Advanced Data Structures

Word Games

Jannik Hösch

April 13, 2024

1 Introduction

This report examines the computational underpinnings of two popular word-based games: Wordle and WordChallenge. Both games, while different in their gameplay mechanics, challenge players to deduce or construct words from a given set of letters, underpinning their entertainment value with considerable algorithmic complexity. At the heart of these games is the efficient management of a dictionary, the generation of feedback based on player guesses, and the strategic selection of guesses by the computer. This analysis aims to explain the data structures and algorithms that facilitate these core functionalities, emphasizing their role in ensuring game performance and player engagement.

In addition, the report presents a performance analysis of the two games, focusing on metrics such as the average number of guesses required in Wordle and the scalability of WordChallenge with respect to word length. By performing benchmarks, I aim to quantify the efficiency of my implemented solution and provide insights into the computational requirements and optimization strategies that characterize these games.

2 Trie

The Trie, a fundamental data structure in computer science, is essential to a wide range of applications, from autocomplete systems to spell checkers and beyond. At its core, a trie is a tree-like structure that efficiently stores a dynamic set of strings by using the common prefixes of the strings. This section explains the characteristics of the Trie data structure, highlighted by a discussion of my implementation of its key properties: nodes, insertion and search.

2.1 TrieNode

The TrieNode class serves as the foundational element of the Trie. Each node contains:

children: A dictionary that maps characters to their subsequent TrieNodes

is_end_of_word: A boolean flag that marks the completion of a valid word

```
1 class TrieNode:
2     def __init__(self):
3         self.children = {}
4         self.is_end_of_word = False
```

This structure enables the Trie to represent each letter of a string with a node in the tree, where paths from the root to a marked end node spell out the strings stored in the Trie.

2.2 Insert Function

The insert method integrates a new word into the Trie with the following steps:

- **Initialization:** Begin at the root node.
- **Traversal and Expansion:** If no child node exists for the character, a new TrieNode is created.
- **Mark Completion:** Upon reaching the end of the word, set the is_end_of_word flag of the final node to True, signifying the presence of a complete word within the Trie.

```
    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True
```

By inserting each word from the dictionary file into the Trie, we can build a comprehensive Trie representing all possible words with very efficient access for the word games. Additionally, I create the dictionary *words_by_length*, that maps the length of words to lists containing all words of that specific length. This improves the selection process for game mechanisms that require a word of a certain length.

```
    def load_dictionary(self, file_path):
        words_by_length = {}
        with open(file_path, 'r') as file:
            for line in file:
                word = line.strip().split(":")[0]
                self.insert(word)
                words_by_length.setdefault(len(word), []).append(word)
        return self, words_by_length
```

2.3 Search Function

The search method locates a word in the Trie with the following steps:

- **Initialization:** Begin at the root node.
- **Traversal:** For each character in the target word, the search moves to the corresponding child node.
- **Check Existence:** If at any point the required child node does not exist, the word is not in the Trie, and the search returns False.
- **Word Completion:** Successfully reaching the end of the word with is_end_of_word flagged as True confirms the word's presence in the Trie.

```
    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word
```

3 WordChallenge

Word Challenge is a game that challenges players to generate all possible words from a given set of letters, while adhering to a dictionary of valid words.

Select word In interactive mode the player chooses the set of letters manually, while in automatic mode, the program selects a random word from the dictionary of length l. First, the program selects a random word from the dictionary of length l. I use the *words_by_length* dictionary created when the initial dictionary was loaded into the Trie to efficiently select a random word of length l. This eliminates traversing the Trie to find all words of a certain length. Before using the selected word as input for the solver, the letters are randomly shuffled.

```
        # Pick a random word
        secret = random.choice(word_lengths[length])

        # Rearrange letters
        letters = list(secret)
        random.shuffle(letters)
        shuffled = ''.join(letters)
```

