. Thus, now on the `Computer`'s turn, *guess_equations* has one more element than the

Algorithm 7 Naive Game State Manager

```
1: Class NaiveGameStateManager
2: Implements: GameStateManager
3: procedure INIT(initial_equations)
4:     possible_equations_versions ← [initial_equations]
5:     guess_equations ← []
6:     feedbacks ← []
7: end procedure
8: procedure GET_POSSIBLE_EQUATIONS
9:     return possible_equations_versions[−1]
10: end procedure
11: procedure SUBMIT_EQUATION(equation)
12:     guess_equations.append(equation)
13: end procedure
14: procedure REVERT_EQUATION
15:     guess_equations ← guess_equations[: −1]
16: end procedure
17: procedure GET_POSSIBLE_FEEDBACKS
18:     feedback_set ← empty set
19:     for each hidden_equation in possible_equations_versions[−1] do
20:         guess ← EvaluateGuess(guess_equations[−1], hidden_equation)
21:         feedback_set.add(tuple(get_feedback_from_guess(guess)))
22:     end for
23:     return [list(feedback) for feedback in feedback_set]
24: end procedure
25: procedure GET_REDUCE_EQUATIONS_SIZE(feedback)
26:     guess ← zip(guess_equations[−1], feedback)
27:     new_possible_equations ← FilterEquationsByGuesses(possible_equations_versions[−1], guess)
28:     return len(new_possible_equations)
29: end procedure
30: procedure SUBMIT_FEEDBACK(feedback)
31:     feedbacks.append(feedback)
32:     guess ← zip(guess_equations[−1], feedback)
33:     new_possible_equations ← FilterEquationsByGuesses(possible_equations_versions[−1], guess)
34:     possible_equations_versions.append(new_possible_equations)
35: end procedure
36: procedure REVERT_FEEDBACK
37:     feedbacks ← feedbacks[: −1]
38:     possible_equations_versions ← possible_equations_versions[: −1]
39: end procedure
40: procedure ZIP(equation, feedback)
41:     return [(e_char, f_char) for e_char, f_char in zip(equation, feedback)]
42: end procedure
43: procedure GET_FEEDBACK_FROM_GUESS(guess)
44:     return [g_char.hint for g_char in guess]
45: end procedure
```

feedbacks. The *possible_equations_versions* list is unchanged. Therefore, the `Computer`'s turn invariant holds in this case.

- `REVERT_EQUATION` reverts the actions made by the latest `SUBMIT_EQUATIONS` by removing the last element that has just been inserted into *guess_equations*. As `SUBMIT_EQUATIONS` preservers the invariant, the revert also preserves the invariant.
- `SUBMIT_FEEDBACK` changes the turn from `Computer` to `User`. It append a new element to the *feedbacks*, which is the feedback associated with the latest equation in *guess_equations*. Based on the previous invariant, there were previously one more element in *guess_equations*. Thus, with this new feedback inserted, there is now exactly the same number of elements in *guess_equations* and in *feedbacks*. The method also computed the newest reduced list of possible equations by using the current list of possible equations and insert a singleton list consisting of only the last guess. By the invariant, the current list already taken into account all previous guess until before the last guess. By induction, the new list taken into account all guesses that has been made so far. This new list will be appended into *possible_equations_versions*, keeping the invariant of having one more element in this list than *guess_equations*.
- `REVERT_FEEDBACK` reverts the actions made by the latest `SUBMIT_FEEDBACK` by removing the last element that has just been inserted into *feedbacks* and *possible_equations_version*. Therefore, the `User`'s turn invariant holds in this case.

The methods below does not change the game turn and does not modify any internal lists. Thus, we have shown that the invariant always holds as this data structure being used. We will now prove that these methods implement the required functionalities correctly.

- `GET_POSSIBLE_EQUATIONS` returns the last element of the *possible_equations_version*, which is the latest version of the list of possible equations by the invariant.
- `GET_POSSIBLE_FEEDBACKS` iterates overall possible equations and uses the `EvaluateGuess` subroutine to compute the feedback return by the game based on the rule of the game. It added them into a set for deduplication and return the elements in a list. This is consistent with the

requirement.

