University of Connecticut

Rabin-Karp String Search Algorithm

Team 11: Luke Pepin, Caleb Heydon, Jordan Howard, Mitchell Worthington
CSE 3500 - Algorithms and Complexity
Professor Lina Kloub
December 1, 2023

Abstract

This report investigates the Rabin-Karp String Search Algorithm, an innovative approach to pattern matching that employs hashing for efficient substring identification. Utilizing a rolling hash function, the algorithm compares hash values to detect pattern occurrences with average case linear time complexity.

A comparative analysis with alternative string search algorithms, such as Knuth-Morris-Pratt, Boyer-Moore, and Brute-Force, establishes Rabin-Karp's superiority. The report delves into the algorithm's intricacies, implementation details, and performance evaluations, emphasizing its applicability in scenarios requiring rapid pattern retrieval.

Results demonstrate the algorithm's accuracy and efficiency, while a comprehensive analysis of time complexity underscores its favorable positioning. In summary, the Rabin-Karp algorithm's unique hashing strategy and versatility make it a potent solution for diverse pattern-matching applications.

I. Introduction

The Rabin-Karp String Search Algorithm is a powerful and versatile method for pattern matching. It works by employing a rolling hash function to quickly compare the hash values of substrings in the text with the hash value of the target pattern. When a hash match is found, the algorithm performs a character-by-character comparison to confirm the match, allowing it to find all occurrences of the pattern in linear time complexity.

This report delves into the intricacies of the algorithm, through comparison with other string search algorithms, justification of its superiority, an in depth explanation of the algorithm, implementation details and an analysis of time complexity. All together this report will provide an in depth review of the Rabin-Karp String Search Algorithm as an efficient tool in string searching and pattern matching.

II. Comparison with Other Algorithms

The Rabin-Karp string search algorithm is a pattern matching algorithm used to find occurrences of a substring (or pattern) within a larger text or string. The following are similarly string search algorithms however their processes differ:

Knuth-Morris-Pratt (KMP) Algorithm: The KMP algorithm efficiently searches for a pattern in a text by preprocessing the pattern to create a partial match table, which is used to skip unnecessary comparisons. It excels in single pattern searches its Average case time complexity is O(n) with a worst case of O(n*m)., n being the length of text and m being the length of pattern.

Boyer-Moore Algorithm: The Boyer-Moore algorithm is another widely used string searching technique. It uses a heuristic approach, focusing on character comparisons from right to left, and can significantly reduce the number of character comparisons in practice. It is the most effective for larger alphabets or texts; its Average case time complexity is O(n) with a worst case of O(n*m).

Brute-Force Algorithm: Brute-force string searching is the simplest method where you compare the pattern to the text character by character. While it's not as efficient as the other algorithms mentioned, it serves as a baseline for performance comparison while also having an easy implementation. Its Average case time complexity is O(n*m)

III. Justification of its Superiority

Compared to the other string search algorithms Rabin-Karp is the only one to make use of hashing to efficiently find patterns within a text. This is important because it allows for quicker identification of patterns by converting substrings into hash values, enabling rapid comparison between the hashes of the pattern and potential matches within the text. This method significantly enhances the speed of pattern matching, especially with larger texts or complex patterns, reducing the computational load compared to linear search algorithms.

Rabin-Karp also excels in multiple pattern searches within a single text compared to the other string search algorithms. Through its use of hashing, the algorithm enables simultaneous comparisons with the text's hashed substrings, enabling efficient and parallel pattern evaluations. This is crucial in applications such as plagiarism detection or DNA sequencing, where numerous patterns need rapid identification within extensive texts or sequences.

And finally, while the time complexity will be explored in greater detail later in the report for comparison its Average case time complexity is O(n+m) with a worst case of O(n*m). This positions it between other string search algorithms in terms of time complexity, yet its superior functionality and flexibility in implementation make it a preferred choice of string search algorithms.

IV. Explanation of the Algorithm

The Rabin-Karp String Search Algorithm is a method to find a pattern within a larger text. Unlike other algorithms with similar purposes it utilizes hashing to more efficiently navigate through the search process. As a result of the utilization of hashing the algorithm is particularly efficient when dealing with large texts or multiple pattern searches within a single text.

The algorithm is comprised of 3 primary sections:

1. Hashing: The algorithm starts by generating a hash value for the pattern being searched as well as for each substring of the text that's the same length as the pattern. Each hash

represents the original data uniquely yet still in a standardized format allowing for rapid identification and comparison of information.