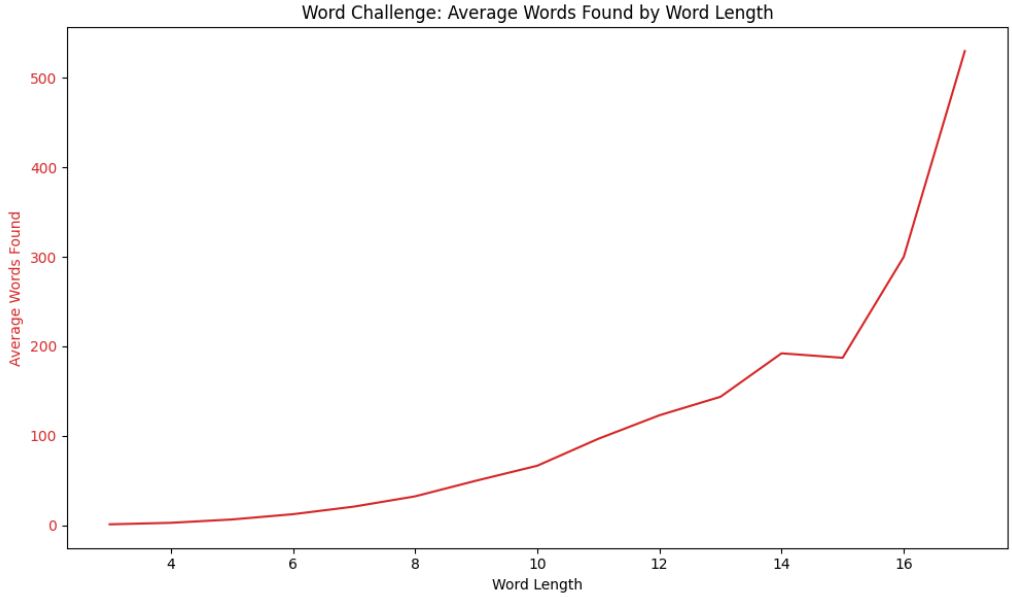
Generate possible words Both in interactive and automatic mode, the program then searches for all possible valid letter combinations in the dictionary. The generation of possible words from a given set of input letters in Word Challenge involves a process that combines algorithmics with the structural advantages of the Trie data structure. Using a backtracking algorithm, the program iteratively constructs candidate words by exploring every possible combination of the given letters. Each step in this recursive process extends a partial word by one letter, checking against the Trie to determine whether the current concatenation remains a prefix of a valid word. Paths that no longer correspond to any word prefix in the Trie are abandoned, significantly narrowing the search space. Upon reaching a node in the trie that marks the end of a valid word, the constructed word is saved as a successful match. The algorithm takes into account the number of letters, ensuring that each letter from the input set is used no more than its available number. This generation process not only guarantees the identification of all valid words within the constraints of the given set of letters, but does so with remarkable efficiency, taking advantage of the Trie's ability to quickly navigate through the set of possible words.

```
class Trie:

    def find_words(self, letters):
        found_words = set()

        def backtrack(node, path, letter_counts):
            if node.is_end_of_word:
                found_words.add(''.join(path))
            for char, child_node in node.children.items():
                if letter_counts.get(char, 0) > 0:
                    letter_counts[char] -= 1
                    path.append(char)
                    backtrack(child_node, path, letter_counts)
                    path.pop()
                    letter_counts[char] += 1

        letter_counts = {char: letters.count(char) for char in set(letters)}
        backtrack(self.root, [], letter_counts)
        return sorted(found_words, key=len)
```

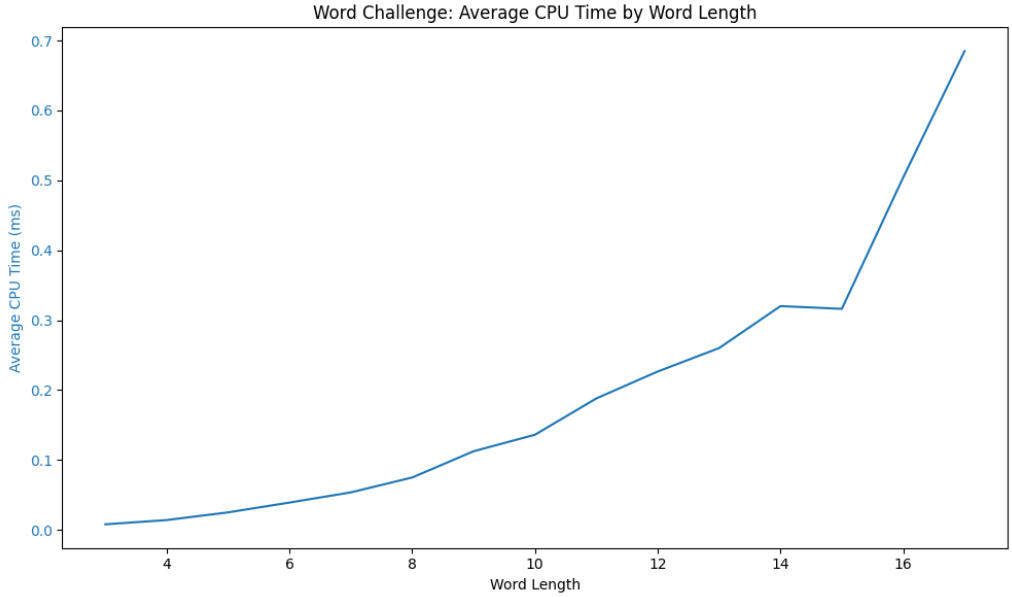
3.1 Benchmarks

To assess the performance of my implementation of Word Challenge I measured the average number of words found and the average CPU time for each word length. The benchmark was run with 10000

iterations for each word length.



