An Efficient, Hybrid, Double-Hash String-Matching Algorithm

Mehmet Bicer \(^1\) and Xiaowen Zhang \(^1,2\)

\(^1\) Department of Computer Science
Graduate Center, City University of New York
365 Fifth Avenue, New York, NY 10016, U.S.A.

\(^2\) Department of Computer Science
College of Staten Island, City University of New York
2800 Victory Blvd, Staten Island, NY 10314, U.S.A.

Email: mbicer@gradcenter.cuny.edu, xiaowen.zhang@csi.cuny.edu

Abstract—In this paper we show that combining some of the good features of the existing popular algorithms can be even more efficient. This new algorithm is hybrid as it employs features from Boyer-Moore-Horspool, Rabin-Karp and Raita algorithms. We compare the right most character as well as use two independent hash functions and no character by character checking - hence leaving a very small probability for a false positive result if there is any. The proposed algorithm particularly does well when the pattern is very long as it will skip checking character by character comparison.

Keywords: Boyer-Moore-Horspool; Rabin-Karp; Raita; Hash functions; BPHashHalf; BKDRHash

I. INTRODUCTION

Pattern search algorithms takes into account many factors: the nature of the problem, the size of the pattern, size of the alphabet, size of the text, whether to find first occurrence of the pattern or all of the occurrences, whether only a single pattern search or multiple different pattern search, whether exact pattern match or approximate pattern match or high probable pattern match or low probable pattern search (i.e. you do not expect to find the pattern at all), type of the pattern – i.e. whether pattern containing digits only, characters only or hybrid (as in ASCII characters) or binary strings or digits. As one can tell, there are many aspects of pattern matching problem and based on the nature of the problem, there are solutions accordingly.

A. Problem description:
Find all of the occurrences of a pattern within the text.

B. Reasons for the modifications (Rationale):
Weaknesses, vulnerabilities and inefficiencies of the original algorithms and how a new algorithm can solve these problems.

C. Layout of the paper:
Section I is about introduction and problem definition. In Section II we explain three algorithms: Boyer-Moore-Horspool \([1]\) \([2]\), Rabin-Karp \([3]\) and Raita \([4]\) briefly. In Section III we explain the new algorithm with double-hash, its key features, algorithm and the pseudo-code. In Section IV we present experiments and test results, Section V continues with a short complexity analysis and finally Section VI concludes the paper.

II. EXISTING ALGORITHMS

A. Preliminaries
Here we will explain three algorithms. But to help the audience understand the paper easier, we will show a very simple example with brute force – i.e. a very simple text and pattern.

1. Notations
Here are a simple alphabet, a pattern, and text. They are used in the examples below. We assume that there is an alphabet \(\Sigma\) and both pattern \((P)\) and a text \((T)\) are comprised of this alphabet \(\Sigma\).

\(\Sigma = \{A, B, D\}\)

Text \(t[0 \ldots n-1]\) //for size \(n\), e.g. \(t[] = "AABDDABADBADDBBBA"

\(n = 17\) (length of text)

Pattern = \(p[0 \ldots m-1]\) //for size \(m\), e.g. \(p[] = "ABDDDB"

\(m = 5\) (length of pattern)

Input: \(t\) and \(p\)
Output: position of the first character of all occurrences of \(p\) in the text \(t\).

2. Bad Match Table (BMT)
Except Rabin Karp algorithm, all of the following algorithms, including the newly proposed double-hash algorithm, are using a bad match table.
BMT is created from the characters of the pattern. All pattern characters are given an index value starting from 0 to m-2 (the last character on the right is not considered hence if it did not appear previously its skip size is m) and using the following formula the skip value is determined.

Example:
Pattern (P) = “ABDDB”
Index (i): 0 1 2 3 4

1) Set all values in alphabet to max size (i.e. pattern size):
For all i in $\Sigma$ BMT[i] $\Leftarrow$ m
2) Check and find each pattern (p) character in BMT and set its skip value according to following formula:
For all i in pattern: BMT [ p[i] ] $\Leftarrow$ m – i – 1

<table>
<thead>
<tr>
<th>Skip table</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1: Bad Match Table (BMT. a.k.a. skip table for the above example – pattern = “ABDDB”)

‘*’ is for any character in the text that does not exist in the pattern

B. Existing Algorithms

1. Brute-force

   a) Key features
   - Goes through each character starting from left to right in the text and compares them with the aligned characters in the pattern.
   - Anytime there is a match or mismatch pattern moves one character to the right and text aligns with it.

2. Rabin-Karp Algorithm

   a) Key features
   - It uses a rolling hash function and compares hash values of the pattern and the text.
   - If hash values match, then checks each character in the pattern with that of text from left to right.
   - Use a prime number for mod operation as the hash value can be very large.
   - Worst case $O(nm)$.

   b) Hash function features
   - Quick to calculate
   - Strong enough to prevent collisions (spurious results). For that Rabin Karp uses the following:
   - For the above example pattern “AABB”, let us assume uses ASCII values:
   - $P[i] \ast \Sigma^{m-1} \ast P[i+1] \ast \Sigma^{m-2} + \ldots + P[i+m-1] \ast \Sigma^0$
   - Rehashes by dropping the previous character of the window and adding the new character to the previous hash calculation as in [3].

   c) Algorithm Description

   It uses a rolling hash function to find matches for the pattern with the text. Once the hash value matches the hash value of the window in the text then the algorithm checks individual characters. If the hash values do not match or an individual character does not match, it moves the window by one character (sliding window!) and recalculates the hash from the existing hash – hence rolling hash function [3].