- `GET_REDUCE_EQUATIONS_SIZE` essentially simulates the logic of `SUBMIT_FEEDBACK` without actually inserting the feedback and result to the internal list. Thus, the methods correctly returns the number of possible equations in the new reduced list if a specific feedback is materialized.

Time Complexity Analysis

Let n be the number of possible equations in the *initial_equations* and let m be the number of possible equations in the latest version of list of possible equations (i.e. $possible\_equations\_versions[-1]$). We know that for any i, $possible\_equations\_versions[i+1]$ is a subset of $possible\_equations\_version[i]$. Therefore, we know that $m = O(n)$.

Most of the time complexity of each methods will be either $O(1)$ *or* $O(m)$.

- `GET_POSSIBLE_EQUATIONS` returns the last element of the *possible_equations_versions*. Thus, the time complexity is $O(1)$.
- `SUBMIT_EQUATION` and `REVERT_EQUATION` are $O(1)$ operations.
- `GET_POSSIBLE_FEEDBACKS` iterates overall possible equations and uses the `EvaluateGuess` subroutine to compute the feedback return by the game based on the rule of the game. The complexity of `EvaluateGuess` is $O(1)$. Therefore, the overall time complexity is $O(m)$.
- `GET_REDUCE_EQUATIONS_SIZE` calls the `FilterEquationsByGuesses` with an input list of length m and a guess list of length 1. The time complexity is this subroutine call is $O(m)$. Other operations inside the methods are $O(1)$. Therefore, the overall time complexity is $O(m)$.
- `SUBMIT_FEEDBACK` is similar to `GET_REDUCE_EQUATIONS_SIZE`, where it calls `FilterEquationsByGuesses` once with an input list of length m and a guess list of length 1. The overall time complexity is $O(m)$.
- `REVERT_FEEDBACK` is an $O(1)$ operation.

7.3 Optimized Implementation

The previous naive implementation is sufficient to solve the next computational problem, which is to compute the next best equation that a user can make using a variant of minimax algorithm.

However, it won't be as optimized. Therefore, we provide an optimized implementation, which implements the exact some requirement, but with different time complexity trade-off.

The key idea is to do additional pre-computation when calling `SUBMIT_FEEDBACK`. This will increase the time complexity of `SUBMIT_FEEDBACK` to linear time but decrease the time complexity of `GET_POSSIBLE_FEEDBACKS` and `GET_REDUCE_EQUATIONS_SIZE`.

Implementation

In addition to the several lists that we keep track in the naive implementation, we are going to store one more list called *equation_clusters_versions*. Similar to *possible_equations_versions*, the word versions here means that it keep track of all versions as the game progress with every guesses made. Given a new equation that the `User` has submitted, we can iterate over all possible equations and call `EvaluateGuess` to get the feedback. Different equations can lead to the same feedback. We want to partition the list of possible equations by the feedback that it leads to. Therefore, the element of *equation_clusters_versions* is a dictionary that maps a feedback into a sub-list of the current list of possible equations which all leads to the same particular feedback.

Proof of Correctness

In addition to the invariants described in the previous $NaiveGameStateManager$, we will add one additional invariants.

When it is the `Computer`'s turn to move, the *equation_clusters_versions* have the same length as *guess_equations*, and the last element is a dictionary representing the partition of the current possible list of equations based on the feedback on the last guess.

All the previous invariants from *Naive Game State Manager* are unchanged as the previous 3 lists are not modified. The `SUBMIT_EQUATION` is the only method modifying *equation_clusters_versions* and it pre-computed the newest partition map by iterating over all possible candidate of hidden equations and evaluating the current guess against the hidden equation. Therefore, this preservers the invariant.

We will also show the correctness of the modified functions:

Algorithm 8 Optimized Game State Manager

```
Class OptimizedGameStateManager
Implements: GameStateManager
procedure INIT(initial_equations)
    possible_equations_versions ← [initial_equations]
    guess_equations ← []
    feedbacks ← []
    equation_clusters_versions ← []
end procedure
procedure GET_POSSIBLE_EQUATIONS
    return possible_equations_versions[−1]
end procedure
procedure SUBMIT_EQUATION(equation)
    guess_equations.append(equation)
    clusters ← {}
    for each hidden_equation in possible_equations_versions[−1] do
        guess ← EvaluateGuess(guess_equations[−1], hidden_equation)
        feedback ← get_feedback_from_guess(guess)
        clusters[tuple(feedback)].append(hidden_equation)
    end for
    equation_clusters_versions.append(clusters)
end procedure
procedure REVERT_EQUATION
    guess_equations ← guess_equations[: −1]
    equations_clusters_versions ← equations_clusters_versions[: −1]
end procedure
procedure GET_POSSIBLE_FEEDBACKS
    return [list(feedback) for feedback in equation_clusters_list[−1]]
end procedure
procedure GET_REDUCE_EQUATIONS_SIZE(feedback)
    return len(equation_clusters_versions[−1][tuple(feedback)]))
end procedure
procedure SUBMIT_FEEDBACK(feedback)
    feedbacks.append(feedback)
    guesses ← [zip(guess_equations[−1], feedback)]
    possible_equations_versions.append(equation_clusters_versions[−1][tuple(feedback)])
end procedure
procedure REVERT_FEEDBACK
    feedbacks ← feedbacks[: −1]
    possible_equations_versions ← possible_equations_versions[: −1]
end procedure
procedure GET_FEEDBACK_FROM_GUESS(guess)
    return [g_char.hint for g_char in guess]
end procedure
```

- `GET_POSSIBLE_FEEDBACKS`: the pre-computed mapping is computed by iterating over all possible candidate of hidden equations. Thus, the list of the keys of the mapping are the list of all possible feedback.
- `GET_REDUCE_EQUATIONS_SIZE`: let f be the particular feedback submitted by the `Computer`. Calling `FilterEquationsByGuesses` with the last guess will filter out the words that does not satisfy the constraint induced by the last guess. However, during the pre-computation, we have done exactly the same thing by calling `EvaluateGuess` and we exactly partitions the list of possible equations into clusters which will be the new list of possible equations if the feedback associated with them is chosen. Thus, we can just use the dictionary to return the size of the new reduced list of equations.
- `SUBMIT_FEEDBACK`: similar to the previous point, we can just use the value from the mapping and append it to the $possible\_equations\_versions$.

Time Complexity Analysis

Let n be the number of possible equations in the $initial\_equations$ and let m be the number of possible equations in the latest version of list of possible equations (i.e. $possible\_equations\_versions[-1]$). We know that for any i, $possible\_equations\_versions[i+1]$ is a subset of $possible\_equations\_version[i]$. Therefore, we know that m *is* $O(n)$.

Most of the time complexity of each methods will be either $O(1) or O(m)$.

- `GET_POSSIBLE_EQUATIONS` returns the last element of the $possible\_equations\_versions$. Thus, the time complexity is $O(1)$.
- `SUBMIT_EQUATION` iterates overall possible equations and uses the `EvaluateGuess` subroutine to compute the feedback return by the game based on the rule of the game. The complexity of `EvaluateGuess` is $O(1)$. Therefore, the overall time complexity is $O(m)$.
- `REVERT_EQUATION` is an $O(1)$ operations.
- `GET_POSSIBLE_FEEDBACKS` iterates over the partition mapping to get the keys which are the feedback list. The number of feedback is bounded by the number of possible equations. Therefore, the overall time complexity is $O(m)$.

Method	Naive Implementation	Optimized Implementation
get_possible_equations	O(1)	O(1)
submit_equation	O(1)	O(m)
revert_equation	O(1)	O(1)
get_possible_feedbacks	O(m)	O(m)
get_reduce_equations_size	O(m)	O(1)
submit_feedback	O(m)	O(1)
revert_feedback	O(1)	O(1)

Table 2: Time Complexity Comparison between Naive and Optimized Implementation

- `GET_REDUCE_EQUATIONS_SIZE` is an $O(1)$ operation because getting an element from dictionary, getting the last element from a list, and getting the length of a list are all constant time.
- `SUBMIT_FEEDBACK` and `REVERT_FEEDBACK` is an $O(1)$ operation.

7.4 Time Complexity Comparison

As shown in the Table 2, there is a difference in time complexity between the Naive Implementation and the Optimized Implementation. They key difference is that `GET_REDUCE_EQUATIONS_SIZE` is optimized from $O(m)$ into $O(1)$. The trade-off is that `SUBMIT_EQUATION` is now $O(m)$ as we are doing precomputation. However, this will lead to an improved runtime for the next computational problem from $O(n^3)$ to $O(n^2)$.

