String Processing Algorithms

Oana Diaconescu

Politehnica University of Bucharest, Romania
oanadiaconescu7@gmail.com

Abstract. String matching algorithms are fundamental to any computer application involving text-editing, DNA sequence analysis, digital forensics, or plagiarism detection. A string matching algorithm can search for all occurrences of a pattern inside a given string. Such algorithms can be applied either to a single input pattern or to multiple input patterns. As a solution to these problems, this paper compares the performance of three different algorithms: Knuth–Morris–Pratt, Rabin-Karp and Aho-Corasick, based on various evaluation criteria (text size, number of occurences, alphabet type, etc).

Keywords: Pattern · String · Text-Editing · Rabin-Karp · KMP · Aho-Corasick

1 Introduction

1.1 Problem Statement

Single Pattern Matching The *Exact Pattern Matching Problem* is stated as follows: given a text T[1, ..., n] and a pattern P[1, ..., m], both of which are strings over the same alphabet Σ find all occurrences of P in T.
A position $s \in \{0, ..., n-m\}$ is considered a *valid shift* in text T, if the pattern P is found in T as shifted by s:

$$P[i] = T[s+i], \quad \forall i \in \{0, ..., m-1\}$$

A string matching algorithm determines all valid shifts for a given pattern inside a block of text.
The single pattern matching problem has various approaches:

- Verify whether a given pattern occurs in the text or not
- Find the number of occurrences for a given pattern inside a text
- Determine a suffix/prefix for a given string, etc.

Multiple Pattern Matching In this case, we are given a text T of length n and want to search simultaneously for a set of strings P = $\{p_1, p_2, ..., p_r\}$. The multiple pattern matching problem is a straightforward generalization of single pattern matching and could be solved by extending single pattern algorithms.

1.2 Applications

String matching algorithms can be used for **analysing DNA sequences**. Since the canonical structure of DNA has four bases (A, C, T, and G), it can be converted to a string and searched for specific pattern matches. Amino-acid sequences of proteins or certain nucleotides can be traced in order to find mutations, infer heredity or reveal evolution and genetic diversity.

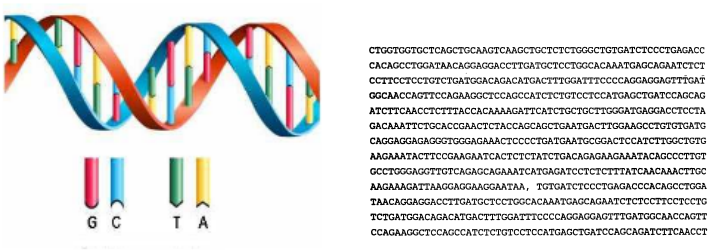

Fig. 1. DNA structure - *Source*

Another important application for string matching algorithms is in **digital forensics**. In this case, algorithms are designed to locate specific text strings of interest to the investigation inside documents such as: email, Internet browsing history,calendar appointments, network activity logs, and system logs.

In a similar manner, **spam filters** use string processing techniques to report unwanted and unsolicited data. A dataset of keywords is searched within the content of an email and in case it matches above a certain threshold, the email is referred to as spam.

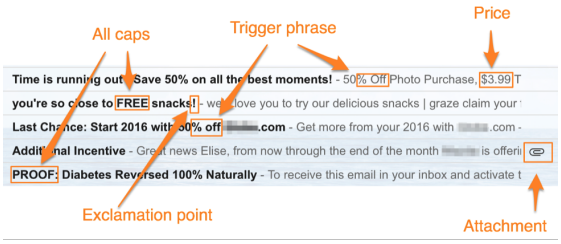

Fig. 2. Spam Filtering - *Source*

String processing is also used for **plagiarism detection**. Rabin-Karp is one of the algorithms used to detect the similarity levels of two strings. The plagiarism level determination is based on comparing the hash value of both documents.

1.3 Selected solutions

a) Rabin-Karp

The Rabin-Karp algorithm makes use of hash functions and the rolling hash technique to find the exact pattern match in a string. Rabin-Karp uses a rolling hash to filter out pieces of the text which do not match the pattern, and then checks for a match at the remaining positions. When a hash-value match is found, the algorithm starts matching individual characters.

b) Knuth–Morris–Pratt

The idea behind the KMP algorithm is based on the propriety that a pattern can have the same sub-pattern appear more than once. The algorithm begins with a preprocessing phase, where an array is created providing the next position in the pattern to be processed after a mismatch. For the searching part, this array is used to avoid re-examining the characters which have been previously checked, thus limiting the number of comparison required.

c) Aho-Corasick

The Aho-Corasick algorithm builds a data structure similar to a trie with some additional links and then constructs a finite state machine, allowing fast transitions between failed string matches.

