Advanced Data Structures (ADS-MIRI): 4-Word Games

Arturo Lidueña

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya,
Barcelona, Spain
May 2024

Abstract :

This report documents the development of a small application designed to play two word-based games: Word Challenge and Wordle. The program utilizes a dictionary of English words of length three or more, provided in a file sorted alphabetically with accompanying frequency counts. The Word Challenge game allows the user to input a set of up to 17 letters, generating all possible words from those letters. An automatic mode simulates this process multiple times, calculating average performance metrics. The Wordle game lets the user play as either the guesser or the keeper, with feedback provided on each guess. An automatic mode simulates multiple rounds, outputting average performance data. The report details the algorithms and data structures used in both games, focusing on high-level explanations. Performance benchmarks and results are also presented.

1 Introduction

The application created as part of this assignment is capable of playing two games: Word Challenge and Wordle, using a dictionary of English words of length three or more, provided in the file `dictionary.txt`. Each entry in the dictionary includes a word and its frequency count, sourced from kaggle.com.

1.1 Word Challenge

In Word Challenge, the user inputs a (multi)set of up to 17 letters, and the program generates all possible words that can be formed using a subset of these letters. For example, given the letters {E, T, F, H, R, R, E, O, E}, the program will list words such as `FOR`, `HER`, `ORE`, `THE`, and `HERE`, ordered first by length and then alphabetically. The dictionary contains only words of length three or more, so results start with words of length three and end with words of length ℓ (or shorter if no longer words can be formed).

Additionally, the program includes an automatic mode where it repeatedly:

1. Selects a random word from the dictionary of a given length ℓ.
2. Randomly rearranges its letters.
3. Generates the list of possible words from these letters.

In this mode, the user specifies the number of games to be played and the word length ℓ, and the program outputs the average number of words found and the average CPU time taken to "solve" a word of length ℓ.

1.2 Wordle

In Wordle, the user can play as either the keeper or the guesser. The length ℓ of the secret word is determined at the start of the game. As the guesser, the program selects a secret word of length ℓ, and in each round, the player guesses a word of the same length. The program responds with a string of digits indicating the accuracy of the guess: 0 means the letter does not occur in the secret word, 1 means the letter occurs but not at the guessed position, and 2 means the letter is correctly positioned. For example, if the secret word is `WORDS` and the guess is `WHERE`, the output is `20010`. The game ends when the secret word is correctly guessed or a maximum number of guesses is reached. When the user plays as the keeper, the computer makes guesses and the user provides feedback.

The program also includes an automatic mode where

the computer acts as both keeper and guesser. The user sets a word length ℓ and the number of games to be played. For each game, a random word of length ℓ is chosen, and the game is played with the computer guessing based on the feedback provided. The program reports the average number of rounds needed to guess the secret word and the average CPU time for the task.

2 Implementation

The implementation involves creating a Ternary Search Tree (TST) for storing and querying dictionary words and managing the logic for the Word Challenge and Wordle games.

2.1 Ternary Search Tree (TST)

A Ternary Search Tree is a type of trie where the child nodes are organized in a binary search tree manner. Each node in the TST contains a character, pointers to the left, middle, and right children, a boolean flag indicating the end of a string, and a count of occurrences.

Algorithm 1 Ternary Search Tree Operations

```
function INSERT(s: string)
    root ← INSERT(root, s, 0)
end function
function ERASE(s: string)
    root ← ERASE(root, s, 0)
end function
function SEARCH(s: string) → boolean
    return SEARCH(root, s, 0)
end function
function FINDWORDS(s: string, foundWords: vector of string)
    FINDWORDS(root, s, 0, "", foundWords)
end function
function FINDPARTIALWORDS(partial_s: string, foundWords: vector of string)
    FINDPARTIALWORDS(root, partial_s, 0, "", foundWords)
end function
```

2.1.1 Node Structure Each node in the TST contains:

Algorithm 2 Node Structure

```
struct Node:
    char c                                  ▷ A character c
    bool isEndOfString                      ▷ indicating if the node marks the end of a word
    Node left, middle and right             ▷ children
    constructor Node(char ch):
        c(ch)
        isEndOfString(false)
        left ← NULL
        middle ← NULL
        right ← NULL
```

