Kingdom of Saudi Arabia
Ministry of Education
Umm Al-Qura University

المملكة العربية السعودية
وزارة التعليم
جامعة أم القرى

جامعة أم القرى
UMM AL-QURA UNIVERSITY

College of Computer Science and Information System

Computer Science

Algorithms1401240-4

String Matching Algorithms

Prepared By:

Lina Khalid Alrefi	ID: 442016671
Shahd Alzubair Alsaleh	ID: 441017891
Ameera Fahad Alfadli	ID: 442000607
Ghadi Jameel Alkehily	ID: 441004352

supervision by:

Dr.Areej Othman Alsini

Contents

- Introduction 3
- Naïve string matching Algorithm 4
    - Data Frame implementation 5
    - Stack implementation 6
    - Dictionary implementation 7
- KMP Algorithm 8
    - List Implementation 10
- Edit Distance Algorithm 11
    - Data Fame Implementation 13
    - Dictionary Implementation 14
    - Queue Implementation 14
- The best algorithm for string matching 16
- The best performance against naïve algorithm 16
- The device 17

Introduction

In computer science, string-matching algorithms are an important class of string algorithms ,the problem of string matching is trying to find a place where one or several strings (also called patterns) are found within a larger string or text .They are used in many real word applications like queries in Database schema, Network systems, searching for a specific word or phrase in a large document.

String Matching Algorithms can broadly be classified into two types of algorithms, the first type called **Exact String Matching Algorithms**, they are used to find one, several, or all occurrences of a defined string (pattern) in a large string (text or sequences) such that each matching is perfect. All alphabets of patterns must be matched to corresponding matched subsequence, there are a lot of algorithms their implementation based on this type such as Naïve Algorithm, KMP (Knuth Morris Pratt) Algorithm and Boyer Moore Algorithm.

The second type is **Approximate String Matching Algorithms;** they are used to find approximate matches for a given string within a larger set of strings. These algorithms are used in many applications such as spell-checking, natural language processing, and information retrieval. The examples of approximate string matching algorithms include the Edit distance algorithm, the Jaro-Winkler distance algorithm, and the Damerau-Levenshtein distance algorithm.

So in this report we will focus in Naïve Algorithm, KMP (Knuth Morris Pratt) Algorithm and edit distance algorithm in fields of their implementation and the best data structure we can use to implement them based on time complexity and at the end we will decide which algorithm is much more efficient in its performance against naïve algorithm.

Naïve string matching Algorithm

Naive string matching is an algorithm for finding a given pattern in a text. It is compares the pattern to every substring of the text in order to find a match.

Naive string matching is a simple and straightforward algorithm used to search for a pattern within a larger text. The algorithm compares each character in the pattern with the corresponding character in the text, starting from the first character, and moving char by char. If all characters in the pattern match the corresponding characters in the text, the pattern is said to be found in the text. If a mismatch occurs, the algorithm continues with the next character in the text and repeats the process until either the pattern is found or the end of the text is reached

The naive string matching algorithm is called "naive" because it has a high time complexity and is not very efficient for large texts.

Also called A brute-force search. It is a type of algorithm that systematically checks all possible solutions to a problem until the correct solution is found.

It is an exhaustive search that is used when no other more efficient algorithms are available.

Brute-force searches are often used in cryptography and password cracking, as they can be used to try every possible combination of characters until the correct one is found.

Algorithm implementation

- Data Frame
- Stack
- Dictionary

I. Data Frame implementation

In this implementation we store our data in data frame which is a two-dimensional data structure in Python's Pandas library that represents a table of data with rows and columns, The second step pass it to method for string matching in this way we send text(data in data frame) and pattern to text matching.

The function takes two arguments: text string and pattern string.

The length of the text and pattern strings are calculated and stored in the variables n and m, respectively, the length of pattern should be equal or less than length of text.

Within the loop, the variable j is initialized to 0 and is used to keep track of the number of characters that match in the pattern and text.

At every iteration if the condition satisfied (compares each character in the pattern to the corresponding character in the text) increment j by one

So, when reach to length of pattern equals to j, here we can know there is matching and return index of column I, which considered as first column from which the matching started, if we need to now the number of row only print variable row its place inside the for loop which Iterates through the joined row column of the Data Frame.

If no match is found, the loop will complete and the function will return -1, indicating that the pattern was not found in the text.

Time Complexity

In the worst-case scenario, the algorithm will have to compare the target string to each string in the data frame, which would result in a time complexity of O(n ^2) where n is the number of rows in the data .

