String Processing Algorithms

Catalin-Stefan Barus

University Politehnica of Bucharest, Faculty of Automatic Control and Computer Science

Abstract

Abstract. This paper analyses the characteristics of one of the most efficient and used pattern searching algorithm, used as the basis for most text editors using a find function(Knuth-Morris-Path or KMP), as well as an algorithm that has seen uses in a lot of natural language processing applications (Edit Distance).

Keywords: String processing algorithms, Knuth-Morris-Path, Edit distance, longest common substring, Levenshtein distance, performance, patterns, time complexity.

1 Introduction

1.1 The main description of the problem

One of the most fundamental problems encountered in computer science is the processing of strings. There is no getting around the fact that most data we encounter, when using any sort of computer, is stored inside of files which are comprised of multitudes of strings, so most key components of any processing system require the necessary knowledge of manipulating these strings. Throughout the history of computer science there have been many algorithms designed to solve most problems in string processing and the most important problem, or perhaps the turning point for any serious data processing was the "The longest common substring problem". Algorithms trying to solve this problem generally fall under the "Pattern Searching" algorithms.

1.2 The use of the chosen problem in practice

Most real case uses and applications of pattern searching algorithms are, as the name implies and the original problem suggested, finding the longest common substring between two strings, essentially finding words or whole sentences in a text. Every program or engine that has a find option implemented (Text editors, PDF viewers, search engines, online dictionaries), makes use of one of these algorithms depending on the required end result or the desired complexity. Additionally, some algorithms have further enhanced this functionality, not only for detecting matching words in two strings, but also similar ones. This has led to large scale uses in computational biology and natural language processing. Software that detects typos, or engines that can make suggestions based on the words we typed, are all important feats achieved with the use of these algorithms. Most programming languages implement these sorts of algorithms using a few functions and already defined data types/frameworks.

1.3 The solutions for the chosen problem

The two chosen algorithms are, perhaps, the most notable algorithms for solving their respective problems. Firstly, we have the first ever linear-time algorithm for exact string matching, the "Knuth-Morris-Pratt" algorithm, developed by analyzing the brute force or naïve algorithm, which still remains one of the best choices for solving this problem, given the enhanced version is used. Secondly, we have another linear algorithm, but this time for computing the minimum cost of a sequence of edit operations, needed to change an input string to another given one. Particularly, what really piqued people's interest with this algorithm is not computing the actual edit distance, but rather finding a sentence with minimal edit distance from the input string, this was the basis for string correction problems.

1.4 Testing criteria

The validity of the implementation of these algorithms will be done by taking into account the correctitude of the outputs on a sufficiently large collection of input data and number of tests. Ideally, the input tests will need to get separated into two sets. One set for the enhanced version of the Knuth-Morris-Pratt, which uses the LPS Theoretical Idea (Longest Proper Prefix/Suffix) and will be used during the analysis of this paper, compared side by side with the original (naïve) method that moves to the next position in the target text when it finds a mismatch between it and the given word. Additionally, the tests will also take into consideration some strings that can prove that this algorithm can solve trickier problems than the Rabin-Karp algorithm more efficiently. The other set, will focus solely on the analysis of two edit distances based on the Levenshtein distance and will predominantly be used for computing the minimal edit distance cost, essentially proving its linear efficiency for string correction problems.

2 Analysis of the chosen algorithms

2.1 KNUTH-MORRIS-PRATT

This algorithm was developed in 1974 by Donald Knuth and Vaughan Pratt, and independently by James H. Morris, with a wide release in 1977. The algorithm was conceived by analyzing the naïve solution, which essentially tries to match a word W with a text T, starting from every available index inside of T and reaches a final complexity of O(N + M), where N is the length of the target text T, and M is the length of the word W. There is a problem with this version of the algorithm however. When it finds a mismatch, the algorithm moves to the next position in T and starts comparing the word W from the beginning. For this reason, an enhanced algorithm was developed that not only resolves this problem, but is still considered one of the most efficient algorithms in pattern searching. This version of the algorithm examines the existence of a pattern P within T on the basis that if it matches the mismatch still occurs. This allows the word to be examined with future iterations of characters from the text T without resetting W's index to 0.