1.4 Evaluation criteria for selected solutions

After completing a performance evaluation of the three algorithms, the goal is to answer the following questions:

- Which algorithm has the best overall performance in single pattern matching scenarios?
- Is Aho-Corasick the most efficient algorithm in case of multiple pattern matching?
- Are there two specific problems which could be solved more efficiently using KMP rather than Rabin-Karp?

Performance will be tested against the following specifications:

1. The size of the alphabet ($|\Sigma| \in (2, 256)$)
2. The size of the input pattern/text
3. The frequency of the pattern inside the text
4. The number of input patterns

The algorithms' evaluation will be based on criteria such as time and space efficiency, consistency, reliability and versatility.

2 Solutions

2.1 Description

Rabin-Karp

Rabin-Karp is an algorithm built around the idea of modular hashing. Hashing is a technique of mapping a large piece of data to a smaller, easily computable value. Furthermore, Rabin-Karp uses a rolling hash function, which provides a way of calculating a hash value without having to rehash the entire string.
Given a pattern P of length M and a block of text T of length N, Rabin Karp returns an array with all ocurrences of P inside T. The procedure is:

Step 1 Compute the hash value for P

Step 2 Iterate through each letter of T computing the hash value for a 'window' of M consecutive letters

Step 3 Compare the hash value for P with the hash value calculated for T at the current iteration i, i > M

Step 4 In case of a hash value match, perform a naive comparison between the pattern and the 'window' it matched

Step 5 Otherwise, continually remove the last letter in the rolling hash and add the next letter from T

As a hashing function, we are going to use the Rabin fingerprint:

$$h_i = t_i B^{M-1} + t_{i+1} B^{M-2} + ... + t_{i+M-1} B^0 \ (mod Q)$$

B is the base of the polynomial and Q is a prime number used to reduce the size of the hash value. In this form, the "rolling" hash function can be updated in constant time:

$$h_{i+1} = (h_i - t_i B^{M-1}) B + t_{i+M} \ (mod Q)$$

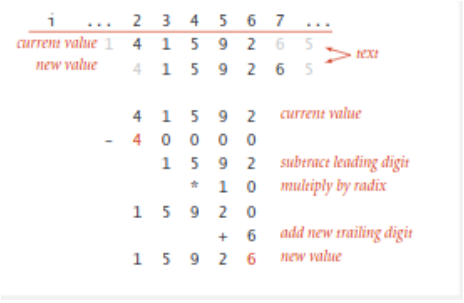

Fig. 3. Rolling Hash example - *Source*

If Rabin Karp would be based only on hash-value comparisons, it would be considered a **Monte Carlo** algorithm. Monte Carlo algorithms are randomized algorithms which may not always give the right answer, as they rely on probabilities. This would happen because the hashing technique is vulnerable to **collisions**, or false matches between different strings with the same hash value.

Thus, the additional comparison performed after matching hash values is what guarantees the correctness of Rabin Karp. However, let's consider the following text:

$$aaaaaaaaaaaaaaaaaaaaaa$$

If we were to search for pattern 'a', the performance of Rabin Karp would degrade to $O(patternSize * textSize)$, which is equivalent to that of the naive pattern matching algorithm. Thus, performing the verification comparisons can drastically deteriorate the efficiency of Rabin Karp in some worst case scenerios.

Therefore, we need to find values for the base B and the modulus Q, as to minimize the probability of collisions. Since we are mapping character values to integers, we can choose the base B as equal to the size of our alphabet. Thus, we can avoid trivial collisions and guarantee the fact that each base conversion results in an unique number. However, after applying the modulus Q, we can still obtain equal results for different strings.

Suppose for any fixed location i, the probability of an incorrect match is δ. If the text size is N and the pattern size is M, then:

$$(N - M + 1)\delta \ \leq \ t\delta$$

We want the final probability of error to be at most 1/2:

$$\delta \ \leq \ \frac{1}{2t}$$

How can we chose a uniformly random prime number in the range 2...K with a chance smaller than 1/(2t) of obtaining (at most) p values which result in collisions? It suffices to choose K large enough such that there are at least $2pt$ primes between 2 and K. This claim can be proved by the Prime Number Theorem.