2. Comparison: Once created the hash of the search pattern is then compared to each of the substring hashes. If they match, it suggests a potential match and they move onto the last step.

3. Verification: Finally, to avoid hash collisions (where different substrings produce the same hash value), the algorithm performs a secondary character-by-character verification check for the matching substrings.

4. Rolling Hash: Rabin-Karp uses a rolling hash algorithm to compute in linear time. The idea behind a rolling hash is to compute the hash for a fixed-size window of data, which in this case is the length of the pattern. Then, the algorithm “rolls” the window one position at a time and re-computes the hash value. Upon each iteration, the rolling hash only updates the hash by adding the hash value of the next character and subtracting the hash value of the previous character. By rolling, the hash can avoid re-computing the hash of a character.

In summary, the Rabin-Karp String Search Algorithm’s use of hashing revolutionizes pattern recognition within texts, enabling swift identification and comparison of patterns. Its innovative approach, though requiring a secondary verification step, greatly enhances efficiency, making it a valuable tool for diverse applications demanding rapid pattern retrieval in large texts or multiple pattern searches within a single text.

V. Implementation Details

The Rabin-Karp String Search Algorithm is implemented in Python as the file 'Rabin_Karp.py'. The implementation consists of a primary function called 'rabin_karp_search', which includes two secondary functions: 'calculate_hash' and 'rehash'.

Within the 'rabin_karp_search' function, the initial pattern and text hashes are determined. The text hash is made with the first characters of the array, matching the pattern's length. This calculation is performed using the 'calculate_hash' function. Additionally, an empty 'matches' array is initialized to store all instances of matching hashes.

The primary for loop iterates through the length of the text, excluding the pattern's length, to ensure that the pattern can fit within the comparison window. Inside this loop, there are two if statements: the first compares the pattern hash with the text hash of the current text window. If true, the index is added to the 'matches' array. The second if statement rotates the text hash to the next element using the 'rehash' function.

Upon completion of the loop for all elements of the text, the function indicates whether the 'matches' array is empty and displays its contents.

Rabin_Karp.py

```
'''Rabin-Karp Algorithm: Finds a text pattern in a body of text'''
def rabin_karp_search(text, pattern):

    '''Calculates the hash of the pattern we'd like to find'''
    def calculate_hash(string, length):
        hash_value = 0
        for i in range(length):
            hash_value = (hash_value * 256 + ord(string[i])) % prime
        return hash_value

    '''Function to handle the rolling hash'''
    def rehash(old_hash, old_char, new_char, length):
        new_hash = (old_hash - ord(old_char) * (256**(length-1)) % prime) * 256 +
ord(new_char)
        return new_hash % prime

    prime = 101  # A prime number for the hash function (can be any prime)
    text_length = len(text)
    pattern_length = len(pattern)
    pattern_hash = calculate_hash(pattern, pattern_length) # calculate initial hash of the
pattern
    text_hash = calculate_hash(text, pattern_length)       # calculate initial hash of the
text

    # List to store indices of matches
    matches = []

    # Iterate through the text from start to end of where the pattern could be found
    for i in range(text_length - pattern_length + 1):

        # Block 1
        # Check if the current hash of the pattern and the hash of the current substring match
        # Also compare the substring with the pattern to confirm if its a match
        if pattern_hash == text_hash: # O(1) initial check
            if text[i:i+pattern_length] == pattern: # O(len(pattern)) double check
                matches.append(i)

        # Block 2
        # if there are more characters in the text to process, update text_hash for the next
iteration
        if i < text_length - pattern_length:
            text_hash = rehash(text_hash, text[i], text[i+pattern_length], pattern_length)
```

