Document Checker Using String Matching and Regular Expression

Benardo - 13522055
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13522055@std.stei.itb.ac.id

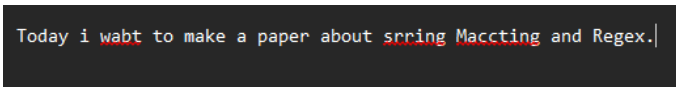
***Abstract*— In this paper, the authors present a document checker using string matching and regular expressions. Today, documents are essential in various fields, including education, work, law, and many others. In some fields, documents must be written perfectly, without any typos or errors. Manually checking each word can be time-consuming, especially if the document contains thousands or even tens of thousands of words. This paper aims to create a document checker to help users review their documents efficiently. The methodology involves implementing the Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM) algorithms for string matching. For word suggestions, Levenshtein Distance is used. Other checks, such as sentence structure, capitalization at the beginning of sentences, avoiding excessive spaces, and ensuring each sentence ends with proper punctuation, are handled using regular expressions. This tool aims to assist users in producing error-free documents with greater ease and accuracy.**

Keywords*— String Matching; Knuth-Morris-Pratt Algorithm; Boyer-Moore Algorithm; Levenshtein Distance; regular expressions; document checker

I. Introduction

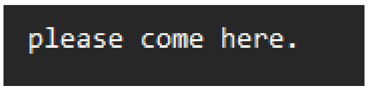
In today's world, documents play a crucial role across various domains such as education, professional work, law, and more. The necessity for accurate and error-free documentation is paramount in many of these fields. For instance, legal documents must be written without any typographical errors or mistakes to ensure clarity and precision. Similarly, academic and professional writings need to maintain high standards to convey the intended information effectively. However, manually checking documents for errors is a time-consuming and tedious task, especially when dealing with lengthy texts containing thousands or even tens of thousands of words. This process not only demands considerable time and effort but is also prone to human error. The traditional approach of proofreading each word and sentence is inefficient and can lead to oversight of subtle mistakes.

A document checker is a tool designed to automate the process of identifying and correcting errors in a text document. This tool can significantly enhance the efficiency and accuracy of the proofreading process by leveraging advanced algorithms and techniques to detect various types of errors, including typographical errors, grammatical mistakes, and stylistic inconsistencies. The need for a document checker arises from the increasing volume of written communication in both personal and professional contexts. With the growing reliance on digital documentation, ensuring the correctness and quality of written content has become more critical than ever. A document checker can help individuals and organizations maintain high standards in their written communication, thereby improving overall productivity and reducing the risk of miscommunication or misinterpretation.



```
Today i wabt to make a paper about srring Maccting and Regex.
```

Figure 1. Example Typo in Sentences
Source : Author’s personal documentation



```
please come here.
```

Figure 2. Example of Lacking Noun on a Sentences
Source : Author's personal documentation

This paper aims to develop a robust document checker that employs string matching algorithms and regular expressions to detect and correct errors in text documents. Specifically, the Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM) algorithms are utilized for efficient pattern matching, while the Levenshtein distance algorithm provides word correction suggestions. Additionally, the tool incorporates regular expressions to enforce grammatical rules such as capitalization at the beginning of sentences, proper punctuation, and avoidance of excessive spaces.

By addressing the challenges associated with manual proofreading, this paper seeks to create a tool that not only enhances the accuracy and quality of documents but also saves time and effort for users. The proposed document checker aims to be a valuable asset for writers, educators, professionals, and anyone who engages in extensive written communication, ensuring their documents are error-free and well-structured.

II. Theory and Concepts

A. *String Matching*

String matching, also known as pattern matching, is an algorithm used to locate occurrences of a specific pattern within a large text. This fundamental algorithm has widespread applications, including search engines, text recognition systems, natural language processing, and more. The primary goal of string matching algorithms is to identify positions where the pattern matched a substring within a text. In string matching, the text is typically a long string of character (with the length m), while the pattern is a shorter string that we are looking for within a text (with the length n), so we can assume that m << n.

There are two concepts that related in string matching:

1. Prefix: Prefix is a substring that appears at the beginning of a word and can alter the meaning of the original word. Prefix must be located between [0…m - 1].

2. Suffix: Suffix is a substring that appears at the end of a word dan can alter the meaning of the original word. Suffix must be located between [1...m-1].

String matching algorithms can be categorized based on the direction in which they search:

1. From Left to Right
   This is the most natural direction, like how we read text. Algorithms in this category include the Brute-Force algorithm and the Knuth-Morris-Pratt (KMP) algorithm.
2. From Right to Left
   Searching from right to left often yields better practical results. An example of an algorithm in this category is the Boyer-Moore algorithm.
3. In a specific order
   Some algorithms search in a specific order determined by the algorithm itself, which can provide the best theoretical results. Examples include the Colussi algorithm and the Crochemore-Perrin algorithm.

Generally, there are 3 main algorithms for performing string matching:

1. Brute Force Algorithm
   The Brute Force algorithm, also known as the Naïve algorithm, is the simplest form of string matching. It operates by sequentially matching the pattern against every possible position in the text until a match is found or the text is completely searched. The process involves:
   a. Comparing the first character of the pattern with consecutive characters of the text.
   b. If a match is found, the subsequent characters are compared until the characters do not match or the entire pattern matches with a substring in the text.
   c. If a mismatch occurs at any point, the pattern is shifted by one character to the right, and the comparison begins anew from the first character of the pattern.
   d. This continues until the pattern is either found within the text or the search reaches the end of the text.

```
   NOBODY NOTICED HIM
 1 NOT
 2  NOT
 3   NOT
 4    NOT
 5     NOT
 6      NOT
 7       NOT
 8        NOT
```

Figure 3. Example of Bruteforce algorithm

Source : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

   The primary disadvantage of the Brute Force algorithm is its time complexity, which is typically O(nm) in the worst-case scenario, where n is the length of the text and m is the length of the pattern. For the best-case scenario, the complexity is O(m), occurring when the first character of the pattern fails to match the first character of the text. This makes the Brute Force algorithm inefficient for large texts or when the pattern is considerably long.

2. Knuth-Morris-Pratt Algorithm (KMP)

   The Knuth-Morris-Pratt Algorithm is developed independently by Donald Knuth, Vaughan Pratt, and James Morris in the 1970s. The KMP algorithm enhances the efficiency of string matching by avoiding redundant comparisons. The Knuth-Morris-Pratt (KMP) algorithm looks for the pattern in the text in a left-to-right order (like the brute force algorithm), but it shifts the pattern more intelligently than the brute force algorithm. The KMP process involves:
   a. Starting the comparison of the pattern with the text from the leftmost side.
   b. If the character matches, it will continue comparison to the next indices of both the text and the pattern.
   c. If the character in the text is not match the character in the pattern, it will find the largest prefix of P[0...j-1] that is a suffix of P[1..j-1]
   d. The process continues until the pattern is found or the text is fully searched.

