

String Matching Algorithms

Although it may not be extremely obvious, string matching algorithms are a part of everyday life in the world we currently live in. Whenever a text/file comparison is performed, a piece of DNA is sequenced, a search engine is used, or a document is spell checked; string matching algorithms are being implemented. They are so well abstracted by the world around us that few realize their existence, let alone stop to think about how they even work.

Despite how straightforward it may seem at first, string matching can actually be a rather complicated endeavor. Upon the exploration of the topic, it becomes apparent rather quickly that there are many different types of these algorithms. Of course this does make sense, because just as in the other areas of computer science, there is an endless number of different methods that can be implemented to achieve the same goal. The two main types of string matching algorithms are those that are exact, and those that are approximate. The type that will be examined in this paper are types of exact algorithms. The problem that employs an exact string matching algorithm is: given a specific string or pattern, find one or all occurrences of that string or pattern in a specific text or longer string.

With most problems, there is a simple way of finding a solution, but it is usually not the most efficient time or space wise. The method usually applied in this simple case is referred to as a “brute-force” method, in that it merely exhausts every possible solution available until it reaches the correct one. In string matching, this would mean that the brute-force algorithm checks every single position in a text to see if the specified pattern being searched for begins there or not. It will start with the first character, and if the pattern does not begin there, it will shift one space to the right and try again. If searching in a short text or with a short pattern, the brute-force approach is not necessarily a bad one, and it would be hard to find an algorithm that could do a much better job. Brute-force quickly becomes inferior to other solutions when the text or search strings get longer though.

Fortunately, in 1975, one of (if not the most) efficient exact string matching algorithms was developed. Bob Boyer, a research mathematician at SRI International in Menlo Park, California, joined forces with J Strother Moore, a research mathematician for Xerox Palo Alto Research Center in Palo Alto, California. The two men began working on their idea in 1974 with a very simplified version of their final algorithm. It involved one preprocessing table rather than the eventual two. Strangely enough, Boyer and Moore learned after the fact that the idea they had been working on was also being developed at the same time by another computer scientist. Fortunately this other project did not gain quite as much momentum as Boyer and Moore, allowing them to continue evolution of their algorithm without much competition (Boyer and Moore).

Finally, after a year or so of further work, as well and input and advice from the likes of Donald Knuth and various others in the computer science realm, the result of Bob Boyer and J Strother Moore's collaboration was born as the Boyer-Moore string search algorithm. The algorithm's main claim to fame is its very unique quality of searching for matches from right to left, rather than the usual left to

right method. The merits of this approach will become more apparent once of the workings of it are further examined in detail.

The Boyer-Moore string search algorithm first preprocesses the string that is being searched for and the text in which it is being searched for. This preprocessing involves creating two tables based on the string's characters, and the data produced in the tables will later dictate the shifts that are performed in the search stage. Although this obviously takes up some time, the difference it makes on the actual searching time is drastic, and very necessary. The efficiency of the algorithm ends up relying on the information acquired from unsuccessful matching attempts to weed out the places where the string is *not* going to match.

The first table possesses an entry for every character in the alphabet of the string and text. The entry for each character will be the length of the string if that character does not appear in the string. If the character does appear in the string, its entry will be the length of the string minus the index of the left-most occurrence of that character in the string.

The second table is much more complicated to create. To begin, for each value of a number, i , that is less than the string's length, a pattern is calculated that is made up of the last i characters of the string, preceded by any character that would produce a mis-match if it were the character before it. The next step is to, first, line it up with the larger text and find the smallest number of characters that the partial pattern needs to be shifted left before it matches the other pattern. It is not necessary to completely understand how this table works to understand the algorithm, and the examples below will clarify the workings if any confusion is present.

The way the algorithm actually operates is easier to understand when visualized, so various examples will be used to aid in comprehension. The first step is to line the first letter of the string up with the first letter of the text. The last letter of the string is then compared to the letter of the text it matches up with. In the example below (Moore), you can see that the E and S would be compared with one another.

EXAMPLE

THIS IS A SIMPLE EXAMPLE

This, obviously, is not a match. And, since there is no S at all in the string, it can be moved down by its length and the possibility of skipping a potential match will not be a problem. This specific shift example is a good demonstration of the best-case performance of the algorithm, which with a text of length N and a string of length M will be N/M . Out of every M characters, only one comparison is going to be made. Furthermore, this example shows one of the Boyer-Moore algorithm's unique characteristics: Generally speaking, the longer the string being searched for, the faster the algorithm works.