The average number of words found seems to increase exponentially with word lengths. This is an expected result, since longer words can often be broken down into a larger number of smaller words, and the number of possible word combinations grows exponentially as more letters are available.



In addition, the computational complexity of finding all possible words also increases exponentially with word length, due to the larger number of letter combinations that the algorithm must process for longer words.

Overall, the results indicate that the computational load of the Word Challenges becomes significantly higher with longer words, suggesting the need for performance optimization for longer word lengths, which I have successfully implemented using a Trie structure.

4 Wordle

Wordle challenges players to guess a secret word through iterative guessing and strategic reasoning, guided by feedback provided for each guess. It can be interactively played in a guesser mode, where the player tries to find the secret word receiving feedback from the program and a keeper mode, where the player provides the feedback for the programs guesses. In automatic mode the program represents both the keeper and the guesser. This section outlines the strategies and data structures that support these core mechanics of the game.

Choose Word The selection of a random word (e.g. the secret word or the guess) is again done using the previously created *words_by_length* dictionary.

Generate Feedback After receiving the guesses from the guesser, the keeper generates a feedback. The used function compares the guess with the secret word and generates a string of digits, where '2' indicates a correctly placed letter, '1' indicates a correct letter in the wrong position, and '0' indicates a wrong letter.

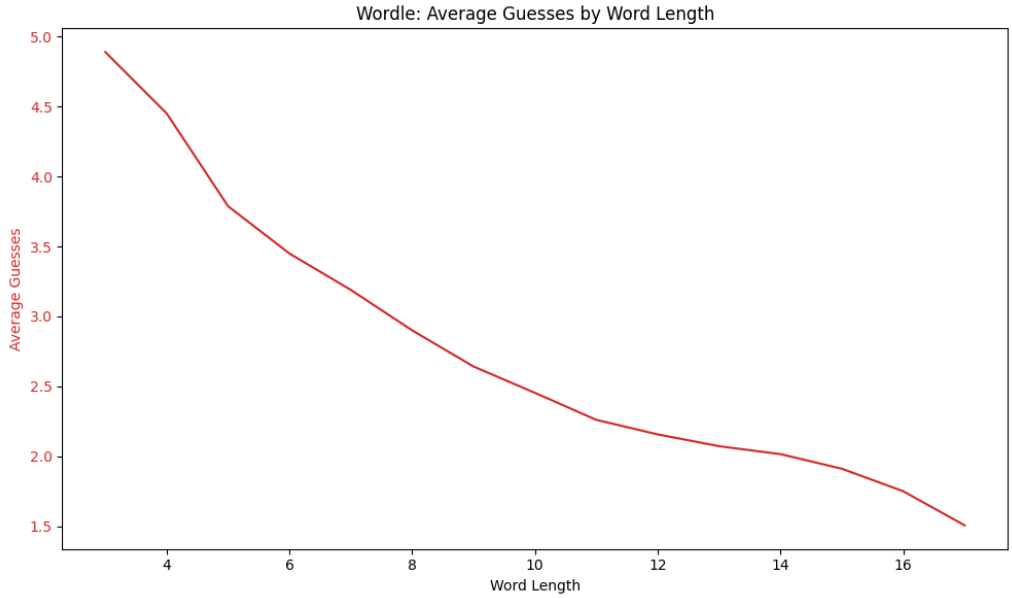
```
def generate_feedback(secret_word, guess):
    feedback = []
    for i, char in enumerate(guess):
        if char == secret_word[i]:
            feedback.append('2')  # Correct position
        elif char in secret_word:
            feedback.append('1')  # Wrong position
        else:
            feedback.append('0')  # Not in word
    return "".join(feedback)
```

Filter Words The guesser randomly selects the next guess from a list of valid words for each attempt, initially containing all words of the correct length. This list can then be refined based on the received feedback, excluding words that do not match the feedback pattern. This iterative filtering narrows down the possibilities, guiding the guesser toward the correct word efficiently.