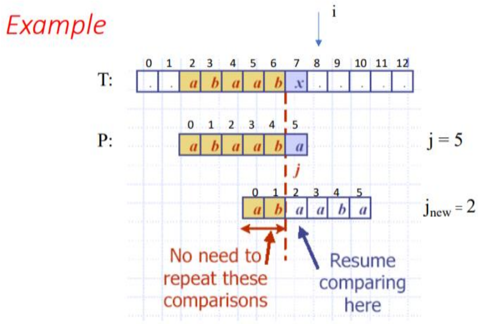


Figure 4. Example of KMP algorithm
Source :
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

The Knuth-Morris-Pratt (KMP) algorithm significantly enhances the efficiency of string matching by utilizing a concept known as the border function. This function plays a critical role in determining how far the pattern should be shifted when a mismatch occurs during the matching process.

The border function, denoted as b(k), is defined as the length of the longest prefix of the pattern P[0...k] that also serves as a suffix within the substring P[1..k]. Here, k represents the position in the pattern immediately preceding the point of mismatch. Essentially, the border function calculates the greatest extent to which a substring around the mismatch can still be considered a repetitive segment of the beginning of the pattern. This ensures that upon a mismatch, the algorithm does not reevaluate characters that have already been matched.

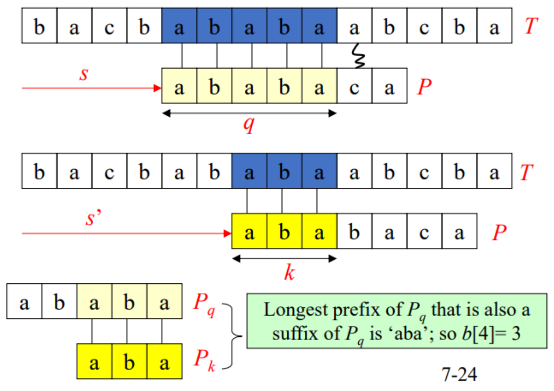

T
s
P
q
s'
k
P_q
P_k
Longest prefix of P_q that is also a suffix of P_q is 'aba'; so $b[4]= 3$
7-24

Figure 5. Example of KMP Algorithm using Border Function
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

The preprocessing of the pattern ensures that the KMP algorithm significantly reduces the worst-case complexity to O(m + n).

3. Boyer-Moore Algorithm (BM)

The Boyer-Moore algorithm represents a significant advancement in string matching, offering a more efficient approach than its predecessors like the Brute Force and Knuth-Morris-Pratt algorithms. It is particularly effective for large text sizes or when the alphabet of the text is relatively small. This algorithm is distinguished by its use of two primary techniques:

A. Looking Glass Technique

The Boyer-Moore algorithm initiates its search from the end of the pattern, moving backwards—a method known as the Looking Glass Technique. This approach allows the algorithm to skip over large sections of the text, thereby reducing the number of comparisons. By starting at the last character of the pattern and comparing backwards, it maximizes the potential for skipping characters that do not match, accelerating the search process significantly.

B. Character Jump Technique

When a mismatch occurs, the Boyer-Moore algorithm employs the Character Jump Technique. This technique determines how far the pattern should be shifted along the text. The shift depends on two main scenarios:

1. Last Occurrence within the Pattern

If the mismatched character is found elsewhere in the pattern, the algorithm shifts the pattern so that the last occurrence of the mismatched character aligns with its current position in the text. This is often the most significant jump, skipping as many characters as possible that do not need to be checked again.

2. Character Not Found

If the mismatched character does not appear in the pattern, the pattern is shifted completely past this character in the text.

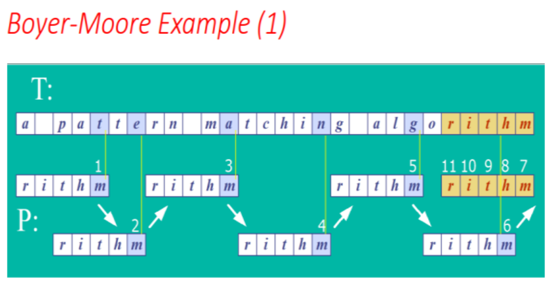
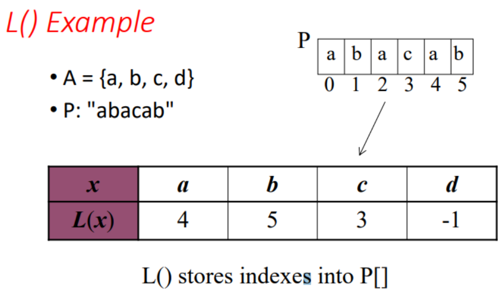


Figure 6. Example of BM Algorithm
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

To search for the last occurrence of a character, Boyer-Moore Algorithm has a function that is called last occurrence function or called L function. This function maps all characters in P into an integer that represents the last occurrence index. If the character doesn't exist in the pattern, the value will be -1. Below is the example of last occurrence function



L() Example

- A = {a, b, c, d}
- P: "abacab"

P	a	b	a	c	a	b
	0	1	2	3	4	5

x	a	b	c	d
$L(x)$	4	5	3	-1

L() stores indexes into P[]

Figure 7. Example of Last Occurrence Function
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

The complexity of Boyer-Moore algorithm for its worst case is O (nm + A). Boyer-Moore is fast when the alphabet (A) is large, slow when the alphabet is small). For searching English text, Boyer-Moore is significantly faster than brute force.

B. *Levenshtein Distance*

Levenshtein distance, also known as edit distance, quantifies the dissimilarity between two text strings by counting the minimum number of operations required to transform one string into another. These operations consist of insertions, deletions, and substitutions of a single character.

The essence of the Levenshtein distance is to determine how many edits are needed to convert one word into another word. For instance, to transform "INTENTION" to "EXECUTION", we need five operations: three substitution, one insertion, and one deletion, so the Levenshtein distance result is five, indicating that five edits are required to equate two words.

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

What?

