National Institute Of Technology Surathkal Mangalore Karnataka-575025

Department Of Information Technology

Lab Assignment:- 06

Name:- Chikkeri Chinmaya

Roll Number:- 211IT017

Branch:- Information Technology (B.Tech)

Section :- S13

Course:- DATA STRUCTURE AND ALGORITHM II (IT251)

Submitted To:-

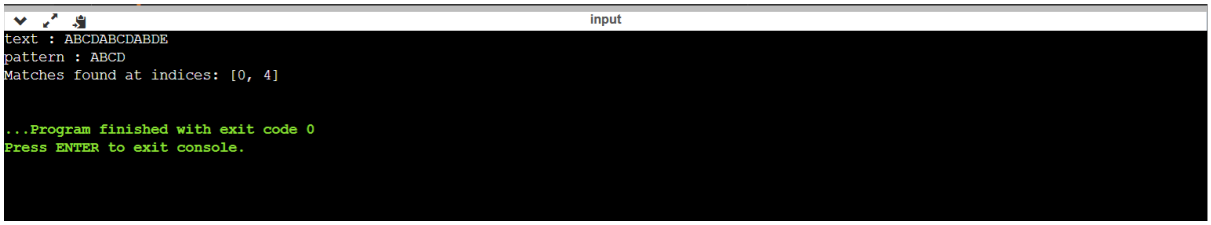
HARIDAS PAI SIR

BOYER_MOORE.PY

```python
def bad_character_heuristic(pattern):
    bad_char = {}
    for i in range(len(pattern)):
        bad_char[pattern[i]] = i
    return bad_char

def good_suffix_heuristic(pattern):
    m = len(pattern)
    suffixes = [0] * m
    border = [0] * m
    j = m - 1

    for i in range(m-2, -1, -1):
        if i > j and suffixes[i + m - 1 - j] < i - j:
            suffixes[i] = suffixes[i + m - 1 - j]
        else:
            if i < j:
                j = i
            while j >= 0 and pattern[j] == pattern[j + m - 1 - i]:
                j -= 1
            suffixes[i] = i - j

    for i in range(m):
        border[i] = m - suffixes[i]

    return border

def boyer_moore(text, pattern):
    n = len(text)
    m = len(pattern)
    result = []

    bad_char = bad_character_heuristic(pattern)
    border = good_suffix_heuristic(pattern)

    i = 0
    while i <= n - m:
        j = m - 1

        while j >= 0 and pattern[j] == text[i + j]:
            j -= 1

        if j < 0:
            result.append(i)
```

```
            i += border[0]
        else:
            bad_char_shift = bad_char.get(text[i + j], -1)
            good_suffix_shift = border[j]
            i += max(bad_char_shift, good_suffix_shift)

    return result

# Example usage:
text = "ABCDABCDABDE"
pattern = "ABCD"
print("text : "+text)
print("pattern : "+pattern)
matches = boyer_moore(text, pattern)
print("Matches found at indices:", matches)
```

OUTPUT:-



1. Bad Character Heuristic: The Bad Character Heuristic aims to skip as many characters as possible in the text by examining the mismatches between the pattern and the text during the search process. It creates a lookup table that stores the rightmost occurrence of each character in the pattern.

In the code provided, the function bad_character_heuristic implements the Bad Character Heuristic. It iterates over each character in the pattern and stores its index in the bad_char dictionary, where the character itself is the key and the index is the value.

2. Good Suffix Heuristic: The Good Suffix Heuristic leverages the information about the matching suffixes of the pattern to efficiently shift the pattern during the search. It calculates two arrays: suffixes and border.
    - The suffixes array determines the length of the longest suffix of the pattern that matches a suffix of the shifted pattern. It is computed using a backward scan of the pattern.
    - The border array represents the length of the shortest substring of the shifted pattern that is not a suffix of the pattern. It is derived from the suffixes array.

The function good_suffix_heuristic in the provided code implements the Good Suffix Heuristic. It initializes the suffixes and border arrays by scanning the pattern from right to left, comparing characters and updating the arrays accordingly.

The main function boyer_moore utilizes both heuristics to perform the Boyer-Moore string search. It iterates through the text, comparing characters from right to left with the pattern. If a mismatch occurs, it calculates the shifts based on the Bad Character Heuristic and the Good Suffix Heuristic, choosing the maximum value between them to determine the next position to start the comparison

Time Analysis of Boyer-Moore Algorithm: The time complexity of the Boyer-Moore algorithm for string matching is generally considered to be O(n/m), where n is the length of the text and m is the length of the pattern. However, in the worst case, the algorithm can have a time complexity of O(n * m). Let's discuss the factors that affect the performance and analyze its strengths and weaknesses:

Strengths of Boyer-Moore Algorithm:

1. Efficient in practice: The Boyer-Moore algorithm is known for its practical efficiency in many scenarios. It performs particularly well when the pattern has a small alphabet size or when there are a lot of mismatches between the pattern and the text.
2. Sublinear complexity: In many cases, Boyer-Moore outperforms other string matching algorithms, such as naive or Knuth-Morris-Pratt (KMP), due to its sublinear time complexity. It achieves this by skipping a larger number of characters in the text based on the heuristic rules.
3. Heuristic optimizations: The Bad Character Heuristic and the Good Suffix Heuristic employed by Boyer-Moore provide significant performance improvements. These heuristics enable the algorithm to skip characters efficiently and eliminate unnecessary comparisons, reducing the overall search time.

Weaknesses of Boyer-Moore Algorithm:

1. Worst-case complexity: In the worst-case scenario where there are many occurrences of the pattern in the text, the Boyer-Moore algorithm may exhibit a time complexity of O(n * m). This occurs when there are frequent matches of the pattern's suffixes with the shifted pattern, resulting in minimal skips.
2. Preprocessing overhead: The Boyer-Moore algorithm requires preprocessing to build the lookup tables for the heuristics. The Bad Character Heuristic requires scanning the pattern to create the lookup table, which takes O(m) time. Although this overhead is usually negligible compared to the overall search time, it becomes more noticeable for short patterns or in situations where the pattern changes frequently.
3. Not suitable for small patterns: Boyer-Moore is generally more effective when the pattern length is larger compared to the text length. For very short patterns, other algorithms like the naive algorithm or KMP may perform better due to their lower constant factors and simpler implementations.