The result of measuring the execution time of a code Using time.perf_counter() function is 0.0036039999999957217 ms

II. Stack implementation

In this implementation we store our data in stack which implemented by array, This algorithm try match a pattern in a given text. The algorithm uses a stack to keep track of the indices in the text that match the characters in the pattern, The algorithm starts with two pointers i and j pointing to the first character of the text and pattern, respectively. The algorithm then compares the characters at these positions. If the characters are equal, both pointers are incremented. If the characters are not equal, the i pointer is incremented and the j pointer is reset to 0 to by this step we shift the pattern to next index According to text.

The algorithm continues this process until either a match is found (j) pointer reaches the end of the pattern) or the end of the text is reached(i) pointer reaches the end of the text). If a match is found, the function returns the index of the match, otherwise it returns -1.

The code then uses this function to find the first occurrence of the pattern in the "joined_row" column of the data frame mat.df. If a match is found, it stores the index of the row in the result variable and breaks out of the loop. Finally, it prints the result indicating whether a match was found or not and the index of the match.

Time Complexity

The time complexity of this algorithm is O(n*m), where n is the length of the text string and m is the length of the pattern string.

This is because in the worst-case scenario, the algorithm needs to compare each character of the text string with the pattern string. This takes m comparisons for each character in the text string, making the total time complexity O(n*m).

The space complexity is O(m) because the maximum size of the stack is m.

The result of measuring the execution time of a code Using time.perf_counter() function is 0.0071154999999761 ms

III. Dictionary implementation

In this implementation we store our data in dictionary which is a built-in data structure that stores key-value pairs. It's unlike the list it is unordered, mutable, and associative collection of items, where each item is a pair composed of a key and a value.

dictionaries are used to store key-value pairs, where the keys are used to identify the values unlike the list which typically used to store collections of items of the same type.

- The algorithm takes two inputs: a text string and a pattern string. The function uses a hash table to store the index of each character in the pattern.
- The algorithm uses a while loop to search for the pattern in the text. The **i** and **j** variables keep track of the current index in the text and pattern, respectively.
- If a character in the text matches the corresponding character in the pattern, both **i** and **j** are incremented.
- If a mismatch is found, the **i** index is adjusted by the value stored in the hash table (if the current character in the pattern is in the hash table), otherwise **i** is incremented by 1 and **j** is reset to 0.
- If the pattern is found in the text, the index of the first character in the match is returned. If the pattern is not found, -1 is returned.

Time Complexity

In the worst-case scenario, The time complexity using the dictionary is O(n * m), where n is the length of the text and m is the length of the pattern. This is because for each character in the text, the algorithm compares it with the corresponding character in the pattern and continues to do so until it finds a mismatch or until it has successfully matched all the characters in the pattern. This process is repeated for each character in the text, resulting in a time complexity of O(n * m).

The result of measuring the execution time of a code Using time.perf_counter() function is 0.010458599999992657 ms

The best data structure for naïve algorithm

The time complexity of a naive algorithm implemented by a data frame is O(n^2), while the time complexity of a naive algorithm implemented by a dictionary and stack is O(n). Therefore, the naive algorithm implemented by a dictionary and stack is more efficient than the naive algorithm implemented by a data frame

Pseudocode

```
FUNCTION naive_string_matching:
    n = len(text)
    m = len(pattern)
    for i in range(n - m + 1):
        j = 0
        while j < m and text[i + j] == pattern[j]:
            j += 1
        if j == m:
            return i
    return -1


result = -1
for i from 0 to length of df["joined_row"] - 1
    st = df["joined_row"][i]
    col = naive_string_matching(st, target)
    if col != -1
        result = i
        break
```

KMP Algorithm

The Knuth-Morris-Pratt (KMP) algorithm is a string matching algorithm that performs an efficient search for a pattern within a text, is an improvement over the naive algorithm.
It utilizes a preprocessing step to calculate the longest prefix-suffix values for the pattern, which are stored in a lps[] array. The search step uses this array to efficiently skip over characters in the text that will not contribute to a match, reducing the number of comparisons needed. The algorithm compares characters in the pattern and text one by one, updating its position based on the lps[] array in case of a mismatch. If a complete match is found, the algorithm outputs the index at which the pattern starts in the text.

The KMP (Knuth-Morris-Pratt) algorithm is typically implemented using arrays and loops because they provide a simple and efficient way to implement the algorithms' core logic, which is to pre-compute a partial match table (the "lps" or "longest proper prefix which is also suffix" array) that allows the algorithm to quickly skip characters in the text string when a mismatch occurs, without having to check all the characters one by one. The array and loops make it possible to traverse the text and pattern strings, compare characters, and update the partial match table dynamically.

The KMP algorithm requires low overhead to keep track of its state, which is efficiently done using variables and arrays, and the loop construct allows for repetitive comparisons, making it an ideal structure for implementing the algorithm.