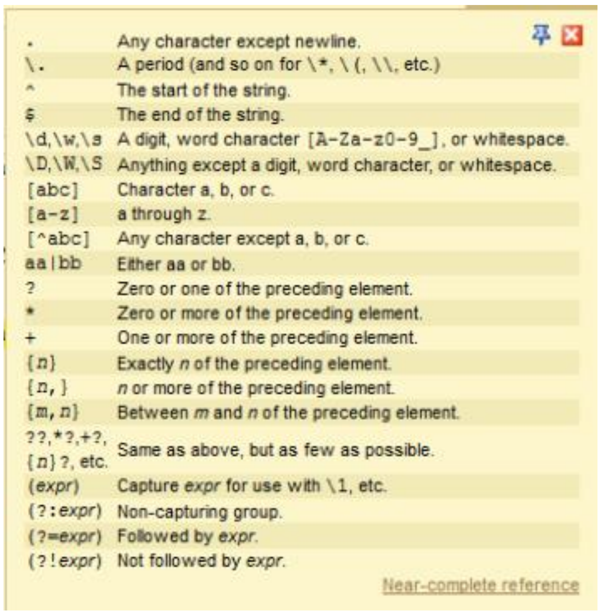
Figure 8. Levenshtein Distance in Mathematical Form
Source:
https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0

C. *Regular Expressions*

Regular Expressions or regex are patterns used to match sequences of character within strings. This powerful tool is integral for searching and manipulating text, making it invaluable for string processing tasks across various applications and programming environments. Regular expressions are supported across various programming and scripting languages, including Python, PHP, Java and others. Regex provides a compact and efficient means to perform text searches, replacement, and manipulation in one single code.

Regular expressions are composed of literals and metacharacters. Metacharacters are special characters that enhance the capabilities of regex by allowing more complex matching rules. Here are some fundamental components used in regular expressions:

1. Character Classes
   Specify a set of characters to match from within a text. For example, [abc] will match any single character of 'a', 'b', or 'c'
2. Negated Character Classes
   Denoted by [^abc], matches any character except 'a', 'b', or 'c'
3. Range Matches
   Such as [a-zA-Z], which matches any alphabetical character
4. Escape Sequences
   To match a character having special meaning in regex, you need to use a escape sequence prefix with a backslash (\). E.g., \. matches "."



.	Any character except newline.
\\.	A period (and so on for *, \\(, \\\\, etc.)
^	The start of the string.
$	The end of the string.
\\d,\\w,\\s	A digit, word character [A-Za-z0-9_], or whitespace.
\\D,\\W,\\S	Anything except a digit, word character, or whitespace.
[abc]	Character a, b, or c.
[a-z]	a through z.
[^abc]	Any character except a, b, or c.
aa | bb	Either aa or bb.
?	Zero or one of the preceding element.
*	Zero or more of the preceding element.
+	One or more of the preceding element.
{*n*}	Exactly *n* of the preceding element.
{*n*,}	*n* or more of the preceding element.
{*m*,*n*}	Between *m* and *n* of the preceding element.
??,*?,+?, {*n*}?, etc.	Same as above, but as few as possible.
(*expr*)	Capture *expr* for use with \\1, etc.
(?:*expr*)	Non-capturing group.
(?=*expr*)	Followed by *expr*.
(?!*expr*)	Not followed by *expr*.

Near-complete reference

Figure 9. Regex Syntax
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/String-Matching-dengan-Regex-2019.pdf

III. IMPLENETATION

In this chapter, the author will explain the functionality of the program, which is implemented in Python and utilizes string matching algorithms such as Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM) to ensure that each sentence contains at least one verb and one noun. The program will also implement Levenshtein distance algorithm to give recommendation word for the word that doesn't exist int the dataset. Additionally, regular expressions (regex) are employed to verify the correctness of sentences.

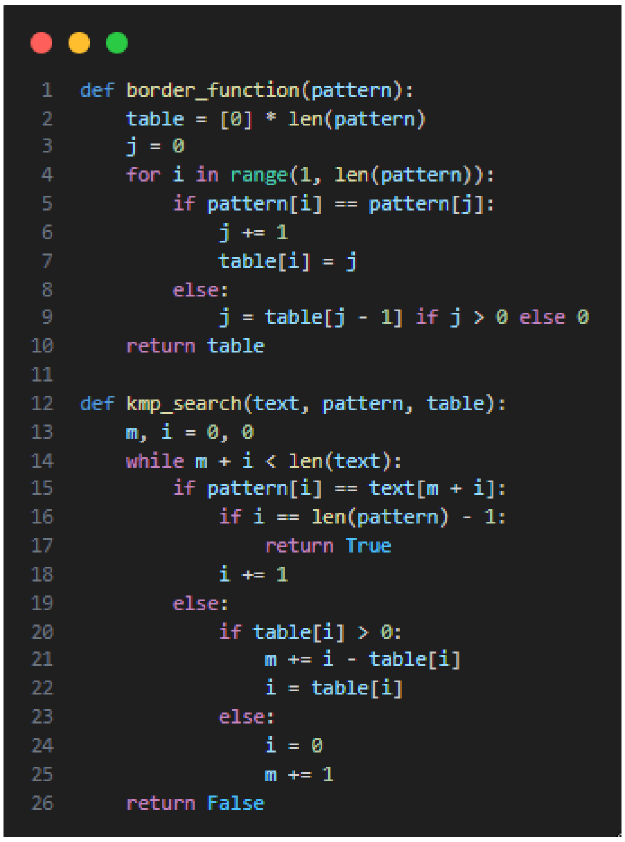
In this program, author use Kaggle's datasets that contains approximately 370000 English words with tags.

(source: https://www.kaggle.com/datasets/ruchi798/part-of-speech-tagging)

However, the dataset contains several mistakes in its tags, so the author uses Python libraries such as spaCy to correct the tags for each incorrectly tagged word.

A. *Knuth-Morris-Pratt (KMP) Algoritm Implementation*

KMP Algorithm is divided into two functions, which are 'kmp_search' and 'border_function'. The 'kmp_search' function accepts three inputs, such as text, pattern, and table (which is the border function of the pattern). This function returns True if the pattern matches and False if it does not. The 'border_funtion' function is used to computes the lengths of the longest proper prefix pf the pattern that is also a suffix, which helps in optimizing the search process by skipping unnecessary comparisons.