2.1.2 Insertion To insert a word into the TST, the algorithm compares each character of the word with the character in the current node:

Algorithm 3 Ternary Search Tree - Insertion

```
function INSERT(node: Node, s: string, i: integer) → Node
    if node = null then
        node ← new Node(s[i])
    end if
    if s[i] < node.c then
        node.left ← INSERT(node.left, s, i)
    else if s[i] > node.c then
        node.right ← INSERT(node.right, s, i)
    else
        node.count ← node.count + 1
        if i + 1 < length(s) then
            node.mid ← INSERT(node.mid, s, i + 1)
        else
            node.isEndOfString ← true
        end if
    end if
    return node
end function
```

- If the character is smaller, the algorithm proceeds to the left child.
- If the character is larger, the algorithm proceeds to the right child.
- If the character matches, the algorithm proceeds to the middle child.

- The process continues until all characters are inserted, with the final node marked as the end of the word.

2.1.3 Search Searching for a word involves traversing the TST by comparing characters similarly to insertion. The search continues until the end of the word is reached or the word is not found.

Algorithm 4 Ternary Search Tree - Search

```
function SEARCH(node: Node, s: string, i: integer) → boolean
    if node = null then
        return false
    end if
    if s[i] < node.c then
        return SEARCH(node.left, s, i)
    else if s[i] > node.c then
        return SEARCH(node.right, s, i)
    else
        if i + 1 = length(s) then
            return node.isEndOfString
        else
            return SEARCH(node.mid, s, i + 1)
        end if
    end if
end function
```

2.1.4 Finding Words To find all words that can be formed from a given set of letters, a recursive function traverses the TST and collects words that can be constructed using the provided letters. The function ensures that words are collected in increasing length and alphabetical order.

Algorithm 5 Ternary Search Tree - findWords

```
function FINDWORDS(node: Node, s: string, i: integer, prefix: string, foundWords: vector of string)
    if node = null then
        return
    end if
    if s[i] < node.c then
        FINDWORDS(node.left, s, i, prefix, foundWords)
    else if s[i] > node.c then
        FINDWORDS(node.right, s, i, prefix, foundWords)
    else
        if node.isEndOfString then
            FOUNDWORDS.PUSH_BACK(prefix + node.c)
        end if
        FINDWORDS(node.mid, s, i + 1, prefix + node.c, foundWords)
    end if
end function
```

2.1.5 Finding Partial Words To find partial words, a similar recursive function is used that accounts for possible partial matches and collects them.

Algorithm 6 Ternary Search Tree - findPartialWords

```
function findPartialWords(node: Node, s: string, i: integer, prefix: string, foundWords: vector of string)
    if node = null then
        return
    end if
    if s[i] = '0' then
        if i + 1 = length(s) and node.isEndOfString then
            foundWords.push_back(prefix + node.c)
            return
        end if
        findPartialWords(node.left, s, i, prefix, foundWords)
        findPartialWords(node.mid, s, i+1, prefix+node.c, foundWords)
        findPartialWords(node.right, s, i, prefix, foundWords)
    else if s[i] < node.c then
        findPartialWords(node.left, s, i, prefix, foundWords)
    else if s[i] > node.c then
        findPartialWords(node.right, s, i, prefix, foundWords)
    else
        if i + 1 = length(s) then
            foundWords.push_back(prefix + node.c)
            return
        end if
        findPartialWords(node.mid, s, i+1, prefix+node.c, foundWords)
    end if
end function
```

3 Word Challenge Implementation

The implementation of the Word Challenge game involves several key components and algorithms. The primary data structure used to store the dictionary words is the Ternary Search Tree (TST). This section explains the TST and its role in the Word Challenge game.

3.1 Automatic Mode

In automatic mode, the program repeatedly selects a random word from the dictionary, scrambles its letters, and finds all possible words from the scrambled letters. It measures and reports the average number of words found and the average CPU time.

3.1.1 Random Word Selection A random word of the specified length is selected from the dictionary. The letters of this word are then randomly rearranged to create different permutations.

