CS 584 - Algorithm Design and Analysis
Amee Sankhesara

String Pattern Matching Algorithms

Table of Contents

1.Abstract

String pattern matching is the problem of finding all occurrences of a character pattern in a text. This paper provides an overview of different string matching algorithms and comparative study of these algorithms. In this paper, I have evaluated several algorithms Brute force approach , Rabin-Karp algorithm, Knuth-Morris-Pratt algorithm and analysed the core ideas of these different string pattern matching algorithms. I have also compared the efficiencies of these algorithms by searching speed, pre-processing time, matching time and the key ideas used in these algorithms. This paper also discusses about the theoretical bounds on their time complexity along with experiments to show their performance empirically.

2. Introduction

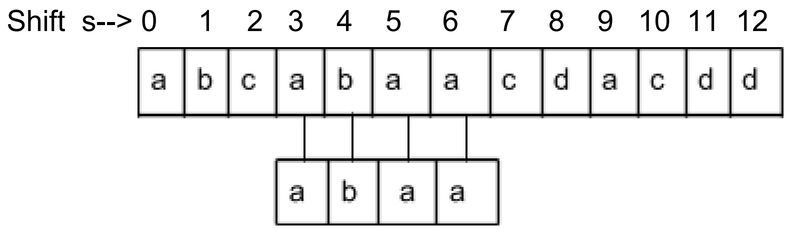
The objective of string pattern matching is to find the location of a specific text pattern within a larger body of text. We can generalize this string pattern matching problem as follow. Assume that the text is an array T[1...n] of length n and that the pattern is an array P[1...m] of length m ≤ n. We further assumes that elements of P and T are characters drawn from a finite alphabet. It may be {0,1} or {a,b,...,z}.

We say that pattern P occurs with shift s in text T if 0 ≤ s ≤ n-m and T[s+1...s+m] = P[1...m]. If P occurs with shift s in T, then we call s a valid shift; otherwise, we call s an invalid shift. The string matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T.

Below is the example of string pattern matching.

We want to find pattern P = abaa from text T = abcabaacdacdd. The pattern P occurs only once in text T at shift s = 3.

Below first is text T and second is pattern P,


Shift s--> 0 1 2 3 4 5 6 7 8 9 10 11 12
a b c a b a a c d a c d d
a b a a

Applications for string matching are, Text editor search, Database queries, DNA sequence matching, spell checker, network simulations and many more. This area expected to grow because of high demand of accuracy and speed coming from genetical science. Below sections will discuss and analyze about brute force approach, KMP and Rabin Karp string matching algorithms.

3. Brute force approach

Bruteforce is a general naive approach where you search single character of the pattern with the given text. Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.

3.1 Algorithm

Input: Strings T (text) with n characters and P (pattern) with m characters.
Output: All the indexes of text T where pattern P is found.
Bruteforce(T, P)

```
1.n = T.length
2.m = P.length
3.for s = 0 to n – m
4.      if P[1..m] == T[s+1..s+m]
5.              Print "pattern match at index s"
```

3.2 Time Complexity analysis

The brute force approach takes O(m*(n-m+1)). For example, consider text T string consists of n characters and pattern P consist of m characters. For each of the n-m+1 possible values of the shift s, the implicit loop on line 4 to compare corresponding characters must execute m times to validate the shift. Thus, the worst-case running time would be $\Theta(m(n-m+1))$, which is $\Theta(n^2)$ if m = (n/2) and this bound is tight in worst case.There is no processing time involved in bruteforce algorithm of matching pattern. So time complexity will be number of comparisons to find match.

Best Case: The best case occurs when the first character of the pattern is not present in text at all. The complexity for best case is O(n) where n is equal to the number of characters in text. Below is the example for best case.
Text = "AABCCAADDEE";
Pattern = "FAA";
To search pattern 'FAA' in text 'AABCCAADDEE' it will do n comparisons where n is the length of text. So it has a complexity of O(n).

Worst Case: The number of comparisons in the worst case is (m*(n-m+1)). Where m = length of pattern and n = length of text. So time complexity in the worst case is O(m*(n-m+1)).
Below are some examples of worst case.

1) When all characters of the text and pattern are the same.
For example, text T = "ZZZZZZZZZZZZZZZZZZZZ";
and Pattern P = "ZZZZ";

2) Worst case also occurs when only the last character is different.
For example, text T = "DDDDDDDDDDDDDDDDDDDDDD";
and Pattern P = "DDDDDA";

4. The Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt (KMP) string searching algorithm can improve performance of brute force algorithm from complexity O(mn) to O(m+n) by keeping track of information gained from previous comparisons.