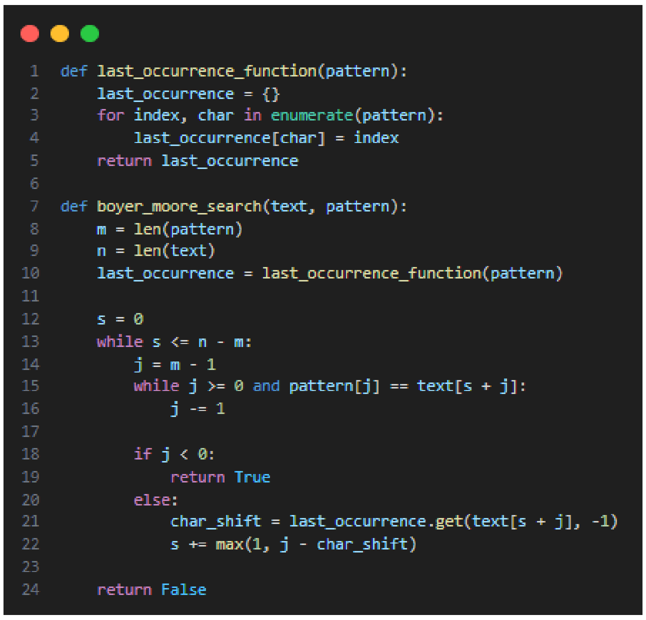


```python
def border_function(pattern):
    table = [0] * len(pattern)
    j = 0
    for i in range(1, len(pattern)):
        if pattern[i] == pattern[j]:
            j += 1
            table[i] = j
        else:
            j = table[j - 1] if j > 0 else 0
    return table

def kmp_search(text, pattern, table):
    m, i = 0, 0
    while m + i < len(text):
        if pattern[i] == text[m + i]:
            if i == len(pattern) - 1:
                return True
            i += 1
        else:
            if table[i] > 0:
                m += i - table[i]
                i = table[i]
            else:
                i = 0
                m += 1
    return False
```

Figure 10. KMP Algorithm Implementation
Source: Author's personal documentation

B. *Boyer-Moore (BM) Algorithm Implementation*

The Boyer-Moore search algorithm implemented in the provided code aims to find a pattern within a text efficiently. The 'last_occurance_function' creates a dictionary mapping each character in the pattern to its last index position. The 'boyer_moore_search' function then uses this dictionary to search for the pattern within the text. Starting from the end of the pattern, it compares characters with the text, shifting the pattern efficiently upon a mismatch. If a match is found, it returns `True`; otherwise, it continues to shift and search. If no match is found after traversing the text, it returns `False`. This approach reduces unnecessary comparisons, improving search performance compared to naive algorithms.

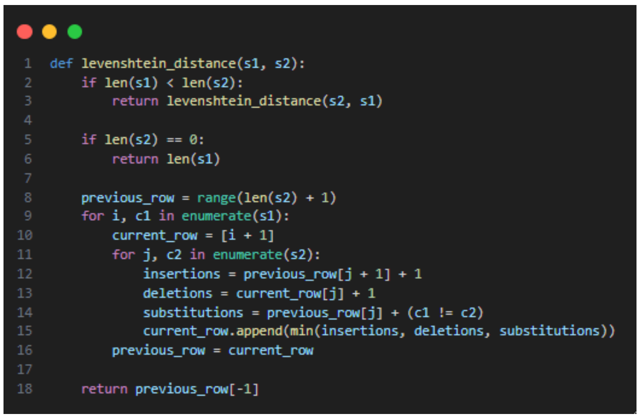


```python
def last_occurrence_function(pattern):
    last_occurrence = {}
    for index, char in enumerate(pattern):
        last_occurrence[char] = index
    return last_occurrence

def boyer_moore_search(text, pattern):
    m = len(pattern)
    n = len(text)
    last_occurrence = last_occurrence_function(pattern)

    s = 0
    while s <= n - m:
        j = m - 1
        while j >= 0 and pattern[j] == text[s + j]:
            j -= 1

        if j < 0:
            return True
        else:
            char_shift = last_occurrence.get(text[s + j], -1)
            s += max(1, j - char_shift)

    return False
```

Figure 11. BM Algorithm Implementation
Source: Author's personal information

C. *Levensteince Distance*

The provided code calculates the Levenshtein distance, which measures the minimum number of edits (insertions, deletions, or substitutions) needed to transform one string into another. The function Levenshtein distance uses dynamic programming, initializing a previous_row to represent distances for transforming prefixes of `s1` into prefixes of `s2`. It iterates through characters of `s1` and `s2`, computing insertion, deletion, and substitution costs, storing the minimum cost in current_row. After processing, the final value in previous_row is the Levenshtein distance between the strings, efficiently comparing their similarity.

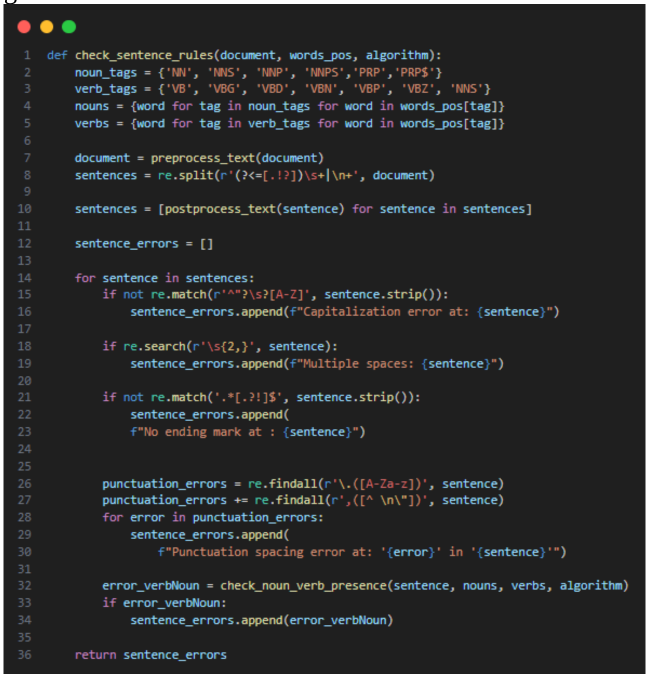


```python
def levenshtein_distance(s1, s2):
    if len(s1) < len(s2):
        return levenshtein_distance(s2, s1)

    if len(s2) == 0:
        return len(s1)

    previous_row = range(len(s2) + 1)
    for i, c1 in enumerate(s1):
        current_row = [i + 1]
        for j, c2 in enumerate(s2):
            insertions = previous_row[j + 1] + 1
            deletions = current_row[j] + 1
            substitutions = previous_row[j] + (c1 != c2)
            current_row.append(min(insertions, deletions, substitutions))
        previous_row = current_row

    return previous_row[-1]
```

Figure 12. Levenshtein Distance Implementation
Source: Author’s personal documentation