Algorithm 7 Random Word Selection and Scrambling

```
function selectRandomWord(dictionary: vector of string, length: int) → string
    words ← filter(dictionary, function(word) returns length(word) = length)
    randomIndex ← randomInt(0, size(words) - 1)
    randomWord ← words[randomIndex]
    return randomWord
end function
function scrambleWord(word: string) → string
    letters ← toVector(word)
    shuffle(letters)
    return toString(letters)
end function
```

3.1.2 Finding All Possible Words The function `findWords` finds all possible words from the scrambled letters using the Ternary Search Tree (TST).

Algorithm 8 Find All Possible Words

```
function findWords(scrambledWord: string) → vector of string
    foundWords ← empty vector of string
    findWordsHelper(root, scrambledWord, 0, "", foundWords)
    return foundWords
end function
function findWordsHelper(node: Node, scrambledWord: string, i: int, prefix: string, foundWords: vector of string)
    if node = null then
        return
    end if
    if scrambledWord[i] < node.c then
        findWordsHelper(node.left, scrambledWord, i, prefix, foundWords)
    else if scrambledWord[i] > node.c then
        findWordsHelper(node.right, scrambledWord, i, prefix, foundWords)
    else
        if node.isEndOfString then
            foundWords.push_back(prefix + node.c)
        end if
        if i + 1 < length(scrambledWord) then
            findWordsHelper(node.mid, scrambledWord, i + 1, prefix + node.c, foundWords)
        else
            findWordsHelper(node.mid, scrambledWord, i, prefix + node.c, foundWords)
        end if
    end if
end function
```

4 Wordle Implementation

The implementation of the Wordle game involves several components: the game manager, the keeper, and the guesser. Each component has a specific role in managing the game flow and interactions.

4.1 WordleGame Class

The `WordleGame` class manages the overall game state, including the maximum number of guesses allowed and the current guess count.

4.1.1 Constructor The constructor initializes the maximum number of guesses and resets the guess count to zero.

Algorithm 9 WordleGame Constructor

```
function WordleGame::init(maxGuesses: int)
    this.maxGuesses ← maxGuesses
    this.guessCount ← 0
end function
```

4.1.2 isCorrectGuess This method checks if the provided feedback indicates a correct guess. If the feedback string consists entirely of '2's, it indicates that the guess is correct.

Algorithm 10 isCorrectGuess Method

```
function WordleGame::isCorrectGuess(feedback: string) → bool
    for char in feedback do
        if char ≠ 2 then
            return false
        end if
    end for
    return true
end function
```

4.1.3 isGameOver This method checks if the game has reached the maximum number of guesses.

Algorithm 11 isGameOver Method

```
function WordleGame::isGameOver → bool
    return this.guessCount ≥ this.maxGuesses
end function
```

4.1.4 getGuessCount This method returns the current number of guesses made in the game.

Algorithm 12 getGuessCount Method

```
function WordleGame::getGuessCount → int
    return this.guessCount
end function
```

4.2 WordleKeeper Class

The `WordleKeeper` class is responsible for managing the secret word and providing feedback on guesses.

4.2.1 Constructor The constructor initializes the secret word by selecting a random word of the specified length from the dictionary. It also creates a set of characters in the secret word for quick lookup.

Algorithm 13 WordleKeeper Constructor

function WordleKeeper::init(*dictionary*: vector of string, *length*: int)
    $this.secretWord \leftarrow$ selectRandomWord(*dictionary*, *length*)
    $this.secretChars \leftarrow$ toSet($this.secretWord$)
end function

4.2.2 feedback This method generates feedback for a given guess. The feedback string consists of '0's, '1's, and '2's, indicating whether each letter in the guess is absent, present but incorrectly positioned, or correctly positioned, respectively.

Algorithm 14 feedback Method

function WordleKeeper::feedback(*guess*: string) → string
    $feedback \leftarrow$ empty string
    for $i \leftarrow 0$ **to** length(*guess*) - 1 **do**
        if $guess[i] = this.secretWord[i]$ **then**
            $feedback \leftarrow feedback +$ "2"
        else if $guess[i] \in this.secretChars$ **then**
            $feedback \leftarrow feedback +$ "1"
        else
            $feedback \leftarrow feedback +$ "0"
        end if
    end for
    return $feedback$
end function

4.3 WordleGuesser Class

The `WordleGuesser` class manages the guessing logic, keeping track of previous guesses and feedback.