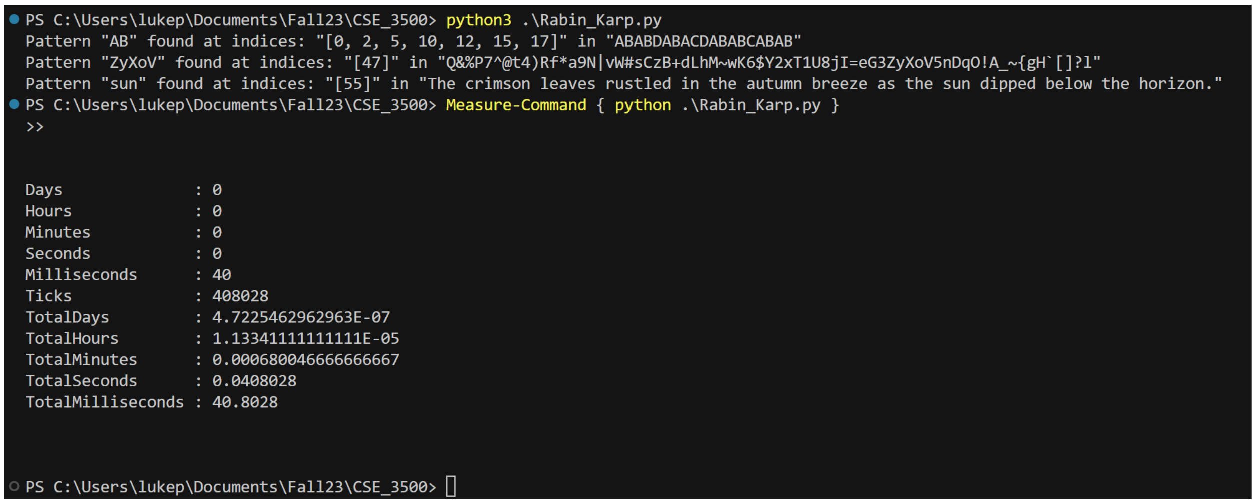
```python
    if len(matches) != 0:
        print(f'Pattern "{pattern}" found at indices: "{matches}" in "{text}"')
    else:
        print(f'Pattern "{pattern}" not found in "{text}"')


# Example usages
text = "ABABDABACDABABCABAB"
pattern = "AB"
rabin_karp_search(text, pattern)

text2 = "Q&%P7^@t4)Rf*a9N|vW#sCzB+dLhM~wK6$Y2xT1U8jI=eG3ZyXoV5nDqO!A_~{gH`[]?1"
pattern2 = "ZyXoV"
rabin_karp_search(text2, pattern2)

text3 = "The crimson leaves rustled in the autumn breeze as the sun dipped below the horizon."
pattern3 = "sun"
rabin_karp_search(text3, pattern3)
```

Additionally, although not part of the algorithm's implementation, the code includes three example usages. These examples serve to vividly demonstrate the functionality of the program in various scenarios.

VI. Results

The following image rk_results.jpg demonstrates two instances of executing the `Rabin_Karp.py` file in the terminal. The first instance showcases its functionality using the typical `python3 ./Rabin_Karp.py` command, while the second utilizes `Measure-Command { python ./Rabin_Karp.py }` to measure the script's execution time.

rk_results.jpg:



```
PS C:\Users\lukep\Documents\Fall23\CSE_3500> python3 .\Rabin_Karp.py
Pattern "AB" found at indices: "[0, 2, 5, 10, 12, 15, 17]" in "ABABDABACDABABCABAB"
Pattern "ZyXoV" found at indices: "[47]" in "Q&%P7^@t4)Rf*a9N|vW#sCzB+dLhM~wK6$Y2xT1U8jI=eG3ZyXoV5nDqO!A_~{gH`[]?1"
Pattern "sun" found at indices: "[55]" in "The crimson leaves rustled in the autumn breeze as the sun dipped below the horizon."
PS C:\Users\lukep\Documents\Fall23\CSE_3500> Measure-Command { python .\Rabin_Karp.py }
>>


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 40
Ticks             : 408028
TotalDays         : 4.7225462962963E-07
TotalHours        : 1.13341111111111E-05
TotalMinutes      : 0.000680046666666667
TotalSeconds      : 0.0408028
TotalMilliseconds : 40.8028



PS C:\Users\lukep\Documents\Fall23\CSE_3500>
```

The initial test run validates the functionality of the function, precisely identifying occurrences of the pattern within the provided text: it successfully detects 7 instances of the pattern "AB," 1 occurrence of "ZyXoV," and another single instance of "sun" within their respective contexts.

The second test serves as a performance evaluation, shedding light on the algorithm's efficiency by measuring its execution time. To further test this concept the following code was made to replace the example usage portion of the code.

Rabin_Karp.py
```
# Table Data
text = ''.join(random.choices(string.ascii_letters + string.digits, k=10000000))
pattern = "ZyXoV"
rabin_karp_search(text, pattern)
```

The table was created by adjusting the value of 'k' to represent different text lengths, using the Measure-Command in the terminal for multiple tests, presenting the subsequent results.

Length of Text	Average Total Milliseconds
10	73.3902
100	77.2534
1,000	77.6421
10,000	84.3241
100,000	120.0382
1,000,000	528.6275
10,000,000	4600.9463

These findings indicate the predicted trend of the algorithm's performance concerning text length, showcasing the varying time taken for pattern identification as the length of the text increases.