D. *Regular Expressions Implementation*

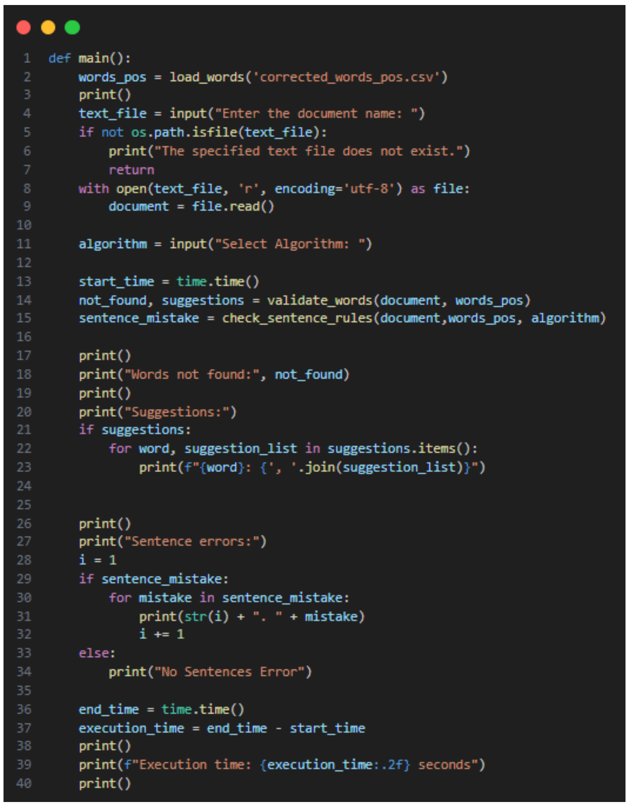
The check_sentence_rules function utilizes regular expressions to validate sentence structure and punctuation within a document. It splits the document into sentences using (?<=[.!?])\s+|\n+ to match end-of-sentence punctuation followed by whitespace or newlines. It checks capitalization at the beginning of sentences with ^"?\s?[A-Z], ensuring sentences start with an uppercase letter. It detects multiple consecutive spaces using \s{2,}, and verifies sentences ending with appropriate punctuation via .*[.?!]$. Additionally, it identifies punctuation spacing errors where periods or commas are not followed by spaces using \.([A-Za-z]) and ,([^ \n\"]). These regex patterns collectively help identify and report common grammatical errors.



```python
def check_sentence_rules(document, words_pos, algorithm):
    noun_tags = {'NN', 'NNS', 'NNP', 'NNPS','PRP','PRP$'}
    verb_tags = {'VB', 'VBG', 'VBD', 'VBN', 'VBP', 'VBZ', 'NNS'}
    nouns = {word for tag in noun_tags for word in words_pos[tag]}
    verbs = {word for tag in verb_tags for word in words_pos[tag]}

    document = preprocess_text(document)
    sentences = re.split(r'(?<=[.!?])\s+|\n+', document)

    sentences = [postprocess_text(sentence) for sentence in sentences]

    sentence_errors = []

    for sentence in sentences:
        if not re.match(r'^"?\s?[A-Z]', sentence.strip()):
            sentence_errors.append(f"Capitalization error at: {sentence}")

        if re.search(r'\s{2,}', sentence):
            sentence_errors.append(f"Multiple spaces: {sentence}")

        if not re.match('.*[.?!]$', sentence.strip()):
            sentence_errors.append(
            f"No ending mark at : {sentence}")


        punctuation_errors = re.findall(r'\.([A-Za-z])', sentence)
        punctuation_errors += re.findall(r',([^ \n\"])', sentence)
        for error in punctuation_errors:
            sentence_errors.append(
                f"Punctuation spacing error at: '{error}' in '{sentence}'")

        error_verbNoun = check_noun_verb_presence(sentence, nouns, verbs, algorithm)
        if error_verbNoun:
            sentence_errors.append(error_verbNoun)

    return sentence_errors
```

Figure 13. Regrex Implementation
Source: Author’s personal documentation

E. *Program Flow*

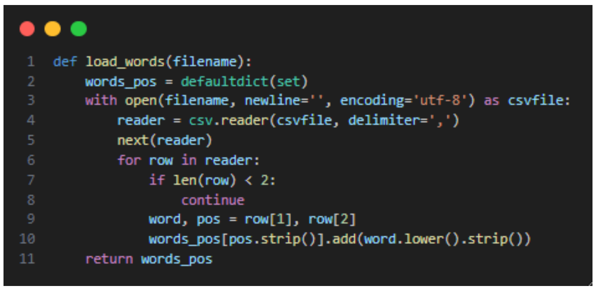


```python
def main():
    words_pos = load_words('corrected_words_pos.csv')
    print()
    text_file = input("Enter the document name: ")
    if not os.path.isfile(text_file):
        print("The specified text file does not exist.")
        return
    with open(text_file, 'r', encoding='utf-8') as file:
        document = file.read()

    algorithm = input("Select Algorithm: ")

    start_time = time.time()
    not_found, suggestions = validate_words(document, words_pos)
    sentence_mistake = check_sentence_rules(document,words_pos, algorithm)

    print()
    print("Words not found:", not_found)
    print()
    print("Suggestions:")
    if suggestions:
        for word, suggestion_list in suggestions.items():
            print(f"{word}: {', '.join(suggestion_list)}")


    print()
    print("Sentence errors:")
    i = 1
    if sentence_mistake:
        for mistake in sentence_mistake:
            print(str(i) + ". " + mistake)
            i += 1
    else:
        print("No Sentences Error")

    end_time = time.time()
    execution_time = end_time - start_time
    print()
    print(f"Execution time: {execution_time:.2f} seconds")
    print()
```

Figure 14. Main Program
Source: author's personal documentation

Here is a detailed program flow for this document checker implementation:

1. Load the Dataset
   The program first will load the dataset that containing words and their parts of speech tags from a CSV file using the load_words function. The data will be stored in a variable called ‘words_pos’, which is a dictionary where each part of speech tag maps to a set of corresponding words.