8 Figuring the best equation the player can use as the next guess

We aim to determine the best possible starting equation that would provide the most information for the player. This approach could be extended later to assist the user in deciding which word to guess throughout the game when there are more clues.

We want to assist the player in solving the game by acting as the player and submitting the most effective guess. The objective is to reduce the total number of guesses required to figure out the solution.

Input

1. List of valid equations (Q).

Output

1. An equation q.

Constraints

1. $q \in Q$.

Objective

1. Find the equation that reduces the search space maximally (results in minimum possible search space)

Min-Average

Minmax algorithm is a decision-making algorithm used in game theory, specifically for two-player zero-sum games. The two players are working towards opposite goals to make predictions about which future states will be reached as the game progresses, and then proceeds accordingly to optimize its chance of victory. It is based on the idea that the algorithm's opponent will be trying to minimize whatever value the algorithm is trying to maximize (hence, "minimax").

We use min-average algorithm, a version of minmax algorithm, to find the next best guess that a user can make. Unlike minmax, the computer's objective is not really to oppose the user but rather evaluate the user's guess against each possible win equation. Therefore, the computer computes a

weighted average of the children's heuristic values. On the other hand, the user's objective is to minimize the search space of possible equations. The user selects the move that corresponds to the lowest value among its children.

There are two major steps in the algorithm

1. Building a tree - The game tree represents all possible moves that both players can make from the current state of the game. It consists of alternate levels of *user* and *computer* nodes. For the user nodes, all possible equations are evaluated and for the computer nodes all possible feedbacks are evaluated. At the terminal nodes, the heuristics are computed to evaluate the desirability of that state.

2. Back-propagating the heuristic - In order to compute the best guess that a user can make, the heuristic values and corresponding moves are propagated back up the tree. If it's the user's turn, they select the move that corresponds to the minimum reduced size of equations among the possible options. If it is the computer's turn, they compute the weighted average of their children's values. The heuristic that we use is the size of the reduced set of possible equations after making a guess and getting a feedback.

Building a Tree

We present the procedure to build the tree in Algorithm 9. Each node in the tree has the following attributes

1. player_type - indicating whether it is a User node or a Computer Node

2. terminal value - stores the heuristic value (only for the terminal nodes)

3. child Node - List of tuples of the form (equation, Node) to keep track of the guess equation that leads to the child node. The user's children nodes take this form whereas the computer's children nodes take the form (None, Node) since their moves are based on feedback.

We build the tree in a depth-first manner using recursion. The *User* nodes are present at the root and in even heights of the tree. *Computer* nodes are present in odd heights.

In case of a *user* node, we do the following: submit an equation as a guess, create a child node of type *computer*, build subtree for the child node and revert the guess equation to try the next one.

In case of *computer* node, we perform the following: submit the feedback, compute the reduced size of resulting equation set, create a child node of type *user*, set the terminal value to heuristic (reduced set size), build subtree for the child node and revert the feedback to try the next one.

Algorithm 9 Build Tree

```
1: procedure BuildTree(self, node = None, depth = 0)
2:     if depth == 0 then
3:         return
4:     else if (depth%2) == 1 then
5:         current_possible_feedbacks ← self.gameStateManager.get_possible_feedbacks()
6:         for f in current_possible_feedbacks do
7:             terminal_value ← None
8:             if depth == 1 then
9:                 reduced_equation_size ← self.gameStateManager.get_reduce_equations_size(f)
10:                terminal_value ← reduced_equation_size
11:            end if
12:            childNode ← Node(terminal_value, USER)
13:            node.children.append((None, childNode))
14:            if depth ≠ 1 then
15:                self.gameStateManager.submit_feedback(f)
16:                BuildTree(self, childNode, depth − 1)
17:                self.gameStateManager.revert_feedback()
18:            end if
19:        end for
20:    else if (depth%2) == 0 then
21:        current_possible_equations ← self.gameStateManager.get_possible_equations()
22:        for eq in current_possible_equations do
23:            childNode ← Node(None, COMPUTER)
24:            node.children.append((eq, childNode))
25:            self.gameStateManager.submit_equation(eq)
26:            BuildTree(self, childNode, depth − 1)
27:            self.gameStateManager.revert_equation()
28:        end for
29:    end if
30: end procedure
```

Back-propagating the heuristic

The procedure to back-propagate the heuristic up the tree is outlined in Algorithm 10. We perfom

back-propagation recursively. In case of leaf node, the heuristic (terminal value) is returned. For the *User* node, the minimum value among the children and the corresponding guess equation is returned. For the *Computer* Node, the weighted average of the children values are returned. The weights are the heuristics in this case.