Furthermore, in the preprocessing algorithm, implemented in the DetermineLPS function of my algorithm from the second phase of the homework, we determine the values in lps[ ]. In order to do that, we keep track of the longest common suffix value for the previous index. For this, we will initialize lps[0] and len with 0 and we will check for a match. If word[len] and word[i] do indeed match, len will be incremented by 1 and assigend to lps[i]. In case word[i] and word[len] do not match and len is 0, len will be updated to lps[len - 1]. Even though it looks like the algorithm has a square complexity, the time complexity is O(M). The reason for this is that each value i will cause j to increase by one. No matter how much we move j back, we will only be consuming the increased amount. Thus, the two nested while loops have a linear complexity.

The KMP Algorithm depends on the lps array. Suppose at some step we were on the i-th iteration inside of the text T and the j-th iteration inside the word W. What this means, is that at the moment we are able to match the prefix of length j from the word W. In case we find a mismatch, we need to determine the second-longest matching prefix of the word W, which is actuality is lps[j]. If a mismatch still occurs j will always be moved back to lps[j], until we either find a match, or j reaches -1.

Let us take as an example. Suppose the text T = aaaab and the word W = aab.

W		a	a	b
LPS	-1	0	1	0

Then let us take a look at the result, after applying the KMP search algorithm to the text T.

Step	**Value of j**	**Match**					**Longest Match**
		a	a	a	a	b	
$i = 0$	$j = 0$	a					$j = 1$
$i = 1$	$j = 1$	a	a				$j = 2$
$i = 2$	Mismatch! Move j one step: $j = 1$		a	a			$j = 2$
$i = 3$	Mismatch! Move j one step: $j = 1$			a	a		$j = 2$
$i = 4$	$j = 2$			a	a	b	$j = 3$

As shown above, in the third and fourth iterations the algorithm used the lps array correctly to set j to its current position, so the algorithm did not have to start the matching process of the word W from the beginning when a new mismtach was detected. In turn, this helped the algorithm tremendously to find the of the word W in the last iteration in a more and effcient way.

The complexity of the search algorithm is O(N), where N is the length of the text T.

The total complexity of the KMP algorithm is O(M + N), where N is the length of the text T and M is the length of the word W.

Advantages and disadvantages

The KMP algorithm is efficient in exact pattern searching and is primarily used when fast pattern matching is required. However, there is a drawback. For different patterns and types of text, KMP has to be applied several times, so that means that if we want to to match multiple patterns it is not as efficient as other string processing algorithms in that regard, such as Trie or Sufix Trees.

2.2 EDIT DISTANCE (LEVENSHTEIN DISTANCE)

In the general terms of computational linguistics and computer science, an "edit distance" is the amount of steps required to transform one string to another given one. These operations are quite diverse, with many different examples and implementations, so for the sake of simplicity and efficiency in testing, this paper will take into account the Levenshtein Distance, as it is the most known type of distance and people generally refer to it when talking about the edit distance.The Levenshtein Distance is a string metric that uses insertion, deletion and substitution as operations for measuring the actual difference between two strings.

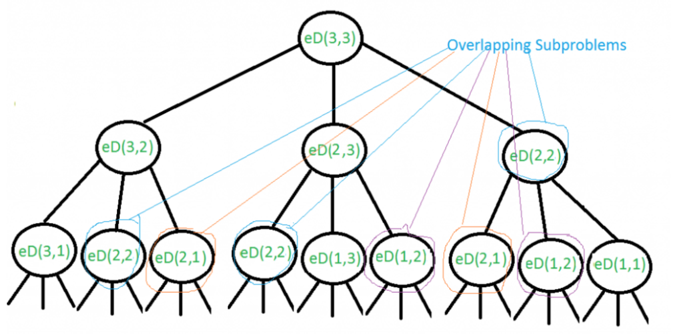
The Levenshetein distance between two strings a and b, given by lev(a, b).

$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}(\operatorname{tail}(a), \operatorname{tail}(b)) & \text{if } a[0] = b[0] \\ 1 + \min \begin{cases} \operatorname{lev}(\operatorname{tail}(a), b) \\ \operatorname{lev}(a, \operatorname{tail}(b)) \\ \operatorname{lev}(\operatorname{tail}(a), \operatorname{tail}(b)) \end{cases} & \text{otherwise.} \end{cases}$$

This definition is directly used in the naive recursive implementation.