The key observation in the KMP algorithm is this: If currently we already matched the first k characters of the pattern (with $2 \le k < m$) but the next character in the text mismatches the next character in the pattern, we don’t necessarily need to restart the match from the beginning of the pattern.

For example, if the pattern is abcabd and we partially matched abcab, if the next character isn’t d then we can shorten the partial match to ab (which is both a prefix and a suffix of abcab) instead of restarting from scratch. Let s be the currently matched k-character prefix of the pattern. If t is some proper suffix of s that is also a prefix of s, then we already have a partial match for t. Compute the longest proper suffix t with this property, and now re-examine whether the next character in the text matches the character in the pattern that comes after the prefix t. If yes, we advance the pattern index and the text index. If no, we repeat this suffix/prefix process until the prefix is empty and we still fail to match the next character.

4.1 Algorithm

COMPUTE-PREFIX-FUNCTION(P)

Input : The pattern string.

Output : The array which has stored where prefixes are located

```
1. m = P.length
2. let π[1...m] be a new array
3. π[1] = 0
4. k = 0
5. for q = 2 to m
6.     while k > 0 and P[k+1] ≠ P[q]
7.         k = π[k]
8.     if P[k+1] == P[q]
9.         k = k + 1
10.    π[q] = k
11. return π
```

KMP-MATCHER(T,P)
Input: Strings T (text) with n characters and P (pattern) with m characters.
Output: All the indexes of text T where pattern P is found.

```
1. n = T.length
2. m = P.length
3. $\pi$ = COMPUTE-PREFIX-FUNCTION(P)
4. q = 0                                  //number of characters matched
5. for i = 1 to n                         // scan the text from left to right
6.     While q > 0 and P[q+1] ≠ T[i]
7.         q = $\pi$[q]                   // next character does not match
8.     if P[q+1] == T[i]
9.         q = q + 1                      // next character matches
10.    if q == m                          // is all of P matched?
11.        print "Pattern occurs with shift" i-m
12.        q = $\pi$[q]                   // look for the next match
```

Above algorithms work as follows :
Preprocessing (Compute-Prefix-Function) :
In the preprocessing part, we calculate values in π[] (longest prefix suffix values). To do that, we keep track of the length of the longest prefix suffix value (we use k variable for this purpose) for the previous index. We initialize π[1] and k as 0. Now we will use for loop from q=2 to m (m=length of pattern String) to find the longest prefix suffix values. In for loop these comparison will be performed. If P[k+1] and P[q] match, we increment k by 1 and assign the incremented value to π[q]. If P[q] and P[k+1] do not match and k is not 0, we update k to π[k]. See Compute-Prefix-Function () in the above algorithm for details.
Matching (KMP-Matcher):
Unlike Naive algorithm, where we slide the pattern by one and compare all characters at each shift, we use a value from π[](array having calculated prefix values using preprocessing function) to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.
Following are the steps from which explains how to use values of π[] to know a number of characters to be skipped or to decide next positions.
-We start comparison of P[q+1] with q = 0 with characters of current window of text.
-We keep matching characters T[i] and P[q+1] and keep incrementing i and q while P[q+1] and T[i] keep matching.
-When we see mismatch

- We know that characters P[1..q] match with T[i-q...i] (Note that q start with 0 and increment it only when there is a match)
- We also know (from above definition) that π[q] is count of characters of P[1…q] that are both proper prefix and suffix.
- From the above two points, we can conclude that we do not need to match these π[q] characters with T[i-q…i] because we know that these characters will anyway match.

4.2 Time Complexity analysis

Preprocessing time of KMP is Θ(m) which we can show by the aggregate method of amortized analysis. (Time complexity of COMPUTE-PREFIX-FUNCTION)

- Here, the only tricky part is showing that the while loop of lines 6–7 executes O(m) times altogether. We shall show that it makes at most m-1 iterations. We start by making some observations about k.
- First, line 4 starts k at 0, and the only way that k increases is by the increment operation in line 9, which executes at most once per iteration of the for loop of lines 5–10. Thus, the total increase in k is at most m-1.
- Second, since k < q upon entering the for loop and each iteration of the loop increments q, we always have k < q. Therefore, the assignments in lines 3 and 10 ensure that π[q] < q for all q = 1,2,...,m which means that each iteration of the while loop decreases k.
- Third, k never becomes negative. Putting these facts together, we see that the total decrease in k from the while loop is bounded from above by the total increase in k over all iterations of the for loop, which is m - 1.
- Thus, the while loop iterates at most m - 1 times in all, and COMPUTE-PREFIX-FUNCTION runs in time Θ(m).