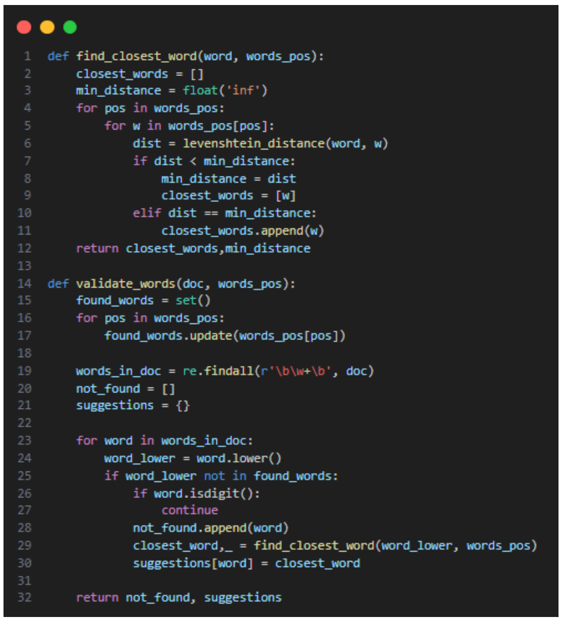


```python
def load_words(filename):
    words_pos = defaultdict(set)
    with open(filename, newline='', encoding='utf-8') as csvfile:
        reader = csv.reader(csvfile, delimiter=',')
        next(reader)
        for row in reader:
            if len(row) < 2:
                continue
            word, pos = row[1], row[2]
            words_pos[pos.strip()].add(word.lower().strip())
    return words_pos
```

Figure 15. load_words funtion
Source: author’s personal documentation

2. Load the Document
   The program will ask the document file name , that user want to check with the algorithm. Then, it will load the document that needs to be checked from a text file and store it in a variable called ‘document’.

3. Validate Words
The program will call the ‘validate_words’ funtion with the document and word_pos as arguments. This function will checks each word in document to see if it is present in the dataset. If not , it is added to the not_found list , and then it will use Levensteince Distance to find the closest word and provides suggestions for corrections.

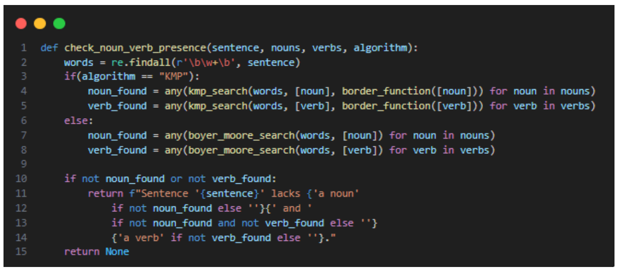


```python
def find_closest_word(word, words_pos):
    closest_words = []
    min_distance = float('inf')
    for pos in words_pos:
        for w in words_pos[pos]:
            dist = levenshtein_distance(word, w)
            if dist < min_distance:
                min_distance = dist
                closest_words = [w]
            elif dist == min_distance:
                closest_words.append(w)
    return closest_words,min_distance

def validate_words(doc, words_pos):
    found_words = set()
    for pos in words_pos:
        found_words.update(words_pos[pos])

    words_in_doc = re.findall(r'\b\w+\b', doc)
    not_found = []
    suggestions = {}

    for word in words_in_doc:
        word_lower = word.lower()
        if word_lower not in found_words:
            if word.isdigit():
                continue
            not_found.append(word)
            closest_word,_ = find_closest_word(word_lower, words_pos)
            suggestions[word] = closest_word

    return not_found, suggestions
```

Figure 16. validate_words function
Source: author’s personal documentation

4. Check Sentences Rules
The program will call ‘check_sentences_rules’ funtion with document and words_pos as arguments. This function will split the document into individual sentensces using regex to handle punctuation and new lines. Then, for every sentence, the function will check the Capitalization at the beginning of the sentence, no multiple space within words, sentences must end with a period, question mark, or exclamation point, proper spacing around punctuation using Regex. This funtion will also check foreach sentence must contain at least one noun and one verb using KMP or Boyer-Moore algorithms.

The implementation of ‘check_sentences_rules’ function can be seen at Figure 13. In that function will call ‘check_noun_verb_presence’ function to check if a sentence contains minimum one noun and verb. In this function, it will using KMP or BM algorithm to match every noun and verbs in dataset to the sentence. If none of the nouns or verbs exist in the sentence, it will return an error message.



```python
def check_noun_verb_presence(sentence, nouns, verbs, algorithm):
    words = re.findall(r'\b\w+\b', sentence)
    if(algorithm == "KMP"):
        noun_found = any(kmp_search(words, [noun], border_function([noun])) for noun in nouns)
        verb_found = any(kmp_search(words, [verb], border_function([verb])) for verb in verbs)
    else:
        noun_found = any(boyer_moore_search(words, [noun]) for noun in nouns)
        verb_found = any(boyer_moore_search(words, [verb]) for verb in verbs)

    if not noun_found or not verb_found:
        return f"Sentence '{sentence}' lacks {'a noun'
            if not noun_found else ''}{' and '
            if not noun_found and not verb_found else ''}
            {'a verb' if not verb_found else ''}."
    return None
```

Figure 17. check_noun_verb_presence function
Source: author’s personal documentation

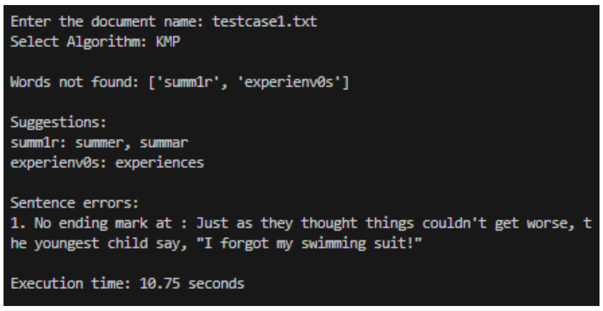
5. Output Results
The program prints the list of words that were not found in the dataset and the suggested corrections. The program will also print a list of sentences that contain several mistakes.

IV. Experiment

Here is some experiment for the program with several test case document that has been prepared:

1. Test Case 1

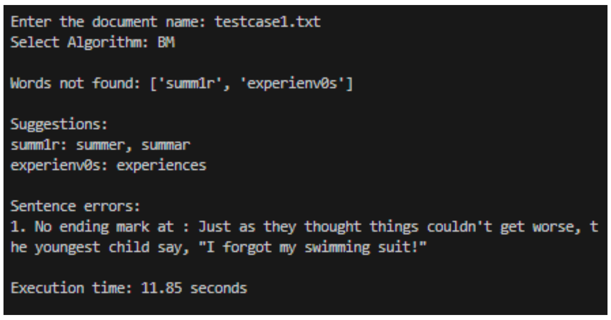
Using KMP Algorithm



```
Enter the document name: testcase1.txt
Select Algorithm: KMP

Words not found: ['summ1r', 'experienv0s']

Suggestions:
summ1r: summer, summar
experienv0s: experiences

Sentence errors:
1. No ending mark at : Just as they thought things couldn't get worse, t
he youngest child say, "I forgot my swimming suit!"

Execution time: 10.75 seconds
```