The algorithm starts from traversing both strings from either left or right (in this analysis the indexing will start from the right). Let us consider two strings, string1, with a length of m and string 2, with a length of n. The first step is to check if the last characters from both strings match, if they do we simply decrement till a possible mismatch, at which point we apply all the operations defined in the edit distance to string1 and string2. The algorithm will recursively compute the insertion for both strings for m and n-1 characters, the deletion for m-1 and n characters and finally the substitution for m-1 and n-1 characters and return the minimum cost between all three of them.

The complexity of the above mentioned algorithm can be exponential. In the worst case, it can reach up to $O(3^m)$ and this happens if none of the characters of the two strings.



Worst case recursion tree when m = 3, n = 3.
Worst case example str1="abc" str2="xyz"

Above, is a recursive call of the worst case.

The time complexity of the enhanced algorithm is O(m $\times$ n), where m is the length of the first time string and n is the length of the second time series, or $O(n^2)$ if the two time strings are of the same lengths. The complexity is still not that favourable, however, we have to take into consideration that the Edit Distance is an universal distance that can be applied to all symbolic represented data objects, where other distance measures are not applicable.

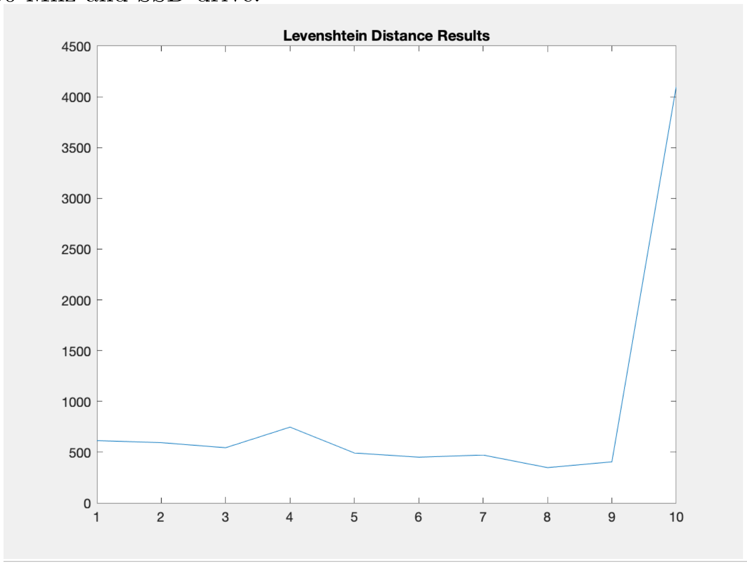
Advantages and disadvantages

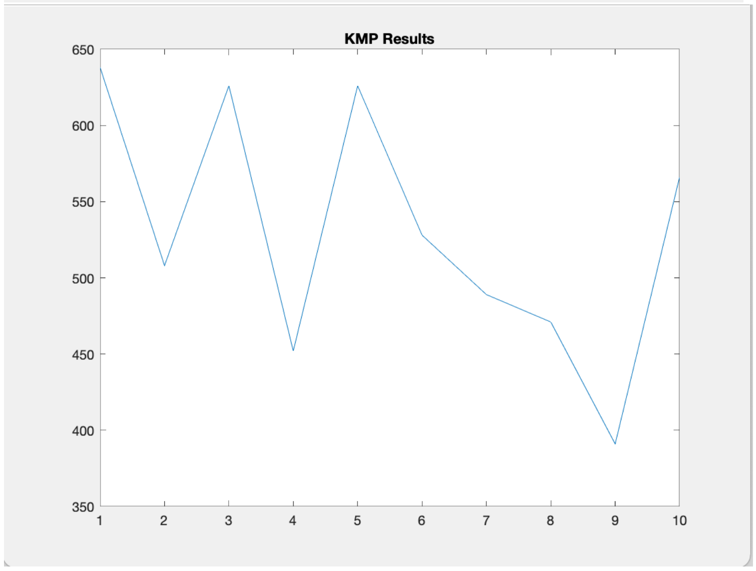
As mentioned in the introduction, the Edit Distance has many practical uses, ranging from pattern correction tools to engines and dictionaries, however, it has its drawbacks too. The main one is that it penalizes all change operations in the same way, without considering the character that is used in the change operation. This flaw is due to the fact that the edit distance is based on local procedures, both in the way it is defined and in the algorithms used to compute it. This can rise questions on the accuracy of the similarities obtained by applying this distance.