In other hand it's difficult to implement the KMP algorithm using a hash table data structure. The KMP algorithm relies on the use of a prefix function to determine the proper alignment of the pattern and the text being searched. Also, because the prefix function it can be difficult to see how a linked list or stack would be a natural fit for this algorithm.
Perhaps there is a possibility, but it considers a complex task that requires a significant amount of experience and more advanced

List Implementation

Python lists are dynamic, which means they can be updated and resized as needed. They are also changeable, which means that their contents can be modified. Indexing allows you to access individual elements in a list and conduct operations such as adding and removing elements from the list.
The use of lists in Python is a good choice for implementing the KMP algorithm for string matching, as it provides a good balance between readability, flexibility, and performance. However, the specific trade-offs between these factors may depend on the specific requirements of the problem and the limitations of the environment.

The function "KMPSearch" takes two arguments, "pat" and "txt", which are the pattern string and the text string respectively.

I. the length of the pattern string and the text string are calculated and stored in the variables M and N respectively. A list, "lps", is created to hold the LPS values of the pattern. The variable "j" is used as an index for the pattern and the variable "i" is used as an index for the text string.

II. calculates the longest prefix suffix (LPS) array for the pattern string using the "computeLPSArray" function. The LPS array is used to optimize the search for the pattern in the text string.

III. The search loop runs as long as the remaining length of the text string is greater than or equal to the remaining length of the pattern string. Inside the loop, if the current character in the pattern matches the current character in the text string, both the "j" and "i" variables are incremented. If "j" becomes equal to M, it means that the pattern has been found in the text string, and the index of the pattern in the text string is printed. If there is a mismatch, the value of "j" is updated according to the value stored in the "lps" array, and the value of "i" is incremented if necessary.

The computeLPSArray function calculates the LPS array for the pattern string.

I. The length of the previous longest prefix suffix is stored in the "len" variable. The first value in the LPS array is always 0.

II. The loop calculates the remaining values for "i" ranging from 1 to M-1. If the current character in the pattern matches the character at the current value of "len", both "len" and the current value of the LPS array are incremented, and "i" is incremented.

III. If there is a mismatch, "len" is updated according to the value stored in the "lps" array and "i" is incremented if necessary

Time complexity

In the worst-case scenario ,the time complexity of the KMP algorithm is O(n + m), where n is the length of the text and m is the length of the pattern.
The result of measuring the execution time of a code Using time.perf_counter() function is 0.00977209 which is consider very efficient time for string matching.

Pseudocode

```
FUNCTION KMPSearch:

    Set M to the length of the pattern
    Set N to the length of the text
    Initialize lps list of length M with all values set to 0
    Set j to 0
    Call computeLPSArray with pattern, M, and lps as arguments
    Set i to 0
    While N-i is greater than or equal to M-j:
        If pattern at j matches text at i:
            Increment i
            Increment j
        If j is equal to M:
            Print "Found pattern at index" + index i-j
            Set j to the value at lps[j-1]
        If i is less than N and pattern at j does not match text at i:
            If j is not equal to 0:
                Set j to the value at lps[j-1]
            If j is equal to 0:
                Increment i
```

```
FUNCTION computeLPSArray:

    Set len to 0
    Set the first value of lps to 0
    Set i to 1
    While i is less than M:
        If pattern at i matches pattern at len:
            Increment len
            Set lps at i to len
            Increment i
        If len is not equal to 0:
            Set len to the value at lps[len-1]
        If len is equal to 0:
            Set lps at i to 0
            Increment i
```

Edit Distance Algorithm

The edit distance algorithm is a method of calculating the minimum number of edits (insertions, deletions, and substitutions) required to transform one string into another. It is commonly used in natural language processing and computational biology for measuring the similarity between two strings. The algorithm works by comparing each character in the two strings and determining the cost of transforming one character into another. The total cost is then calculated by summing up all the individual costs.

Algorithm implementation

- Data frame.
- Dictionary.
- Queue.

I. Data Fame Implementation

This implementation of edit distance uses Dynamic Programming to calculate the minimum number of edits required to transform one string into another. It creates a data frame with size (len(str1)+1)*(len(str2)+1) and fills the first row and column with 0's. The data frame is then filled with edit distance values by comparing each character in the two strings and assigning a cost of 0 if they are equal, or 1 if they are not. The minimum of the three values (the value from the cell above, the value from the cell to the left, and the value from the diagonal cell) is then assigned to each cell in the data frame. Finally, it returns this data frame which contains all of the edit distance values.

Time Complexity

In the worst-case scenario, the algorithm will have to compare the target string to each string in the data frame, which would result in a time complexity of O(m*n) where m is the length of the first string and n is the length of the second string.