Matching time of KMP is Θ(n) which we can show by the aggregate analysis. (Time complexity of KMP-MATCHER function)

- To show that the running time of KMP-MATCHER is Θ(n), we'll show that the total number of executions of the while loop of line 6 is Θ(n). Observe that for each iteration for the for loop of line 5, q increases by at most 1, in line 9. This is because π(q) < q. On the other hand, the while loop decreases q. Since q can never be negative, we must decrease q fewer than n times in total, so the while loop executes at most n − 1 times. Thus, the total runtime is Θ(n).

5. Rabin Karp Algorithm

The Naive String Matching algorithm slides the pattern one by one. After each slide, it one by one checks characters at the current shift and if all characters match then prints the match. Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one. But unlike the Naive algorithm, the Rabin–Karp algorithm seeks to speed up the testing of equality of the pattern to the substrings in the text by using a hash function. A hash function is a function which converts every string into a numeric value, called its hash value; for example, we might have hash("hello")=5. The algorithm exploits the fact that if two strings are equal, their hash values are also equal. Thus, string matching is reduced (almost) to computing the hash value of the search pattern and then looking for substrings of the input string with that hash value. Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for following strings.
1) Pattern itself. 2) All the substrings of text of length m.

However, there is a problem with this approach. First, because there are so many different strings and so few hash values, some differing strings will have the same hash value. If the hash values match, the pattern and the substring may not match; consequently, the potential match of search pattern and the substring must be confirmed by comparing them; that comparison can take a long time for long substrings. This kind of hits when hash value of substring of text and pattern is the same but that substring of text is not same as pattern string it is called spurious hit or hash collision. Luckily, a good hash function on reasonable strings usually does not have many hash collisions(spurious hit), so the expected search time will be acceptable. So we need to calculate hash values efficiently to get good hash value.Since we need to efficiently calculate hash values for all substrings of size m of text, we must have a hash function which has the following property.

Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say *hash(txt[s+1 .. s+m])* must be efficiently computable from *hash(txt[s .. s+m-1])* and *txt[s+m]* i.e., *hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1])* and rehash must be O(1) operation.

The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is numeric value of a string. For example, if all possible characters are from 1 to 10, the numeric value of “122” will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value. Rehashing is done using the following formula :

hash(txt[s+1 .. s+m]) = (d (hash(txt[s .. s+m-1]) – txt[s]*h) + txt[s + m]) mod q
hash(txt[s .. s+m-1]) : Hash value at shift s.
hash(txt[s+1 .. s+m]) : Hash value at next shift (or shift s+1)
Where d = Number of characters in the alphabet, q = A prime number,h= d^{m-1} mod q

5.1 Algorithm

RABIN-KARP-MATCHER(T,P,d,q)
Input : Text String(T),Pattern String(P),Number of characters in the alphabet(d),prime number(q)
Output : All the indexes of text T where pattern P is found.

```
1. n = T.length
2. m = P.length
3. h = $d^{m-1}$ mod q
4. p = 0
5. $t_0$ = 0
6. for i=1 to m                                // preprocessing
7.        p = (dp + P[i]) mod q
8.        $t_0$ = (d$t_0$ + T[i]) mod q
9. for s = 0 to n - m                                  // matching
10.       if p == $t_s$
11.            if P[1...m] == T[s+1...s+m]
12.                 print “Pattern occurs with shift” s
13.       if s < n-m
14.            $t_{s+1}$ = (d($t_s$-T[s+1]h)+T[s+m+1]) mod q
```

Above algorithm works as follows.
All characters are interpreted as radix-d digits. The subscripts on t are provided only for clarity; the program works correctly if all the subscripts are dropped. Line 3 initializes h to the value of the high-order digit position of an m-digit window. Lines 4–8 compute p as the value of P[1....m] mod q and t_0 as the value of T[1....m] mod q. The for loop of lines 9–14 iterates through all possible shifts s, maintaining the following invariant:
Whenever line 10 is executed, t_s = T[s+1…..s+m] mod q.
If p = t_s in line 10 (a "hit"), then line 11 checks to see whether P[1....m] = T[s+1….s+m] in order to rule out the possibility of a spurious hit. Line 12 prints out any valid shifts that are found. If s < n - m (checked in line 13), then the for loop will execute at least one more time, and so line 14 first executes to ensure that the loop invariant holds when we get back to line 10. Line 14 computes the value of t_{s+1} mod q from the value of t_s mod q in constant time using equation shown in line 14.