Algorithm 10 Expectimin

```
1: procedure EXPECTIMIN(self, node = None)
2:     if node.is_terminal() then
3:         return None, node.terminal_value
4:     else if node.player_type == USER then
5:         value ← float('inf')
6:         eq ← []
7:         for n in node.children do
8:             exp_value ← self.expectimin(n[1])
9:             if exp_value[1] < value then
10:                value ← exp_value[1]
11:                eq ← n[0]
12:            end if
13:        end for
14:        return eq, value
15:    else if node.player_type == COMPUTER then
16:        value ← 0
17:        num_children ← len(node.children)
18:        if num_children ≠ 0 then
19:            nr ← 0
20:            dr ← 0
21:            for n in node.children do
22:                exp_value[1] ← self.expectimin(n[1])
23:                nr ← nr + exp_value[1] × exp_value[1]
24:                dr ← dr + exp_value[1]
25:            end for
26:            value ← nr/dr
27:        end if
28:        return None, value
29:    end if
30:    return
31: end procedure
```

Correctness

We perform min-average at each level of the tree (consecutive *user* and *computer* depths) and back-propagate the result to the level above it. At the root node, we get the guess that maximally

reduces the search space.

We prove this using induction.

Base Case For $level = 1$ (that is $depth = 2$), we have one layer of user node and one layer of computer nodes. The computer nodes return the weighted average of the heuristic (reduced set size) and the user node chooses the guess corresponding to the minimum weighted average. So, the user makes a guess that leads to a maximum reduction in set of possible equations

Inductive Hypothesis We assume that the user makes the best guess for a tree of $level = k$ where $k > 1$

To prove We need to prove that the user makes the best guess for a tree of $level = k + 1$

Proof

We have to show that the problem for $level = k + 1$ is exactly the same as the sub-problem with $level = k$.

The algorithm consists of two phases:

1. Building the tree - the tree is constructed in a recursive fashion depth-first manner. Let R be the new root *user* node, C be the child *computer* nodes and U is the list of all the children of *computer* nodes. We call the build tree on R which creates all child nodes in C. We then call build tree for the nodes in C which creates the nodes in U. When we call build tree for all nodes in U, it is exactly the same as building a tree for $level = k$ which is optimal by the induction hypothesis.

2. Back-propagating the feedback - we are back-propagating the heuristic also in a recursive fashion. By inductive hypothesis, we have the best guess equations and the averaged heuristics at $level = k$, i.e, we have the best guess equations and the averaged heuristics for all nodes in U. For $level = k + 1$, we pick the equation that has the minimum averaged heuristic among all the best guesses of nodes in U. Hence, we can say that we are choosing the best move for $level = k + 1$.

Time Complexity

Total time complexity of the min-average algorithm is the sum of the following:

1. Time Complexity for Building the Tree - Let n be the number of possible equations. Let *level* represent a pair of consecutive *user* and *computer* layers, i.e, $level = depth/2$.

   First, we will find the number of nodes in the tree.

   The number of nodes at $depth = 0$ is 1. The number of nodes at $depth = 1$ is n where n is the number of possible equations. The number of nodes at $depth = 2$ (corresponding to the children of *Computer* nodes) is $n * m$ where m is the number of possible feedback. But the number of feedback is bounded by the number of possible equations. Hence, it is n^2. Though there might be a significant decrease in the size of possible equations in subsequent levels, it is still bounded by $O(n^2)$. Generalizing this, the number of nodes at $depth = d$ is $O(n^d)$. In terms of *level* (one layer of *user* nodes and one layer of *computer* nodes), we have $depth = 2 \cdot level$. Therefore, for $level = k$ the number of nodes is $O(n^{2 \cdot k})$.

   Now, we look at the time complexity for each node in one level.