VII. Analysis of Time Complexity with Proofs

Firstly, let the length of the text = n, and the length of the pattern = m.

Analysis of Subfunctions:
Consider the calculate_hash function, which is used to calculate the pattern hash and initial hash of the text, where string = initial substring, and length = length of the pattern:

```
    '''Calculates the hash of the pattern we'd like to find'''
    def calculate_hash(string, length):
        hash_value = 0
        for i in range(length):
            hash_value = (hash_value * 256 + ord(string[i])) % prime
        return hash_value
```

Since the calculate_hash function iterates through the substring of length m, adding O(1) arithmetic to hash_value each time for m iterations, calculate_hash takes O(m) time.

Next, consider the rehash function, which is the main function simulating the rolling hash functionality.

```
    '''Function to handle the rolling hash'''
    def rehash(old_hash, old_char, new_char, length):
        new_hash = (old_hash - ord(old_char) * (256**(length-1)) % prime) * 256 +
ord(new_char)
        return new_hash % prime
```

The rehash function takes four inputs, does O(1) arithmetic on all inputs, and calculates a new hash based on the arithmetic. Therefore rehash takes O(1).

Analysis of the algorithm:
Next we will analyze the rest of the rabin-karp algorithm. The function starts by calculating the hashes of the initial substring and pattern, taking O(2m) time. The matches array is also initialized in O(1).

```
    pattern_hash = calculate_hash(pattern, pattern_length) # calculate initial hash pattern
    text_hash = calculate_hash(text, pattern_length)# calculate initial hash text
    matches = []
```

Next, the algorithm iterates over the larger text in a for loop. Block 1 first determines if there is a match between the pattern and current substring. There is first an initial check of hash comparisons in O(1), followed by a second check of a string comparison in O(m). On average, the pattern can only be found a couple of times throughout the entire text, meaning it is more likely that the O(m) double check will not be utilized, meaning that Block 1's iteration time is closer to O(1).

Block 2 calculates the rehash of the next iteration if there is more text to be analyzed. This takes O(1) time as calculated previously.

```
    for i in range(text_length - pattern_length + 1):
        # Block 1
        if pattern_hash == text_hash: # O(1) initial check
            if text[i:i+pattern_length] == pattern: # O(m) double check
                matches.append(i)

        # Block 2
        # if more characters in the text to process, update hash for next iteration
        if i < text_length - pattern_length:
            text_hash = rehash(text_hash, text[i], text[i+pattern_length], pattern_length)
```

Best and Average Case Analysis:
On average, there will only be a few instances of the pattern in the text, meaning that inside the for loop will take on average Θ(1) time. Since the algorithm iterates over the whole text once, the for loop takes Θ(n) time. Lastly, the initial hashes take Θ(2m) time. Therefore the average case time complexity is Θ(n+2m) or Θ(n+m).

Worst Case Analysis:
The worst case occurs when the text is the pattern repeated every time. An example of this is when the pattern is "AA" and the text is "AAAAA." In this case, the O(m) double check is always calculated since the pattern_hash will always equal the text_hash. Therefore, O(m) time to find a match is iterated n times, resulting in a O(n*m) time complexity. It also takes O(2m) to find the initial hashes, however n*m will always grow faster than 2m for big-O, so the resulting worst case time complexity is still O(n*m).

VIII. Conclusion

The Rabin-Karp String Search Algorithm stands out as a versatile and efficient tool for pattern matching, leveraging hashing to achieve quick and accurate results. Its superiority over other string search algorithms is evident in its ability to handle large texts and multiple pattern searches simultaneously, while also being fairly simple to understand. Despite its superiority in larger texts, the only downside to using the Rabin-Karp String Search Algorithm is when the pattern is heavily abundant in the text. The time complexity establishes the algorithm's average case at Θ(n+m) and the worst case at O(n*m), positioning it favorably in comparison to other algorithms.

References

1. GeeksforGeeks. "Rabin-Karp Algorithm for Pattern Searching." *GeeksforGeeks*, https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/.

2. Programiz. "Rabin-Karp Algorithm." *Programiz*, https://www.programiz.com/dsa/rabin-karp-algorithm.

3. GeeksforGeeks. "KMP Algorithm for Pattern Searching." *GeeksforGeeks*, https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/.

4. GeeksforGeeks. "Boyer-Moore Algorithm for Pattern Searching." *GeeksforGeeks*, https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/.

5. freeCodeCamp. "Brute Force Algorithms Explained." *freeCodeCamp*, https://www.freecodecamp.org/news/brute-force-algorithms-explained/.