Returning to the example though, a tip can be made to help visualize further shifts. Moore himself proposes this when discussing the algorithm on his own website. It can be imagined that the S that was just found not to be a match is simply placed before the string. Since the algorithm always lines up the left-most occurrence of a character in the string with the same letter from the text, the two strings now line up like this, with the S's lined up:

SEXAMPL E

THIS IS A S I M P L E E X A M P L E

The imaginary S can now be disregarded, and the E in the string and the P in the text will be compared. This is not a match, but since there is a P in the string, the string can be slid down to the left-most (and in this case, only occurrence of) P so that the two characters are lined up.

EXAMPLE

THIS IS A S I M P L E E X A M P L E

Starting from the right, characters are then compared. The E, L, P, and M are all found to be matches. But, when the comparison between A and I is reached, a choice has to be made. Just as before, the I could be placed before the string and lined up with its match in the text, like this:

I EXAMPLE

THIS IS A SIMPL E E X A M P L E

or, since it has already been deduced that the only occurrence of MPLE in the string does not line up correctly with the text, MPL (not the E, since it already is the first letter of the string) could be put in front and consequently lined up like this:

M P L E X A M P L E

THIS IS A SIMPLE E X A M P L E

And, as Moore himself explains, “In general the algorithm always has a choice of two shifts it could make and it takes the larger of the two” (Moore). So, since the latter example produces a larger shift, this is the one that will be implemented. The next step will be to compare the E in the string with the P in the text. As shown before, since there is a P in the string, the left-most occurrence of P in the string will be lined up with the P in the text. Remember that the P in the MPL placed before the string is only imaginary, and will not be the P used in the shift. Now that the P's are lined up, comparisons will be carried out starting from the right hand side. Ultimately, it will be concluded that a match does indeed exist between the string and the text.

After learning about the simple brute-force method and the rather complex Boyer-Moore algorithm, the difference in performance between the two drastically different options might be a question at the tip of one's tongue.

With regards to the brute-force method, one perk is that there is no preprocessing phase. Also, comparisons can be done in any order, since it is inevitable that every character will be compared. In the [Handbook of Exact String Matching Algorithms](#), authors Christian Charras and Thierry Lecroq, well known computer scientists and authors of many other pieces of string algorithm literature, perform extensive examinations of numerous different string matching algorithms. For their implementation of a simple brute-force algorithm, the search phase is found to be in $O(mn)$ time complexity with an average number of comparisons being $2n$. One major downside of this algorithm is only being able to shift by 1 space to the right each time.

With Boyer-Moore, preprocessing does have to be done, and this phase is in $O(m+\sigma)$ time and space complexity, with the search phase in $O(mn)$ time complexity (Charras and Lecroq). In a worst case scenario, the Boyer-Moore will perform with a time complexity of $O(m+n)$, but as stated before, it

has a best performance of $O(m/n)$, and can have a sub-linear execution time since it does not always have to check every letter in the text. As with numerous other well designed algorithms, the Boyer-Moore will work best on certain kinds of data, and not so well on other kinds. It is best to know what kind of data is being dealt with in order to choose the most efficient way of dealing with it. For instance, if DNA data were being dealt with, the Boyer-Moore would not prove to be the most efficient. As it has been shown, the Boyer-Moore works best on text with a larger alphabet, and larger text and strings. DNA data would include a rather limited alphabet, and therefore would not provide the Boyer-Moore algorithm with the ability to shift very large distances.

All in all, it can be agreed that text and text processing play extremely significant roles in the world today. An immense amount of data is stored in text files, and it is extremely necessary for this information to be easily accessible to the people working on it. Any new idea or implementation that makes this more efficient is unquestionably important and would have a profound effect on text processing as a whole. String matching algorithms, and the Boyer-Moore algorithm in particular, have done just that, and should be acknowledged for the ways in which they have made so many operations more effective.

Works Cited

Boyer, Bob and J Strother Moore. "A Fast String Searching Algorithm." Communications of the Association for Computing Machinery 20 (1977): 762-772.

Charras, Christian and Thierry Lecroq. "Handbook of Exact String Matchign Algorithms." 14 Jan. 1997. Laboratoire D'Informatique De Rouen. 17 Apr. 2008 <<http://www-igm.univ-mlv.fr/~lecroq/string/>>.

Strother Moore, J. "The Boyer-Moore Fast String Searching Algorithm." Dept. of Computer Science, University of Texas At Austin. 15 Apr. 2008 <<http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/index.html>>.