Therefore, choosing a random prime (from a sufficiently large set of primes) for the modulus Q, ensures the fact that there are no specific inputs that would cause the algorithm to run slowly.

b) Knuth–Morris–Pratt

Similar to Rabin-Karp, Knuth–Morris–Pratt returns an array of all occurrences of a pattern P in a text T. The idea behind KMP is that after a mismatch, the pattern itself allows us to determine where to begin the next match, avoiding

re-examination of previously matched characters. There are two phases for the Knuth–Morris–Pratt algorithm: *preprocessing* and *searching*.

Preprocessing consists of building a "prefix table", which is used to skip character comparisons while matching:

Step 1 Define an array π with the size equal to that of the pattern

Step 2 Iterate through the pattern starting from index i = 1 and compare $\pi[i]$ with $\pi[patternMatch]$, where patternMatch is the size of the prefix match found at the current point

Step 3 If they are matched, set $\pi[patternMatch] = i + 1$ and increment both values

Step 4 Otherwise, check the value of index i: if i=0, then set $\pi[patternMatch] = 0$ and increment the patternMatch value by one; if i $\neq$ 0 then set i = $\pi[i - 1]$.

Step 5 Repeat the steps until π is filled

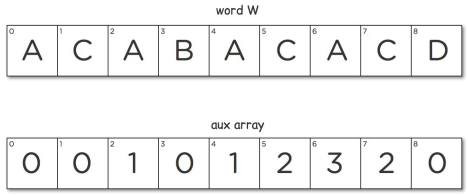

Fig. 4. Preprocessing - *Source*

Searching consists of using the "prefix table", while performing comparisons between the text and the pattern. In case of a mismatch, or in case an occurrence has been found, the value for the current index in the π array is used in order to go back to the point after the longest prefix-suffix.

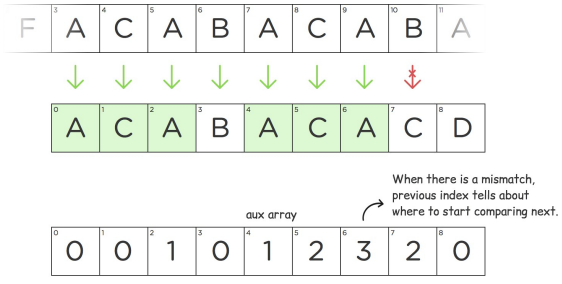

Fig. 5. Searching - *Source*

c) Aho-Corasick

Aho-Corasick takes in an array of k patterns $P_1, .., P_k$ and a block of text T, returning a hashmap which maps each pattern to an array of its occurrences. Just like KMP, Aho-Corasick has a **preprocessing** and a **searching** phase. During preprocessing, it constructs a finite state automata for the set of predefined patterns which are supposed to be found in the text string. The transitions between the defined states are represented by the characters existing in the particular pattern.As for searching, it uses the pre-constructed auromata to search the text for matches. Therefore, Aho-Corasick can be considered a generalization of the KMP algorithm.

Step 1 Construct a trie (an array of Node structures), where each edge of the trie is labeled by some letter. Thus, if we write out the labels of all edges on the path, we will be able to say that each leaf vertex will correspond to one pattern from the set.

Step 2 Traverse the trie from the root to the leaves and find the maximum-length suffix of the current substring that also is the prefix of some pattern in the trie. Store the link to the last character of the maximum suffix of the current substring.

Step 4 In order to process the text, go over each of its letters and, starting from the root of the tree and try to determine if there is an edge from the current state to the letter in the text. If there isn't a direct edge, follow suffix links until finding an edge with this letter or until reaching root.

Step 5 If an edge was reached by following suffix links, we check every state that can be reached from the current prefix, in order to find all possible patterns which use it.

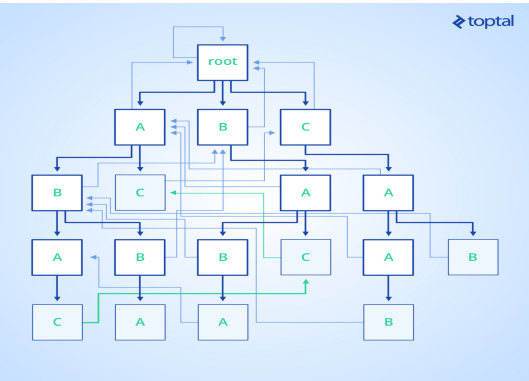

Fig. 6. Trie Structure, Aho-Corasick - *Source*
Tree edges = deep blue, Suffix links = light blue

2.2 Complexity analysis

a) Rabin-Karp

Consider the pattern P with length m and the text T with length n. The operations performed during Rabin-Karp are:

1. Calculating the hash for the pattern: **m + 1** operations
2. Computing the hash for the first "window" in the text: **m + 1** operations
3. Calculating B^{m-1} (which will be used at every step within the formula $h_{i+1} = (h_i - t_i B^{M-1})B + t_{i+M}\ (mod Q)$): **m** operations
4. Comapring the hash values (one per each m sized substring in T): **n-m+1** operations
5. If there is a total of k hash matches, then we compare each of those patterns with the current "window", character by character: **k*m** operations
6. Calculating the new hash value: **n - m + 1** operations

The complexity of Rabin-Karp is highly dependent on the number of naive comparisons which have to be performed after matching hash values.