5.2 Time Complexity analysis

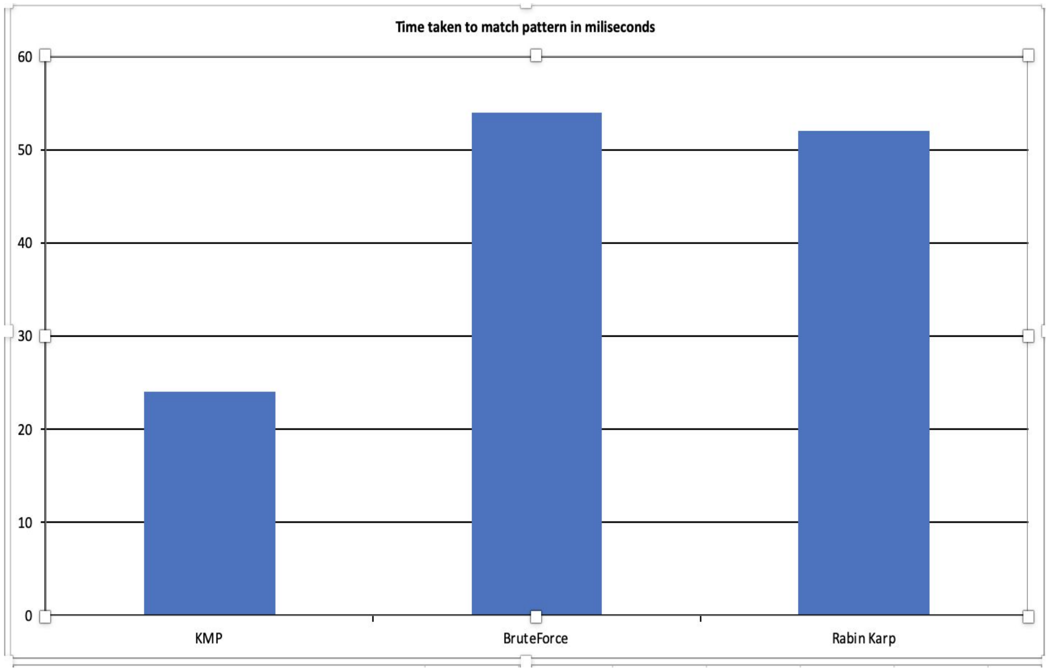
RABIN-KARP-MATCHER takes Θ(m) preprocessing time, and its matching time is Θ((n-m+1)m) in the worst case, since like the naive string-matching algorithm the Rabin-Karp algorithm explicitly verifies every valid shift. If P = a^m and T = a^n, then verifying takes time Θ((n-m+1)m), since each of the n-m+1 possible shifts is valid. Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of text match with hash value of pattern. For example pattern = "AAA" and text = "AAAAAAA". So for each hash value it will compare whole substring to pattern in this case. So worst case time complexity of Rabin-Karp algorithm is O(m(n-m+1)). In many applications, we expect few valid shifts—perhaps some constant c of them. In such applications, the expected matching time of the algorithm is only O((n-m+1)+cm)=O(n+m), plus the time required to process spurious hits. We can base a heuristic analysis on the assumption that reducing values modulo q acts like a random mapping from Σ* to Zq (where Σ={0,1,2,3,...,9} and d=|Σ| and each character is a digit in radix-d notation). We can then expect that the number of spurious hits is O(n/q), since we can estimate the chance that an arbitrary t_s will be equivalent to p, modulo q, as 1/q. Since there are O(n) positions at which the test of line 10 fails and we spend O(m) time for each hit, the expected matching time taken by the Rabin-Karp algorithm is O(n)+O(m(v+n/q)), where v is the number of valid shifts. This running time is O(n) if v=O(1) and we choose q ≥ m. That is, if the expected number of valid shifts is small (O(1)) and we choose the prime q to be larger than the length of the pattern, then we can expect the Rabin-Karp procedure to use only O(n+m) matching time. Since m ≤ n,this expected matching time is O(n).