   For each *user* node, we get the set of possible equations. The time complexity for this part is $O(1)$. For each guess of the *user* node, we have the following operations: submit an equation as a guess $O(n)$, revert the guess equation to try the next one $O(1)$. The time complexity for n such guesses is given by $O(n \cdot n) + O(n \cdot 1) = O(n^2)$. Therefore, time complexity for one *user* node is T(get possible equations) + T(operations for all the guesses) $= O(1) + O(n^2) = O(n^2)$

   For each *computer* node at $height \neq 1$ or $depth \neq 2 \cdot k$, we get the set of possible feedback. The time complexity for this part is $O(n)$. For each feedback of the *computer* node, we have the following operations: submit the feedback $O(1)$, revert the feedback to try the next one $O(1)$. The time complexity for n such feedback is given by $O(n * 1) + O(n * 1) = O(n)$ Therefore, time complexity for one *computer* node is T(get possible feedback) + T(operations for all the feedback) $= O(n) + O(n) = O(n)$

For each *computer* node at $height == 1$ $depth == 2 \cdot k - 1$, we get the set of possible feedback. The time complexity for this part is $O(n)$. For each feedback of the *computer* node, we have the following operations: compute the reduced size of resulting equation set $O(1)$. The time complexity for n such feedback is given by $O(n \cdot 1) = O(n)$ Therefore, time complexity for one *computer* node is T(get possible feedback) + T(operations for all the feedback) = $O(n) + O(n) = O(n)$.

We do not perform any operations for the leaf nodes, i.e, nodes at $depth = 2 \cdot k$. For each *computer* node, the time complexity for all operations is $O(n)$. For each *user* node, the time complexity for all operations is $O(n^2)$.

The total time complexity is $\sum_{i=0}^{k-1} n^{2 \cdot i} \cdot O(n^2) + \sum_{i=0}^{k-1} n^{2 \cdot i + 1} \cdot O(n) = O(n^{2 \cdot k})$

2. Time complexity for Back-propagating the heuristic - It is linear in terms of the number of nodes in the tree. Above, we proved that the number of nodes at the end of $level = k$ is $O(n^{2 \cdot k})$. Therefore, the time complexity for $level = k$ is $O(n^{2 \cdot k})$.

Total Time Complexity is given by

$$\begin{aligned} Time\ Complexity &= O(n^{2 \cdot k}) + O(n^{2 \cdot k}) \\ &= O(n^{2 \cdot k}) \end{aligned}$$

9 Experiments

We implemented our project and ran all our experiments in Python 3.10. The experiments were run on a Apple Macbook Pro (3.2 GHz Octa-Core Apple M1 Pro) with 16 GB RAM.

9.1 Time complexity for generating valid equations and expressions

It took us 13.50 seconds to generate all possible 17723 valid equations. This function utilizes `generate_expressions` internally and calls it thrice with parameters = 4, 5 and 6. We chose to profile the function `generate_expressions` since it is responsible for producing a core set of expressions upon which the logic for generating valid equations depends.