3. Boyer-Moore-Horspool Algorithm

   Horspool [2] is a simplified version of Boyer-Moore algorithm as it uses only the bad character rule instead of also good suffix rule used in original Boyer-Moore algorithm [1].

   a) Key features
   - Preprocess the pattern and create bad match table (BMT).
   - Check from right side towards left character by character.
   - If there is a mismatch, skip based on BMT.
   - Worst case complexity $O(nm)$.
   - Average case or on random text $O(n)$.
   - Best case $O(n/m)$.

   b) Algorithm Description

   Preprocess the pattern and create bad match table (BMT). Then align the pattern under the text and compare the characters from right to left one by one until there is a mismatch or until the beginning character in the pattern. If all matches, that means there is a pattern, so, output the index number of the beginning character of the pattern. If at any time there is a mismatch, look for that mis-match character of the text in the BMT table for skip value and skip the text based on this skip value starting from the right most character with that of the corresponding character in the window of the text. If they all match, it means that there is a pattern and get the index code of the matching pattern and continue from the next character right after the window, i.e., align the pattern right pass of the current window and continue until the end of the text.
4. Raita Algorithm

a) Key features
- Check right most character first, then left most and the middle one.
- Skip based on bad match table (as in Horspool algorithm).
- Match character by character.

b) Algorithm Description
First preprocess the data and create bad match table (BMT). Then align the pattern under the text and compare the right most character. If there is a match, look the left most chars if they also match look in the middle of the chars. If this also match, starting from the second char of the window compare the characters one by one until one char before the last char in the window. If they all match, it means that there is a pattern and return the first index number of the matching text window. For the next occurrences look at the last character of the text in the BMT table and skip the characters based on this –i.e. align the pattern with the text based on the skip value of the last character of the text and continue this process until the end of the text.

III. DOUBLE-HASH STRING-MATCHING ALGORITHM

The new algorithm employs features from all of the algorithms mentioned above (i.e. Rabin Karp, Boyer-Moore-Horspool and Raita).

A) Key features
- Preprocess the pattern and create Bad Match Table as in Boyer-Moore-Horspool.
- Check right most character only.
- Use two hash functions [5].
- Skip based on bad-match table if right-most character does not match
- Align or skip if one of the hash comparisons fails.

B) Algorithm Description
1. One-time operations
   - Preprocess the pattern and create bad match table that will be used to lookup for skip value when there is a mismatch of the right most character occurs, or a pattern found.
   - Get hash H1(pattern) and hash H2(pattern).

2. Multiple time operations (until the end of text)
   - Align the pattern underneath the text and check from right hand side. If the right most character matches the corresponding character (i.e. the right most character of the current window) then get the first hash value of the window (i.e., H1(Wi)) and compare it to H1(P). If the first hash values match, get the second hash value of the window (H2(Wi)) and compare it to H2(P). If both matches, shift the pattern based on the skip value of the last character of the text to the right and repeat the above steps.
   - If the right most characters do not match, look at the bad char table and skip (that is move the pattern to the right based on the skip value of the character in text which does not match). If one of the hashes don't match, we look for another occurrence of the same character in the pattern, if it exists, we align that char with that of the text otherwise move past the right most char completely and put the pattern right after that. Repeat this process until the end of the text.

C) Pseudo code
Preprocessing:
Function BMT(P)  // create skip table
pHash1 = H1(pattern)
pHash2 = H2(pattern)

Main logic:
Align pattern under the left most text position
While pattern beginning position < n-m
  Starting from right most position
    Compare last char of window with that of pattern
    If match
      // get the hash1 value for the window
      wHash1 = H1(w)
      if pHash1 == wHash1
        // get the hash2 value for the window
        wHash2 = H2(w)
        if pHash2 == wHash2
          return (beginning position of text)
    If no match or hash functions do not match
      look in BMT table and move pattern beginning position starting from the last char of the window plus the skip value. Skip value is the corresponding value for the last character of the window of the text in the BMT table.

Algorithm 2: Pseudo code for new double-hash algorithm

D) Double-Hash Functions
We have used the following two hash functions:
- BPHashHalf function
- BKDRHash Hash function [5]

BPHashHalf function processes only the first half of the string for both text window and the pattern hence significantly increasing the speed of quickly eliminating most of the possible matches due to the positive matches

1 A modified version of BPHash function [5]
of the last character while allowing some collisions (spurious matches!) which later gets picked up by a second, stronger hash function.