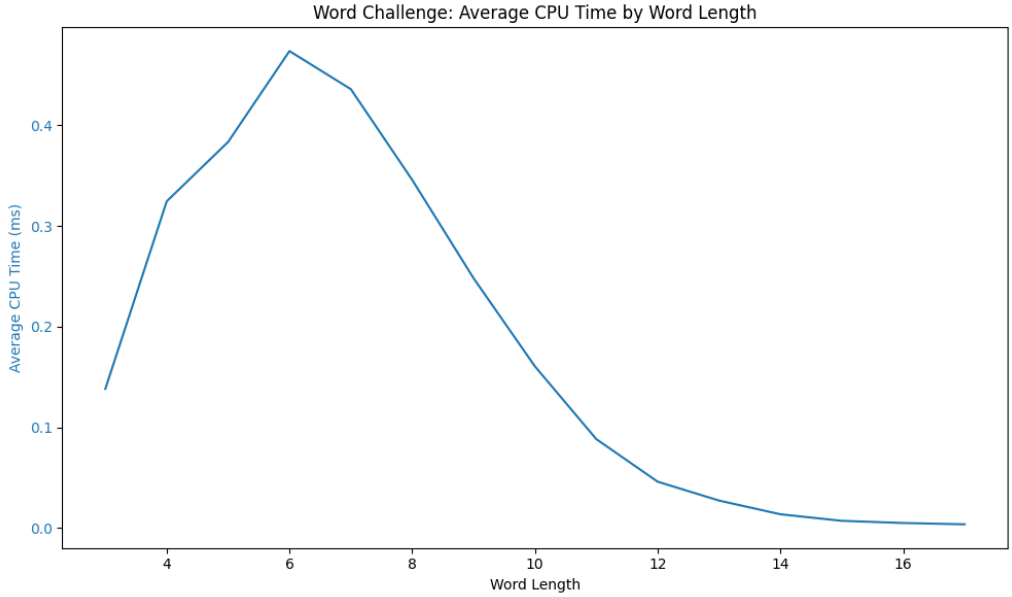
```
def filter_words(valid_words, guess, feedback):
    new_valid_words = []
    for word in valid_words:
        match = True
        for i, char in enumerate(guess):
            if feedback[i] == '2' and word[i] != char:
                match = False
                break
            if feedback[i] == '1' and (char not in word or word[i] == char):
                match = False
                break
            if feedback[i] == '0' and char in word:
                match = False
                break
        if match:
            new_valid_words.append(word)
    return new_valid_words
```

4.1 Benchmarks

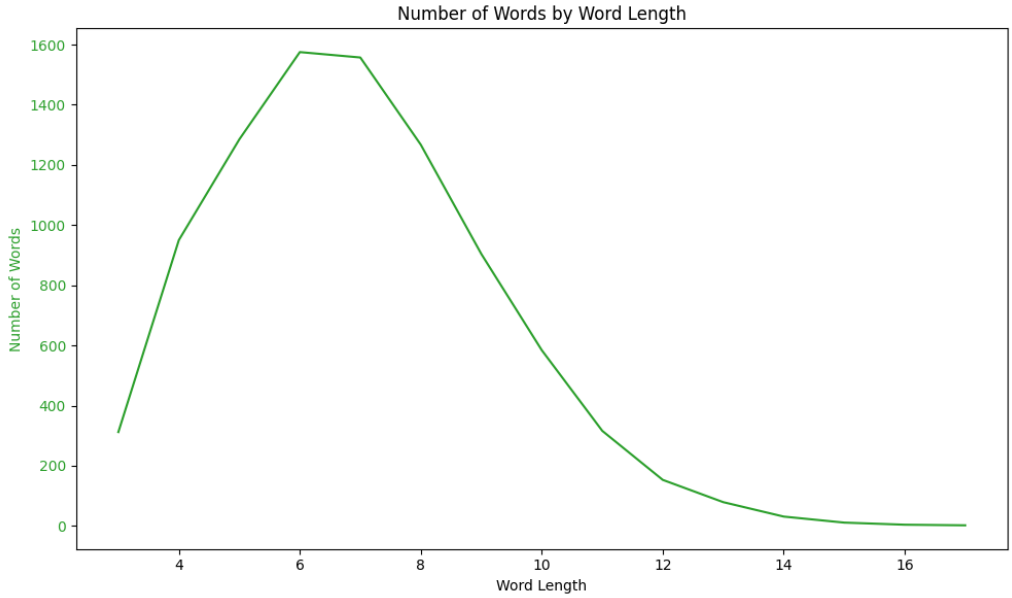
To assess the performance of my implementation of Wordle I measured the average guesses needed to guess the correct word and the average CPU time for each word length. The benchmark was run with 10000 iterations for each word length.



The first graph suggests a linear relationship between average guesses and word length: as word length increases, the average number of guesses required to solve the problem decreases. Several factors could cause this decrease, such as the ease of eliminating incorrect letters in longer words, or the shortage of longer words in the dictionary.



At the same time, we observe an interesting pattern where CPU time peaks at a certain word length of 6 before decreasing, which suggests a point of maximal computational complexity relative to the length of the words being processed. Considering the distribution of the number of words for each word length, we notice that the CPU time seems to strongly correlate to the number of words. Less possible words in the dictionary, simplify the search and decision-making process of the algorithm.



Overall, these trends in the Wordle game highlight a contrast to those observed in Word Challenge, demonstrating how different algorithmic strategies can yield varied performance across word lengths. This analysis underscores the importance of tailored optimization strategies for each game to maintain consistant performance.