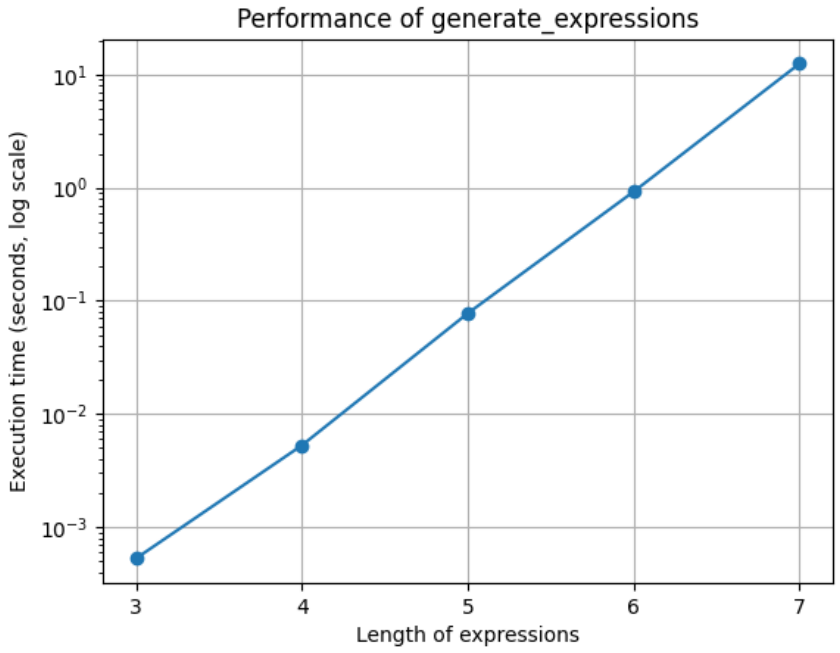

Figure 5: Execution Time (seconds, log scale) vs. Length of Expressions

The log plot is linear suggesting a exponential time complexity of the function. The gradient of the plot is roughly 2.570. $e^{2.570} \approx 13.06$. We expected the exponent to be 14 if we had done no pruning at all, and 10 if we only used digits. This sets the range for the observed exponent between 10 and 14. We can also deduce that since there are 14 choices for every character and we rule out consecutive operators, using probably we get 180 valid choices / 196 total choices = 0.918. $0.918 \times 14 = 12.85$ which is very close to what we observed.

9.2 Time complexity for figuring the best equation

We are now able to combine all components of our projects and evaluate the feasibility of using our project to effectively figure out the best equation for solving Nerdle.

The main part of the solver is running the Min-Average algorithm with a particular implementation of GameStateManager. Min-Average can be run using different search depth, but the default search depth is 1). There are also two different implementation of GameStateManager: naive and optimized.

Naive vs Optimized implementation

For our first experiment, we set the search depth = 1, and use both the naive and optimized GameStateManager. We tested by first sampling a subset of the original set of 17K equations. We use power of two for the size of the subset.

The result can be found in Figure 6

Evaluation:

We have previously shown that for search depth = 1, the runtime for figuring out best equation is:

- Using Naive GameStateManager: $O(n^3)$
- Using Optimized GameStateManger: $O(n^2)$.

The experiment result is shown in a log-log graph. The result is as expected because the runtime for the naive implementation has higher slope which implies higher exponent for the overall runtime compared to the optimized version.

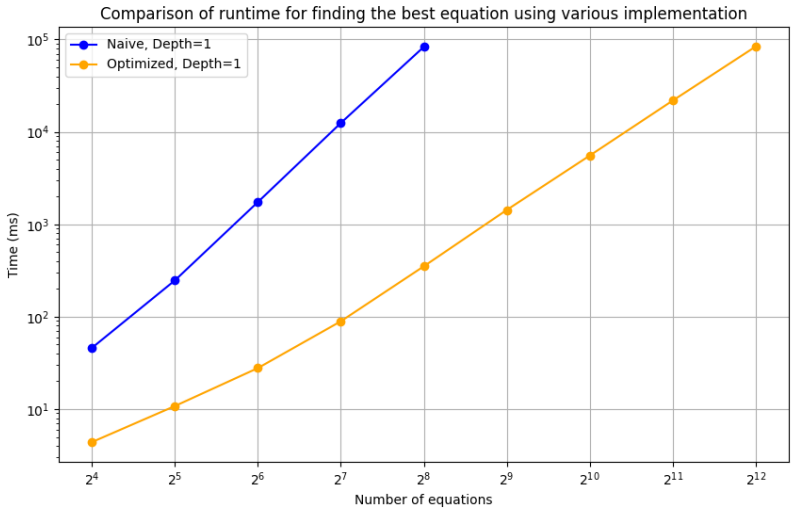

Figure 6: Execution Time for various implementation

Search depth = 1 vs 2

The next experiment that we conduct is to compared the runtime for running the Min-Average algorithm using the optimized GameStateManager using different search depth = 1.

The result can be found in Figure 7.

Evaluation:

We have previously shown that if we use the Optimized GameStateManager, the runtime for figuring out best equation is:

- Search depth = 1: $O(n^2)$
- Search depth = 2: $O(n^4)$

The experiment result is shown in a log-log graph. The result shown that with search depth = 2, the runtime is slower than search depth = 1. However, the runtime growth is not as slow as expected, which is $O(n^4)$. The reason is that in general, after the first guess, the possible equation set has reduced significantly as demonstrated in the next section. For example, with initial set of 1024 equations, after the first guess the possible equation set is reduced to less than 10 equations.