4.3.1 Constructor The constructor initializes the length of the word, feedback and memo strings, and sets to store guesses and mandatory characters.

Algorithm 15 WordleGuesser Constructor

function WordleGuesser::init(*length*: int)
    $this.length \leftarrow length$
    $this.feedback \leftarrow$ empty string
    $this.memo \leftarrow$ empty string
    $this.guesses \leftarrow$ empty set
    $this.mandatoryChars \leftarrow$ empty set
end function

4.3.2 makeGuess This method generates a new guess based on the current state of the game, using the Ternary Search Tree to find potential words. It filters out words that have already been guessed or do not contain mandatory characters.

Algorithm 16 makeGuess Method

function WordleGuesser::makeGuess(*tst*: TernarySearchTree) → string
    $potentialWords \leftarrow$ findAllWords(*tst*, this.length) **for** word*in*potentialWords **do**
        $word \notin this.guesses$ **and** containsAllChars(*word*, $this.mandatoryChars$)
        $this.guesses.insert(word)$
        return $word$

    return ""
end function

4.3.3 updateFeedback This method updates the internal state of the guesser based on the feedback from the keeper. It records correctly guessed letters and updates the memo and mandatory characters sets.

Algorithm 17 updateFeedback Method

```
function WordleGuesser::updateFeedback(guess: string, feedback: string)
    for i ← 0 to length(feedback) - 1 do
        if feedback[i] = "2" then
            this.memo[i] ← guess[i]
        else if feedback[i] = "1" then
            this.mandatoryChars.insert(guess[i])
        end if
    end for
end function
```

4.4 Interactive Mode

In interactive mode, the user can choose to play as either the keeper or the guesser. The game proceeds with the user making guesses and receiving feedback or the computer making guesses and the user providing feedback.

4.4.1 Playing as Keeper The user provides feedback on each guess made by the computer until the correct word is guessed or the maximum number of guesses is reached.

Algorithm 18 Playing as Keeper

```
function playAsKeeper(wordleGame: WordleGame, wordleKeeper: WordleKeeper, wordleGuesser: WordleGuesser)
    while not wordleGame.isGameOver do
        guess ← wordleGuesser.makeGuess(tst)
        feedback ← wordleKeeper.feedback(guess)
        wordleGuesser.updateFeedback(guess, feedback)
        if wordleGame.isCorrectGuess(feedback) then
            print("Guesser found the word!")
            return
        end if
    end while
    print("Game over. Secret word was: ") wordleKeeper.print_secret
end function
```

4.4.2 Playing as Guesser The user makes guesses based on the feedback provided by the computer, attempting to guess the secret word within the allowed number of guesses.

Algorithm 19 Playing as Guesser

```
function playAsGuesser(wordleGame: WordleGame, wordleKeeper: WordleKeeper)
    while not wordleGame.isGameOver do
        guess ← input("Enter your guess: ")
        feedback ← wordleKeeper.feedback(guess)
        print("Feedback: ") feedback
        if wordleGame.isCorrectGuess(feedback) then
            print("Congratulations! You guessed the word!")
            return
        end if
    end while
    print("Game over. Secret word was: ") wordleKeeper.print_secret
end function
```

4.5 Automatic Mode

In automatic mode, the computer simulates multiple games, acting as both the keeper and the guesser. Performance metrics such as the average number of rounds and CPU time per game are calculated and reported.

4.5.1 Simulation The simulation involves running the game multiple times, with the computer selecting a random word, making guesses, and updating feedback automatically.