3 Performance evaluation

For the evaluation of the validity and practical performance of the algorithms, multiple tests were conducted that test both the KMP algorithm and the Edit

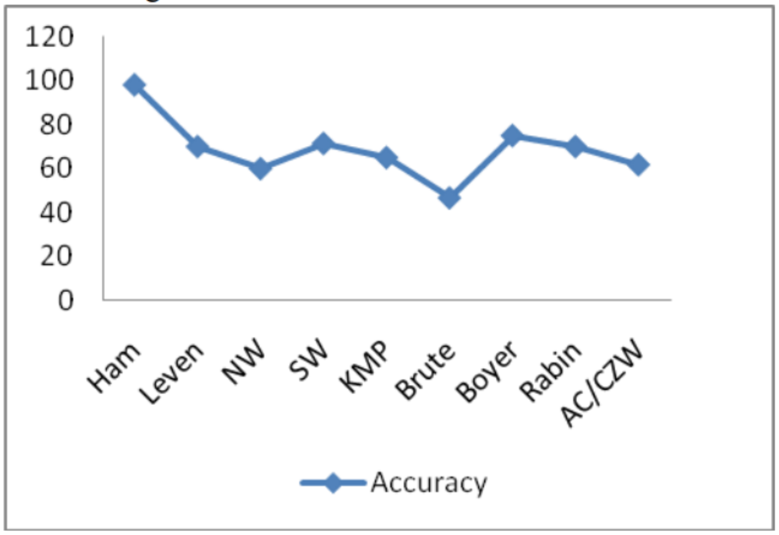
Distance, but in different scenarios, the ones that better fit their purpose and show their efficiency in said cases. The code has been compiled using the g++ compiler under Clang: Apple clang version 11.0.0 clang-1100.0.33.17 and run on a system with macOS 11.1 64-bit machine, Intel Core i7-4870HQ, 16GB DDR3 1600 Mhz and SSD drive.





As we can see from the graphs, especially for the Levenshtein Distance, the algorithm performs well under strings that are not exactly that different in length, however, the performance suffers a severe spike on the last test, when the length of the characters is larger than 150 characters.

As for KMP, we can see that it certainly has no problem keeping up with the tests, but importantly, we it more accurate, in fact it probably on of the best in terms of accuracy, as referenced in the chart below:



4 Conclusions

Taking into account the performance data and the theoretical analysis, we can fully state that both chosen topics are top class algorithms for their respective problems, one that can find a matching pattern in way that is both fast and accurate (KMP), and one that succeeds in applying an edit distance efficiently in computer science. (Edit Distance).

References

[1] Giovanni Pighizzini, “How Hard Is Computing the Edit Distance?”,

Available here:
`https://www.sciencedirect.com/science/article/pii/S0890540100929146/`

[2] Pandiselvam. P, Marimuthu. T, Lawrance. R, “A comparative study on string matching algorithms of biological sequences”,

Available here:
`https://arxiv.org/pdf/1401.7416.pdf`

[3] Muhammad Marwan Muhammad Fuad and Pierre-François Marteau, “The Extended Edit Distance Metric”,

Available here:
`https://arxiv.org/pdf/0709.4669.pdf`

[4] Mukku Bhagya Sri, Rachita Bhavsar, Preeti Naroota, “String Matching Algorithms”,

Available here:
`https://www.researchgate.net/publication/323988995`$_String_Matching_Algorithms$

[5] Geeks for Geeks,

Available here:
`https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/`
`Last time accessed on : December 17th 2020`

[6] Geeks for Geeks,

Available here:
`, https://www.geeksforgeeks.org/edit-distance-dp-5/ Last time accessed on: December 17th 2020`

[7] Baeldung,

Available here:
`https://www.baeldung.com/cs/knuth-morris-pratt Last time accessed on: December 17th 2020`

[8] iq.opengenus,

Available here:
`https://iq.opengenus.org/knuth-morris-pratt-algorithm/ Last time accessed on: December 17th 2020`