Therefore, the **best and average-case** running time of Rabin-Karp is $O(m + n)$. However, in case of a very poor hash function that results in a false positive at each step, or in case of an unfavorable input(such as the previous example: 'aaaa...' and 'a'), m letters will be checked n times(at each step) resulting in a **worst-case** running time of $O(nm)$

b) Knuth–Morris–Pratt

In the case of KMP, there is also a pattern P with length m and the text T with length n.

1. For the preprocessing phase, there is a loop which iterates through all characters in the pattern, at each step performing a comparison and assigning a value to the current position: **2m** operations
2. During the searching phase, there is a comparison between a text and a pattern character; in case of matching, we advance a position in both: at most **2n** operations
3. In case of mismatch, or after finding a complete match for pattern p, we go back to the position after the longest prefix-suffix or simply advance in the array: at most **n** operations (if there is no suffix-prefix in the pattern)

By summing up the results, we obtain the fact that KMP has an overall complexity of $O(m + n)$. Since the size of the pattern is smaller than that of the text, we can write KMP's complexity as $O(n)$

c) Aho-Corasick

Let m be the length of the text and n the total length of the pattern strings ($n = |P_1| + ... + |P_k|$).

There are multiple ways of implementing the nodes in the Aho-Corasick trie. The algorithm will have a complexity of $O(m + n + z)$ in the case of an array, where z is the **total number of matches.**. In this case, the additional memory requirement will be $O(n * q)$, where q is the length of the alphabet, since that is the maximum number of children a vertex can have.

By choosing a structure with $O(log(q))$ access to its elements, we will have an additional memory requirement of $O(n)$. However, the complexity of the whole algorithm will be $O((m + n) * log(q) + z)$.

For the implementation we will chose a hash table, which has $O(n)$ additional memory, and generates a complexity of $O(N + L + Z)$ for the whole algorithm.

2.3 Advantages and Disadvantages

Rabin-Karp

Advantages

- In practice, it has a good average-case complexity of O(n+m) and a reliable hashing function can boost its perfomance
- Because of its hashing technique, it is efficient in multiple pattern matching and plagiarism detection
- It is easy to implement

Disadvantages

- In worst case scenarios, it performs just as bad as brute force matching

Knuth-Morris-Pratt

Advantages

- Unlike Rabin-Karp, it is guaranteed worst-case efficient, with a linear complexity of $O(n)$
- Processing can be done a single time, so KMP proves effective when searching for the same pattern a multiple number of times
- It never moves backwards therefore it is good for processing large files

Disadvantages

- As the size of the alphabet increases, KMP becomes less effective
- It requires additional space for array storage

Aho-Corasick

Advantages

- It has a complexity of $O(n + m + z)$, where z is total number of occurrences of patterns in the text; thus, it works better than KMP, which has a complexity of $O(n * k + m)$, where k is the number of patterns
- It is suitable in cases with large numbers of keywords, since all keywords can be simultaneously matched in one pass

Disadvantages

- It requires additional space for storing the trie
- Does not perform very well with large data; as the size of the automaton increases, its performance decreases

3 Performance evaluation

Single pattern matching can be solved using either Rabin-Karp or Knuth-Morris-Pratt. We are going to compare the results of the two methods, trying to determine which one has the best overall performance.Given that Aho-Corasick can be considered a generalization of KMP, we are going to verify if its efficiency in the case of **multiple pattern matching** exceeds that of Knuth-Morris-Pratt. Lastly, we consider two specific string processing problems for which the preferred solution would be KMP rather than Rabin-Karp.

There are a number of criteria which can influence the performance of string processing algorithms:

- The size of the alphabet
- The size of the input text
- The number of occurrences for one pattern
- The number of patterns to search for

The tests ran on the algorithms were crafted based on these criteria, seeking to determine their results in best, average, and worst case scenarios.
As stated before, the single pattern matching problem is tested on Rabin-Karp and KMP, and their results are compared against each other to determine their correctness. Simlarly, multiple pattern matching is tested on RK, KMP and Aho-Corasick.