Algorithm 20 Automatic Simulation

```
function simulateGames(dictionary: vector of string, iterations: int, wordLength: int, maxGuesses: int)
    totalRounds ← 0
    totalTime ← 0
    for i ← 1 to iterations do
        wordleGame ← WordleGame(maxGuesses)
        wordleKeeper ← WordleKeeper(dictionary, wordLength)
        wordleGuesser ← WordleGuesser(wordLength)
        startTime ← currentTime
        while not wordleGame.isGameOver do
            guess ← wordleGuesser.makeGuess(tst)
            feedback ← wordleKeeper.feedback(guess)
            wordleGuesser.updateFeedback(guess, feedback)
            wordleGame.guessCount ← wordleGame.guessCount + 1
            if wordleGame.isCorrectGuess(feedback) then
                break
            end if
        end while
        endTime ← currentTime
        totalRounds ← totalRounds + wordleGame.guessCount
        totalTime ← totalTime + (endTime − startTime)
    end for
    averageRounds ← totalRounds/iterations
    averageTime ← totalTime/iterations
    return (averageRounds, averageTime)
end function
```

5 Experiment Setup for Word Challenge and Wordle

Dictionary Loading: A Ternary Search Tree (TST) is used to load the dictionary from a text file. The success of this operation is checked before proceeding with the experiments.

5.1 Setup for Word Challenge Experiments

The Word Challenge experiments aim to evaluate the performance of finding all possible words from scrambled letters of a randomly selected word. The setup is detailed as follows:

1. **Dictionary Loading**
2. **Parameters:** The experiments are conducted for various word lengths ranging from 3 to 10 letters. For each word length, the number of games played is set to 5, 10, and 20 games.
3. **CSV Logging:** An output CSV file is prepared to log the results of these experiments, with columns for word length, number of games, average number of words found, and average time taken.
4. **Experiment Loop:** For each combination of word length and number of games:
    - A random word of the specified length is selected from the dictionary.
    - The letters of the word are scrambled randomly.
    - The game records the time taken to find all valid words from the scrambled letters using the dictionary.
    - The number of valid words found and the time taken are recorded.
5. **Data Aggregation:** After running all games for each parameter set, the average number of words found and the average time taken are calculated and logged into the CSV file.

5.2 Setup for Wordle Experiments

To evaluate the performance of our Wordle implementation, we conducted a series of experiments by varying the word lengths and the number of games played. The experimental setup is structured as follows:

1. **Dictionary Loading**
2. **Parameters:** The experiments test various word lengths ranging from 3 to 20 letters. For each word length, the number of games played is varied, typically set to 5, 10, and 20 games.
3. **CSV Logging:** An output CSV file is prepared to log the results of the experiments. The CSV file contains columns for word length, number of games,

average number of rounds, and average time taken per game.

4. **Experiment Loop:** For each combination of word length and number of games:
    - A new WordleKeeper, WordleGuesser, and WordleGame instance are created.
    - The game runs until the maximum number of guesses is reached or the correct word is guessed.
    - The number of rounds taken and the time elapsed for each game are recorded.
5. **Data Aggregation:** After running all games for a given combination of parameters, the average number of rounds and average time taken are calculated and written to the CSV file.

6 Results

6.1 Result Word Challenge Experiments

The results from the Word Challenge experiments focus on the average number of words found and the average time taken to find those words across different word lengths and numbers of games played.

- **Average Words Found:** This metric indicates the average number of valid words found per game.
- **Average Time:** Represents the average time taken (in seconds) per game.

