String Processing Algorithms

Catalin-Stefan Barus

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

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.

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.

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 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.

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.

1.5 References

Giovanni Pighizzini, "How Hard Is Computing the Edit Distance?"
Available here:
https://www.sciencedirect.com/science/article/pii/S0890540100929146/pdf?md5=52a5ca72cdefbc6272db9fb738de8d17&pid=1-s2.0-S0890540100929146-main.pdf

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

Mukku Bhagya Sri, Rachita Bhavsar, Preeti Naroota, "String Matching Algorithms"
Available here:
https://www.researchgate.net/publication/323988995_String_Matching_Algorithms

Geeks for Geeks, https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
Last time accessed on : November 12th 2020

Geeks for Geeks, https://www.geeksforgeeks.org/edit-distance-dp-5/
Last time accessed on: November 12th 2020

Baeldung, https://www.baeldung.com/cs/knuth-morris-pratt
Last time accessed on: November 12th 2020

Wikipedia,
https://en.wikipedia.org/wiki/Edit_distance#:~:text=In%20computational%20linguistics%20and%20computer,one%20string%20into%20the%20other.
Last time accessed on: November 12th 2020