Figure 18. Test Case 1 with KMP
Source: author’s personal documentation

Using BM Algorithm

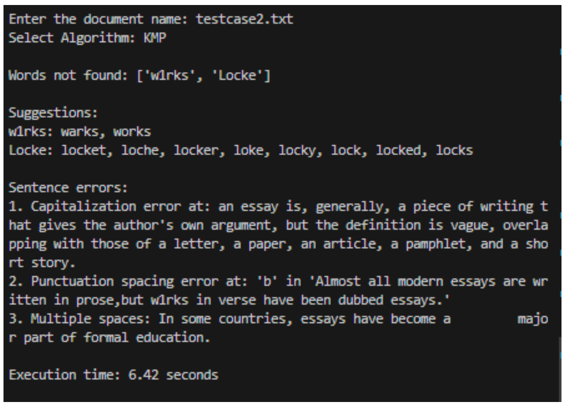


```
Enter the document name: testcase1.txt
Select Algorithm: BM

Words not found: ['summ1r', 'experienv0s']

Suggestions:
summ1r: summer, summar
experienv0s: experiences

Sentence errors:
1. No ending mark at : Just as they thought things couldn't get worse, t
he youngest child say, "I forgot my swimming suit!"

Execution time: 11.85 seconds
```

Figure 19. Test Case 1 with BM
Source: author’s personal documentation

2. Test Case 2

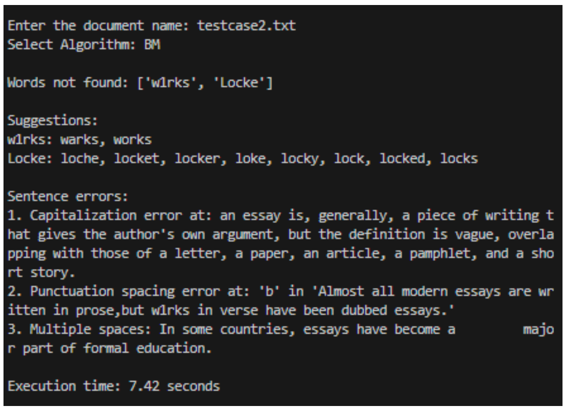
Using KMP Algorithm



```
Enter the document name: testcase2.txt
Select Algorithm: KMP

Words not found: ['w1rks', 'Locke']

Suggestions:
w1rks: warks, works
Locke: locket, loche, locker, loke, locky, lock, locked, locks

Sentence errors:
1. Capitalization error at: an essay is, generally, a piece of writing t
hat gives the author's own argument, but the definition is vague, overla
pping with those of a letter, a paper, an article, a pamphlet, and a sho
rt story.
2. Punctuation spacing error at: 'b' in 'Almost all modern essays are wr
itten in prose,but w1rks in verse have been dubbed essays.'
3. Multiple spaces: In some countries, essays have become a        majo
r part of formal education.

Execution time: 6.42 seconds
```

Figure 20. Test Case 2 with KMP
Source: author's personal documentation

Using BM Algorithm

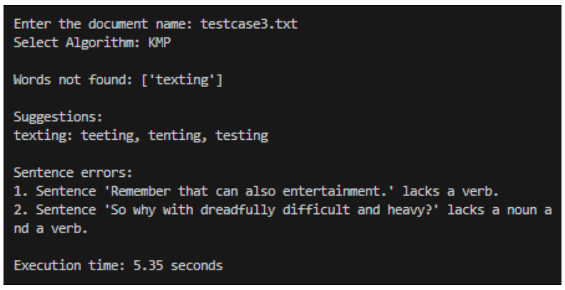


```
Enter the document name: testcase2.txt
Select Algorithm: BM

Words not found: ['w1rks', 'Locke']

Suggestions:
w1rks: warks, works
Locke: loche, locket, locker, loke, locky, lock, locked, locks

Sentence errors:
1. Capitalization error at: an essay is, generally, a piece of writing t
hat gives the author's own argument, but the definition is vague, overla
pping with those of a letter, a paper, an article, a pamphlet, and a sho
rt story.
2. Punctuation spacing error at: 'b' in 'Almost all modern essays are wr
itten in prose,but w1rks in verse have been dubbed essays.'
3. Multiple spaces: In some countries, essays have become a        majo
r part of formal education.

Execution time: 7.42 seconds
```

Figure 21. Test Case 2 with BM
Source: author's personal documentation

3. Test Case 3

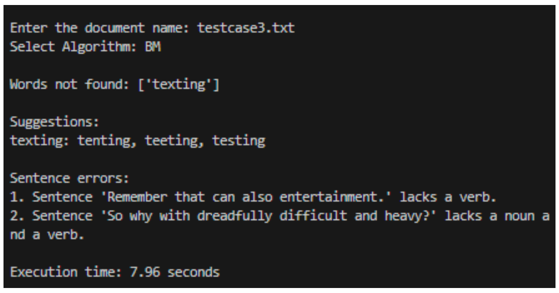
Using KMP Algorithm



```
Enter the document name: testcase3.txt
Select Algorithm: KMP

Words not found: ['texting']

Suggestions:
texting: teeting, tenting, testing

Sentence errors:
1. Sentence 'Remember that can also entertainment.' lacks a verb.
2. Sentence 'So why with dreadfully difficult and heavy?' lacks a noun a
nd a verb.

Execution time: 5.35 seconds
```

Figure 22. Test Case 3 with KMP
Source: author's personal documentation

Using BM Algorithm

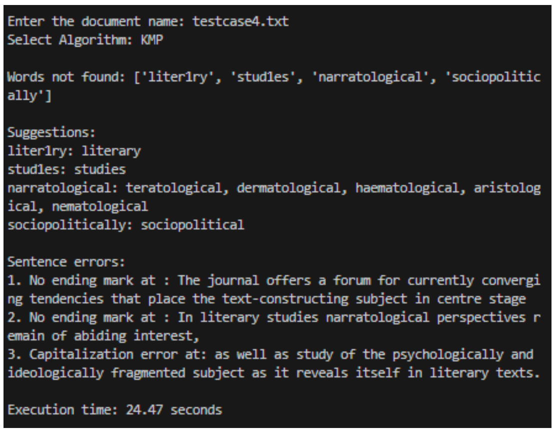


```
Enter the document name: testcase3.txt
Select Algorithm: BM

Words not found: ['texting']

Suggestions:
texting: tenting, teeting, testing

Sentence errors:
1. Sentence 'Remember that can also entertainment.' lacks a verb.
2. Sentence 'So why with dreadfully difficult and heavy?' lacks a noun a
nd a verb.

Execution time: 7.96 seconds
```