6. Result analysis

I have taken a long string which have more than a million characters and pattern with more than 10000 characters.
For example, string with millions of character "a" and pattern with more than 10000 character "a" and last character added as "b".
So, in this example there is not a pattern match in string so the Rabin karp exhibits its worst case O(mn) But on the other hand The KMP reduces this to O(*n*) time using precomputation to

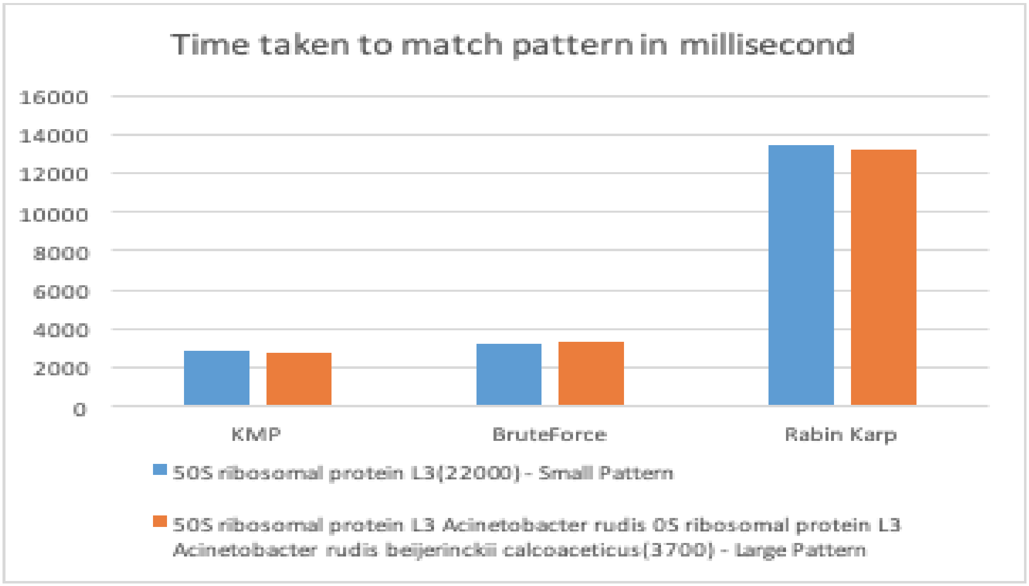
examine each text character only once. You can see this result by run time taken for each algorithm. For each graph, the tests were run 5 times back to back and the graphs show the average time taken by each algorithm.

String Matching Algorithms	**Average Time taken to match pattern**
KMP	24 ms
BruteForce	54 ms
Rabin Karp	52 ms



I have taken another interesting example of protein DNA sequence match. I got a large file from Genome project about protein DNA. The size of file is more than 1 GB. This can contain various protein DNA patterns.
For example, I took two different strings to search. First string (“**50S ribosomal protein L3”** (22000 matches)) was relatively smaller compared to second string (“**50S ribosomal protein L3 Acinetobacter rudis 0S ribosomal protein L3 Acinetobacter rudis beijerinckii calcoaceticus”** (3700 matches)). Moreover first string have more number of matches 22000 compared to large string having 3700 matches. For each graph, the tests were run 5 times back to back and the graphs show the average time taken by each algorithm.

String Matching Algorithms	**Small string pattern average time taken in ms**	**Large string pattern average time taken in ms**
KMP	2816	2714
BruteForce	3217	3333
Rabin Karp	13502	13191



7. Conclusion

This analysis reviews some typical string matching algorithms to observe their performance under various conditions and gives an insight into choosing the efficient algorithms. By analyzing these string matching algorithms, it can be concluded that KMP string matching algorithm is very efficient. KMP decreases the time of searching compared to the Brute Force algorithm and Rabin karp algorithm. Rabin karp algorithm sometimes gives almost the same performance as brute force as shown in the first graph (Test case of having last character mismatch of pattern and text). Sometimes Rabin karp gives worse performance than brute force as shown in the second graph (Test case of finding small and large pattern in text). It is due to the large number of hash collisions (substring of text having the same hash value as pattern’s hash value even though substring is not same as pattern) occurred when running the algorithm. So Rabin karp’s performance is totally depend on how efficient hash value is generated but it always gives worse performance compared to KMP algorithm. In real world Rabin karp is used in plagiarism detection applications. In real world KMP algorithm is used in those applications where pattern matching is done in long strings, whose symbols are taken from an alphabet with little cardinality. A relevant example is the DNA alphabet, which consists of only 4 symbols (A,C,G,T). Innovation and creativity in string matching can play an immense role for getting time efficient performance in various domains of computer science.

8. References

CLRS - Chapter - 32 - String Matching
https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm
https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm
https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/
https://github.com/
https://www.ncbi.nlm.nih.gov/guide/dna-rna/