The result of measuring the execution time of a code Using time.perf_counter() function is 4.999999987376214e-07 ms

II. Dictionary Implementation

This implementation of the edit distance algorithm uses dynamic programming to find the minimum number of edits (insertions, deletions, and substitutions) needed to transform one string into another. It creates a 2D array (dp) with m+1 rows and n+1 columns, where m and n are the lengths of the two strings. The first row and column are initialized with 0s, while the rest of the cells are filled in by comparing each character in s1 to each character in s2. If they match, then the value is taken from the cell above and to the left. Otherwise, 1 is added to the minimum of three adjacent cells (above, left, or above-left). The final value in dp[m][n] is then returned as the edit distance between s1 and s2.

The result of measuring the execution time of a code Using time.perf_counter() function is 0.017001100000015867 ms

Time Complexity

In the worst-case scenario, the algorithm will have to compare the target string to each string in the data frame, which would result in a time complexity O(n^2) where n is the length of the longer string

III. Queue Implementation

This implementation of the edit distance algorithm uses dynamic programming to calculate the minimum number of operations (insert, remove, replace) needed to transform one string into another. It creates a 2D array (dp) with the dimensions of the two strings and then iterates through each element in the array. If either string is empty, it assigns the length of the other string to that element. If both strings have characters at that index, it compares them and if they are equal, assigns the value from the previous element. Otherwise, it puts all three operations into a queue and takes out the one with minimum value and adds 1 to it before assigning it to that element in dp. Finally, it returns the value from last element in dp which is the edit distance between two strings.

Time Complexity

In the worst-case scenario, the algorithm will have to compare the target string to each string in the data frame, which would result in a time complexity O(n^2) where n is the length of the longer string

The result of measuring the execution time of a code Using time.perf_counter() function is 6.000000212225132e-07 ms

The best data structure for edit distance algorithm

The time complexity of the edit distance algorithm implemented by data frame is O(mn), where m and n are the lengths of the two strings being compared. The time complexity of the edit distance algorithm implemented by dictionary and queue is O(n^2), where n is the length of the longer string. Therefore, the data frame implementation has a better time complexity than the dictionary and queue implementation.

Pseudocode

```
FUNCTION editDistDP(str1,str2):
  df = pd.DataFrame(columns = range(len(str2) + 1),
  index = range(len(str1) + 1))
  for i in range(len(df.columns)):
        df.iloc[0][i] = i
  for j in range (len(df.index)):
    df.iloc[j][0] = j
  for i in range (1, len (df.index)) :
        for j in range (1, len (df.columns))
             if str1[i-1] == str2[j-1]
                 cost = 0
             else
                 cost = 1
      df.iloc[i][j] = minimum((df.iloc[i-1][j] + 1), (df.iloc[i][j-1] + 1),
      (df.iloc[i-1][j-1] + cost))
 return df
```

The reasons of using these data structures in string matching algorithms

To run the code effectively, the string matching algorithms require assistance from indexing therefore we use the data frame ,list and dictionary Additionally, using the stack and queue to compare the time periods using FIFO and LIFO

The best algorithm for string matching

The best algorithm for string matching depends on the specific application. Generally, the KMP (Knuth-Morris-Pratt) algorithm is considered to be the most efficient for string matching, as it has a time complexity of O(n). The edit distance algorithm and naive algorithm are both slower than KMP, with time complexities of O(mn) and O(n^2), respectively.

The best performance against naïve algorithm

After having reviewed of this algorithms and observation their implementation in different data structures and their time complexity, we now can decide which algorithm has more efficient performance against Naïve algorithm.

The KMP algorithm is much more efficient than the Naïve algorithm. The KMP algorithm uses a pre-processing step to create a partial match table which allows it to skip over sections of the text that have already been checked, while the Naïve algorithm checks each character in the text one by one. This makes the KMP algorithm much faster and more efficient than the Naïve algorithm, as it can quickly identify patterns in large amounts of text.

Edit distance algorithm is more efficient than the Naïve algorithm. Edit distance algorithm uses dynamic programming to find the minimum number of operations required to transform one string into another. This makes it more efficient than the Naïve algorithm which simply compares each character of one string with each character of the other string and counts the number of mismatches. The edit distance algorithm is also able to take into account transpositions, insertions, and deletions which makes it more accurate than the naïve algorithm.

The Device

the device Asus ZenBook 14

Windows 11 Ryzen 5800H-7 Processor 16 GB RAM

512 GB SSD

We read 1740 rows of data, which are divided into 11 columns.

due to half of the hard drive's capacity being utilized, we did not experience any capacity-related issues, and delays were avoided.For the CPU, it has 6 cores, making it quick to execute simultaneous computations.

References:

- https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/
- https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
- https://www.geeksforgeeks.org/edit-distance-dp-5/