Figure 23. Test Case 3 with BM
Source: author's personal documentation

4. Test Case 4

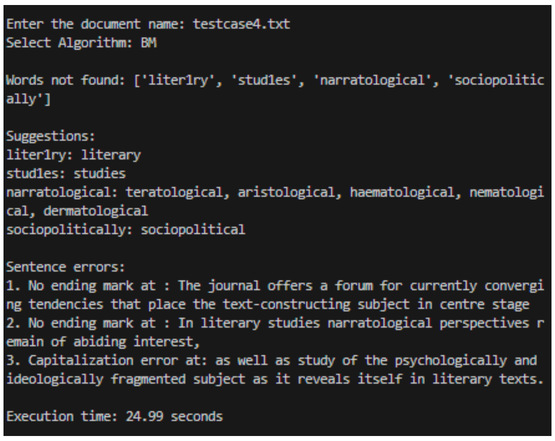
Using KMP Algorithm



```
Enter the document name: testcase4.txt
Select Algorithm: KMP

Words not found: ['liter1ry', 'stud1es', 'narratological', 'sociopolitic
ally']

Suggestions:
liter1ry: literary
stud1es: studies
narratological: teratological, dermatological, haematological, aristolog
ical, nematological
sociopolitically: sociopolitical

Sentence errors:
1. No ending mark at : The journal offers a forum for currently convergi
ng tendencies that place the text-constructing subject in centre stage
2. No ending mark at : In literary studies narratological perspectives r
emain of abiding interest,
3. Capitalization error at: as well as study of the psychologically and
ideologically fragmented subject as it reveals itself in literary texts.

Execution time: 24.47 seconds
```

Figure 24. Test Case 4 with KMP
Source: author's personal documentation

Using BM Algorithm



```
Enter the document name: testcase4.txt
Select Algorithm: BM

Words not found: ['liter1ry', 'stud1es', 'narratological', 'sociopolitic
ally']

Suggestions:
liter1ry: literary
stud1es: studies
narratological: teratological, aristological, haematological, nematologi
cal, dermatological
sociopolitically: sociopolitical

Sentence errors:
1. No ending mark at : The journal offers a forum for currently convergi
ng tendencies that place the text-constructing subject in centre stage
2. No ending mark at : In literary studies narratological perspectives r
emain of abiding interest,
3. Capitalization error at: as well as study of the psychologically and
ideologically fragmented subject as it reveals itself in literary texts.

Execution time: 24.99 seconds
```

Figure 25. Test Case 4 with KMP
Source: author's personal documentation

V. ANALYSIS

Based on extensive testing with various test cases, the program demonstrates the effective application of string matching algorithms for document checker. The program successfully identifies all errors present in the test cases and provides accurate and relevant suggestions for misspelled words. This enhances the overall functionality of the document checker by ensuring that misspelled words are not only detected but also corrected with appropriate recommendations. The program performs well in checking sentence structures, meeting the expectations for basic grammatical rules. It correctly identifies sentences that lack either a noun or a verb. However, this approach has limitations since it does not consider the proper sequence of nouns and verbs within a sentence. A sentence might contain both a noun and a verb, but the order could be incorrect, which the program does not currently address. Despite this, the program effectively ensures that each sentence starts with a capital letter and ends with appropriate punctuation marks such as periods, question marks, or exclamation points. Additionally, it detects extra spaces within sentences, ensuring proper spacing and formatting. All these aspects were correctly identified in the conducted test cases, indicating the program's robustness in handling various sentence rules.

For the execution time, it can be observed that the program takes more time when using the BM algorithm compared to the KMP algorithm. This is primarily due to the complexity of preprocessing BM, which involves creating the last occurrence table. Additionally, because the pattern length is short, the advantages of BM are not fully realized.

VI. CONCLUSION

Based on the theory and experiments conducted, it can be concluded that the application of string matching algorithms like KMP, BM, regex, and Levenshtein distance can effectively check the correctness of a document. Sentence structures can also be verified using string matching and regex for elements such as capitalization, ending punctuation marks, and proper spacing. The most efficient algorithm for matching in this application depends on the pattern size. Since the pattern size corresponds to words in the dictionary, it is concluded that the KMP algorithm is more suitable than BM for this application due to the shorter pattern length. This application can help users check their documents more effectively, eliminating the need for manual verification.

However, this application of string matching and regex for document checking has limitations. It does not verify the grammatical correctness of sentences or their proper order, such as ensuring Subject + Verb structure, or checking for correct use of plural and singular forms. Therefore, the current program can be further optimized and upgraded to include grammar checking, making it even more beneficial for the community.

VIDEO LINK AT YOUTUBE

For more detail explanation, video can be seen at https://youtu.be/qCGdBVRAom8?si=tZCtwoY6DPeGCu6

.

APPENDIX

The complete program of this document checker implementation can be found below:

https://github.com/Benardo07/document_checker

ACKNOWLEDGMENT

First and foremost, I would like to express my gratitude to God for providing me with the strength and perseverance to complete this paper. Additionally, I would like to extend my sincere thanks to my friends who have continuously supported me throughout this journey. I am also deeply grateful to the professors of the Algorithm Strategy class, Mr. Rinaldi Munir, Mr. Rila Mandala, and Mrs. Nur Ulfa, for their invaluable guidance and for providing essential materials related to algorithm strategy.

REFERENCES

[1] Munir, Rinaldi. “Pencocokan String (String/Pattern Matching).”, https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf. Accessed 10 June 2024.

[2] Munir, Rinaldi. "String Matching dengan Regex.", https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/String-Matching-dengan-Regex-2019.pdf. Accessed 10 June 2024.

[3] Supardi. "Analisis Penerapan Algoritma String Matcing pada Aplikasi Pencarian Berkas di Komputer.", https://repository.uinjkt.ac.id/dspace/bitstream/123456789/15152/1/SUPARDI-FST.pdf. Accessed 10 June 2024.

[4] Nam, Ethan. "Understanding the Levenshtein Distance Equation for Beginners.", https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0. Accessed 11 June 2024.

[5] Chua, E.H. "How to Use Regular Expressions.", https://www3.ntu.edu.sg/home/ehchua/programming/howto/Regexe.html. Accessed 11 June 2024.

STATEMENT

I hereby declare that the paper I wrote is my own work, not an adaptation, or a translation of someone else's paper, and it is not plagiarism.

Bandung, 12 Juni 2024

[illegible]

Benardo 13522055