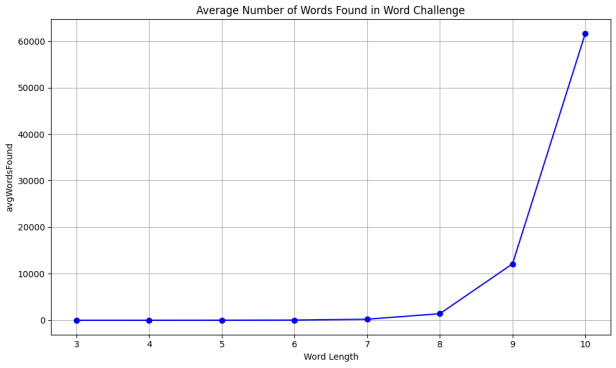

Figure 1. Average Number of Words Found in Word Challenge

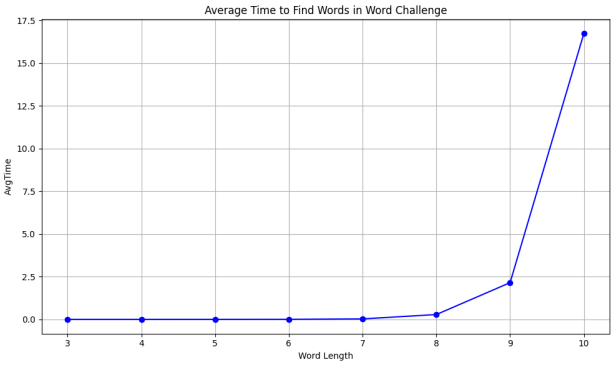

Figure 2. Average Time to Find Words in Word Challenge

6.1.1 Analysis

- **Impact of Word Length:** Shorter words (e.g., 3 to 5 letters) tend to result in finding fewer average words compared to longer words. This is due to the larger set of permutations and combinations possible with longer words.
- **Consistency Across Games:** The number of games (5, 10, 20) shows consistent trends across different word lengths, indicating stable performance metrics in terms of average words found and average time.
- **Performance Metrics:** Both average words found and average time show trends that increase as word length increases. Longer words are harder to find.

6.2 Result Wordle Experiments

The results from the Wordle experiments provide insights into the performance of the game based on varying word lengths and the number of games played. Here are the summarized findings:

- **Average Rounds:** This metric indicates the average number of guesses (rounds) taken to either correctly guess the word or exhaust all allowed guesses.
- **Average Time:** Represents the average time taken (in seconds) per game.

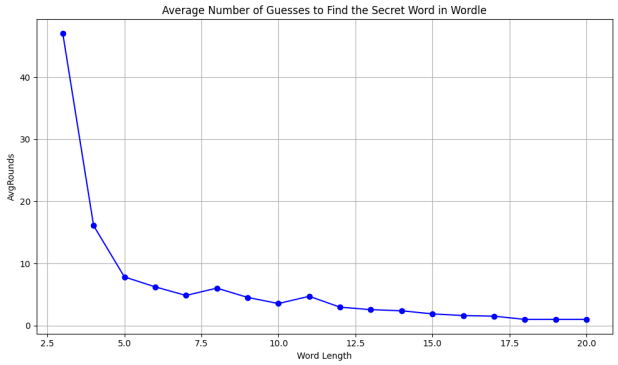

Figure 3. Average Number of Guesses to Find the Secret Word in Wordle

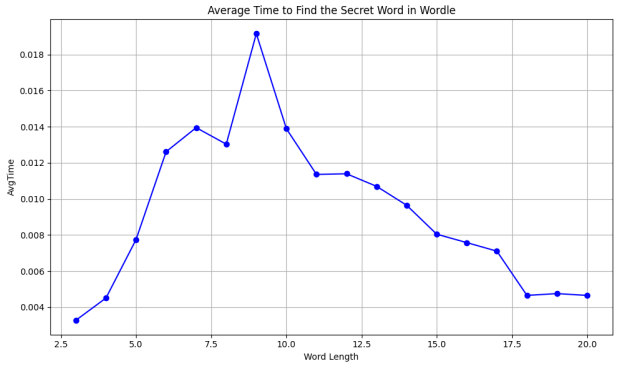

Figure 4. Average Time to Find the Secret Word in Wordle

6.2.1 Analysis

- **Impact of Word Length:** As expected, shorter words (e.g., 3 to 5 letters) generally require more rounds to guess correctly compared to longer words. This is likely due to the higher number of possible combinations for shorter words, leading to more iterations needed to find the correct one.
- **Performance Metrics:** Both average rounds and average time exhibit expected trends—decreasing as word length increases. This pattern indicates that longer words are generally easier to guess within fewer rounds and less time, likely due to their more distinctive letter patterns.

7 Conclusion

The experiments on the Word Challenge and Wordle games provide valuable insights into the performance and complexity of word-based puzzles. Key findings from each game are summarized below.

7.1 Word Challenge Game

The Word Challenge results highlight the following:

- **Average Words Found:** Longer words yield more valid words due to a larger set of permutations.
- **Average Time:** The time required increases with word length, reflecting greater computational complexity.

7.2 Wordle Game

The Wordle results reveal the following:

- **Average Rounds:** Fewer guesses are needed for longer words due to distinctive letter patterns.
- **Average Time:** Time per game decreases as word length increases.
- **Impact of Word Length:** Shorter words require more rounds and time to guess correctly.