IV. EXPERIMENTS AND RESULTS

We have tested and compared, (see Table 2 and Figure 1), four algorithms -- Robin Karp, Boyer-Moore-Horspool, Raita and the new algorithm Double-Hash. The test is done with a MacBook Pro computer with i7 quad-core processor with 2.5 GHz speed and 16 GB memory. The data is downloaded from Amazon cloud storage S3.amazonaws website for Citi bike-trip data for June 2014. The data size is 29.9 MB zipped and 183.1 MB unzipped format. It has 936,881 trips (i.e. transactions) [6].

We have done five test cases. Test 1 featured in having many occurrences of the pattern, Test 2 had a single occurrence for a timestamp where we had an opportunity to test for digits. Test 3 had a very long string (pattern) search. Test 4 had a very short pattern search (we have added a single occurrence of “bb” to one of the transactions for testing purposes!). Test 5 tested a pattern that did not exist in our data set.

In all of the above tests our new double-hash algorithm consistently outperformed all of the other algorithms while it performed even better when the pattern being searched is longer. Raita was the second-best performer overall for all test cases. Rabin-Karp did better than Boyer-Moore-Horspool when the data is very short or non-existent but when the pattern was long Boyer-Moore-Horspool clearly did better than Rabin-Karp algorithm. Boyer-Moore-Horspool skips when there is a match or mis-match but Rabin-Karp does not skip any characters, which is the major performance issue for this algorithm.

V. COMPLEXITY ANALYSIS

For our double-hash string-matching algorithm, the performances are:

A) Worst case: Text character in each step is equal to m-2nd character in the pattern hence minimum skip takes place and hash value1 is matching for each search and therefore hash value2 is also compared and found. O(n+n+n) + 2n hash calls. i.e. O(3n) comparisons + 2n hash calls.

B) Average case: O(n/2) comparisons + n hash calls.

C) Best case: O(n/m) comparisons --i.e. the character in the text being compared doesn’t exist in the pattern for the entire search hence skipping m value in every step. There are no hash calls in this case.

VI. CONCLUSION

We have compared Rabin-Karp, Boyer-Moore-Horspool, Raita and double-hash algorithms. Rabin Karp in most cases consistently were the slowest one among all of them, as it does not use skip table hence, comparing n hash values for the text with size of n. The only time Robin-Karp did better than Boyer-Moore-Horspool is when the pattern size was very short. Boyer-Moore-Horspool was the third best performer for most of our tests as it used skip table anytime there is a mismatch, or a pattern is found. The second-best performer was the Raita algorithm. Raita algorithm was consistently better than both Rabin Karp and Boyer-Moore-Horspool as it compares last, first and middle characters before attempting to compare the remaining characters. Many mismatches speed up Raita algorithm by skipping before checking second, third characters and the entire remaining characters.

Overall the best performer in all of our tests was double-hash algorithm. Double-hash skips quickly once the last character does not match. A very quick hash function goes through only first half of the string or pattern to create hash value 1 after the last character match hence quickly determines whether there is a mismatch or not. Hash value 1 is a light hash value but created very fast for speed while some collisions are expected. If any collisions or spurious results occur, then they get filtered out by the second stronger hash value 2. We observed that the longer the pattern the better the search performance was overall.

REFERENCES

<table>
<thead>
<tr>
<th>Input:</th>
<th>2014-06-Citi-Bike-trip-data.csv</th>
<th></th>
<th></th>
<th>Number of records:</th>
<th>936881</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1:</td>
<td>Test 2</td>
<td>Test 3</td>
<td>Test 4</td>
<td>Test 5</td>
<td></td>
</tr>
<tr>
<td>Test feature:</td>
<td>Many patterns</td>
<td>Timestamp - digits test</td>
<td>Very long</td>
<td>Very short</td>
<td>Not exists</td>
</tr>
<tr>
<td>Pattern</td>
<td>Shevchenko Pl &amp; E 7 St</td>
<td>&quot;2014-07-01 00:05:06&quot;</td>
<td>Elizabeth St &amp; Hester St, &quot;40.71729&quot;, &quot;73.996375&quot;, &quot;250&quot;, &quot;bb&quot;</td>
<td>&quot;bb&quot;</td>
<td>&quot;bbb&quot;</td>
</tr>
<tr>
<td>Algorithms</td>
<td>Rabin-Karp</td>
<td>7.79</td>
<td>7.83</td>
<td>8.17</td>
<td>7.75</td>
</tr>
<tr>
<td></td>
<td>Boyer-Moore-Horspool</td>
<td>6.75</td>
<td>6.83</td>
<td>7.72</td>
<td>8.65</td>
</tr>
<tr>
<td></td>
<td>Raita</td>
<td>6.12</td>
<td>6.26</td>
<td>6.32</td>
<td>6.87</td>
</tr>
<tr>
<td></td>
<td>Double-Hash</td>
<td>5.63</td>
<td>5.74</td>
<td>5.67</td>
<td>6.43</td>
</tr>
</tbody>
</table>

Table 2: Test 1, input, pattern, algorithms and their running times

![Comparison of 4 Algorithms](image)

Figure 1: Algorithms and their running times in seconds