Since we know that the complexity of KMP ($O(n)$) is better than that of Rabin-Karp ($O(nm)$), we can assume that any problem related to single pattern matching would be more efficiently solved by KMP. However, since the avarage complexity of Rabin-Karp is actually better in practice, this assumption is not guaranteed. Therefore, we will examine the performance of the two against the following problems:

1. Determine if a given string forms suffix and a preffix for another string

2. Determine if a string is a rotation of another string

The tests were performed on system with an Intel Core i7-4720HQ CPU @ 2.60GHz CPU, 4GB RAM, running Ubuntu 18.04.1. The obtained running time for each algorithm is expressed in **miliseconds**.

Table 1. Running time of KMP and RK based on the alphabet size

Text size	**Alphabet size**			
	4		256	
	KMP	RK	KMP	RK
10	1,208	**0,778**	**0,994**	1,033
1000	**6,650**	6,928	8,226	**7,234**
1000000	**4819,695**	8038, 095	**4128,058**	7134,171

Observing the results, we can see that the alphabet size has little impact on the efficiency of the two algorithms. The running time of KMP and RK is fairly the same in the case of a four character alphabet or a 256 character alphabet. However, we can oberve the fact that KMP performed better than RK in the majority of the tests.

Table 2. Running time of KMP and RK based on the number of occurrences

Text size	**Number of occurrences**			
	≈ 1		$\approx$ textSize	
	KMP	RK	KMP	RK
10	1,542	**1,259**	1,603	**1,261**
1000	**5,395**	9,216	**6,476**	11,328
1000000	**381,537**	4275,701	**4128,058**	8714,759

We can see that the running times of the two algorithms are greater in case of patterns with large number of occurrences. This result is not unexpected, since the processing complexity for matching is greater than that for mismatching in both algorithms.

After carrying out the tests, we can conclude that the most impactful criteria when it comes to the algorithms' efficiency is the size of the text. The greater the size of the text, the worse the performance of the algorithm.

Table 3. Running time for multiple pattern matching on KMP RK and AC (Small number of patterns)

Text size	**Algorithm**		
	AC	RK	KMP
10	6,692	1,782	**1,632**
1000	49,155	**45,490**	65,369
1000000	**1469,564**	2832,619	8844,178

Table 4. Running time for multiple pattern matching on KMP RK and AC (Large number of patterns)

Number of patterns					
10000			1000000		
AC	RK	KMP	AC	RK	KMP
8112,539	**3579,358**	8993,391	**4686,761**	35094,991	10367,304

Based on the previous tests, we can conclude that Aho-Corasick is the most efficient algorithm for multiple pattern matching, since it works well on large text sizes as well as large datasets of patterns. Rabin Karp is also a reasonable choice for multiple pattern matching, although its performance decreases as the load of data becomes greater.

4 Conclusion

Each of the algorithms analyzed in this paper have important practical applications. As we have seen, Aho-Corasick is an algorithm used for multiple pattern matching, suited for processing large sets of data. For this reason, Aho–Corasick was used for the original Unix command fgrep, as well as other multiple pattern processing applications such as the signature based anti-virus. Rabin-Karp also performs well in multiple-pattern matching, but it cannot handle large sets of data as well as Aho-Corasick. Rabin-Karp's most important application is plagiarism detection, but it can also be used as an alternative for single pattern matching. However, its vulnerability to collisions and its unfavorable worst-case complexity, are the reason why Knuth-Morris-Pratt is generally preferred in the case of single pattern matching. Its complexity is linear and guaranteed and it generally performs well under any conditions. Nonetheless, in the case of KMP, a small alphabet size is preferred for better performance, and this is why KMP is generally used for DNA sequence analysis.

References

1. Donald E. Knuth, James H. Morris, Jr. and Vaughan R. Pratt, *Fast Pattern Matching in Strings*
2. Vidya SaiKrishna, Prof. Akhtar Rasool and Dr. Nilay Khare, *String Matching and its Applications in Diversified Fields*
3. Akhtar Rasool,Amrita Tiwari, Gunjan Singla an Nilay Khare *String Matching Methodologies:A Comparative Analysis*
4. https://web.stanford.edu/class/cs97si/10-string-algorithms.pdf
5. https://www.cs.ubc.ca/ hoos/cpsc445/Handouts/kmp.pdf
6. https://algs4.cs.princeton.edu/lectures/53SubstringSearch.pdf
7. http://www.cs.cmu.edu/afs/cs/academic/class/15451-f14/www/lectures/lec6/karp-rabin-09-15-14.pdf
8. https://www.toptal.com/algorithms/aho-corasick-algorithm