Therefore, the time complexity does not grow as fast as $O(n^4)$. Our experiment result is expected.

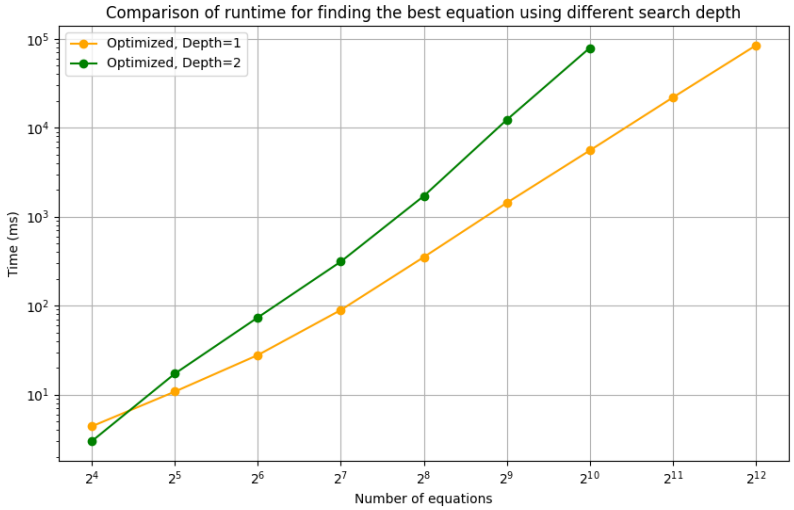

Figure 7: Execution Time for search depth

9.3 Best starting equation

We use our algorithm to find the best equation as the starting point. We use the Optimized implementation with search depth = 1 and we found that the best equation to start is

$$48 - 16 = 32$$

The overall time required to compute this best starting equation is 23 minutes. While this is very slow, this computation is only needed once as the best starting equation is always the same as there are no guesses yet to be made.

Figure 8 presents a histogram of reductions in the search space size using our best initial guess $(48 - 16 = 32)$ for all possible win equations. We can conclude that the search space has reduced to size less than 110 for all the win equations. Majority of win equations have their search space reduced to less than 50. We can see a decreasing trend in the number of win equations as the reduction values increases. Hence, we can conclude that our initial guess was indeed the best.

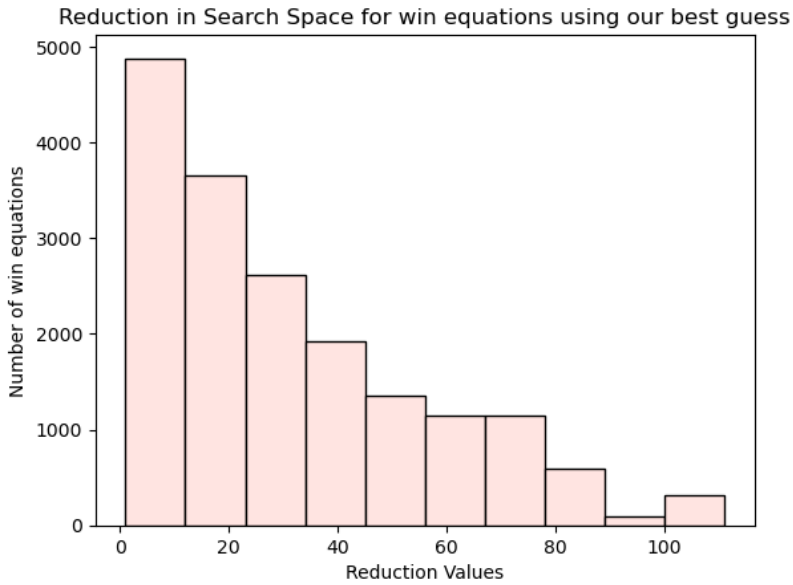

Figure 8: Histogram showing reductions in search space using our best initial guess

9.4 Conclusion

Our main conclusion is that our implementation is a feasible approach to solve Nerdle effectively and quickly. While figuring the best starting equation is slow, this is a one-time operation that need not be repeated every time. After the first guess, based on our statistical finding, the set of possible equations is reduced to less than 100 equations. At that number of equations, running our optimized implementation with search depth = 1 requires less than 0.1 second per guess, and with search depth = 2 requires less than 0.5 second per guess.