Evaluation of String Searching Algorithms
Simon Wahlström
Mälardalen University, Västerås, Sweden
Papegojvägen 10
72220 Västerås
0730606495
Svm09001@student.mdh.se

ABSTRACT
How does the computer find strings when the user searches for them? How is it possible to find several hundred thousands results when searching for "Obama" on Google? The answer is string searching algorithms or string pattern matching. String searching algorithms are designed to find one or more specific patterns in a larger text. Without them, we would not have any search aggregators and we would not be able to use spell check on our reports. It would be almost impossible to find a specific string in a larger text, like finding a needle in a haystack. This is going to be almost impossible to do. For larger text, like finding a needle in a haystack.

In this paper, an evaluation of five string searching algorithms will be presented; Brute Force, Boyer-Moore, Knuth-Morris-Pratt, Karp-Rabin and the Horspool algorithm. How they work, when they work and when they are best suited for a particular problem will be explained.

Categories and Subject Descriptors
I.1.2 [Algorithms]: Analysis of Algorithms

General Terms
Algorithms, Documentation, Performance

Keywords
String Searching Algorithms; String Pattern Matching; Pattern Search Algorithms

1. INTRODUCTION
String searching algorithms are being used every time we use our computers. They help us find our files, search for strings on search aggregators and correct our misspelled words. They are an important class of string algorithms that try to find a place where one or several strings (called pattern) exists in a larger string or in a text [11].

For larger texts, string searching can be almost impossible for a human. Imagine looking for a certain name or phrase in the bible, or finding a word in an unsorted dictionary. That is going to require a great deal of time and may even seem impossible. This is where computers and string searching algorithms come in. They make what seems to be impossible for a human seem trivial.

There are many different algorithms for string searching or string matching as it is also called. This report will cover some of the basic algorithms as well as some of the more advanced. There will be a general evaluation between the algorithms and some explanation on how and when they should be used.

When talking about string searching, the first thing to come to mind is often search aggregators, e.g., Google. They search through an almost endless sea of information to get the result based on the pattern the user have specified. The web is made out of almost 60 trillion individual webpages, and is growing every day. To make it possible for search aggregators to find all this information they need to use string searching algorithms.

There are several different types of string searching algorithms. These are the naïve search, which is the simplest and least efficient way to check if a string exits within another string. The naïve search uses brute force to find a pattern in a text. The hashing algorithms, which uses a technique called hashing to find a pattern in a text. This means that the data is mapped and then used to find a match. Another type is the search algorithms that use index methods. They are based on preprocessing text and finding a string with the help of a substring index e.g., a suffix tree to find the string faster. There are also the Finite set of pattern based search algorithms, which use only use a finite set of patterns to search for a string. [11].

The reason to why there is so many algorithms is simple. We have a lot of different kinds of strings that needs to be searched. Let Σ be an alphabet (finite set). The alphabet can be the characters of the English language. It can also be both lower-case and capital characters, or it may be the binary alphabet (Σ = {0, 1}), and do not forget about the bioinformatics alphabet (Σ = {A, C, G, T}). Therefore, depending on how the string is encoded it can affect the executable algorithm. One example is if there is a variable width encoding in use. Then it will be very slow (time proportional to N) if we look for the Nth character. Even the most advanced string search algorithms will be significantly slowed down by this. A possible solution to this problem may be to search for the sequence of code units instead, but there can still be false matches unless the encoding is designed for that type of algorithms.

The algorithms that will be explained are the following: The Brute force algorithm, which is considered the most basic string searching algorithms that exists. The Knuth-Morris-Pratt algorithm which was published in 1977 and is still used today. The Boyer-Moore algorithm, which is a very effective string searching algorithm and has been used as a benchmark for practical string search literature [9]. The Karp-Rabin algorithm,
which was presented in 1987 and uses a form of hashing to complete the pattern matching. Finally, the Horrobin algorithm is presented which is a modification of the Boyer-Moore algorithm.

2. PROBLEM DEFINITION

Searching for strings is not always an easy task. More data equals more strings to search through. There are several fundamental problems in the area of string searching algorithms. They can be theoretical considerations or practical problems, the latter primarily coming from bioinformatics. Usually, the string searching algorithms main task is to find a pattern to be searched for in a larger object, like text. There are often several different conditions that need to be fulfilled to get the desired search. Both the pattern and the text are in principle arrays of symbols. These symbols are typically parts of different finite alphabets \( \Sigma \), e.g., \( \{0, 1\}, \{a \ldots z\}, \{C, G, T, A\} \) and the ASCII table.

In this section, a pattern will be denoted as \( P = P[1 \ldots m] \) and the text will be \( T = T[1 \ldots n] \) where \( m \) and \( n \) are their lengths. The number of characters in \( \Sigma \) will be denoted as \( d \) and a finite alphabet will be denoted as \( a \).

One of the fundamental problems that exist in the string pattern recognition is the string-searching existence problem which is called SME. It can be stated as follows: If there is a pattern \( P \) and a text \( T \), ascertain if there is an occurrence of \( P \) in \( T \). The immediate generalization of that problem is the string-searching computation problem, called SMC. The SMC problem is namely to report how many times the pattern \( P \) occurs in the text \( T \). When talking about string matching problems it is generally the SMC problem that is referred to if not stated otherwise [2].

To demonstrate this, we will work with alphabet \( a = \{a, b, c\} \). We want to know if the pattern \( P = \{abbac\} \) occurs in the text \( T = \{abbbabacabbacabhbabcabcabc\} \). Without the boldface it would be very tedious to see if \( P \) exits in \( T \) or not with just eye-inspection. As we can see, \( P \) occurs on both position 9 and 17. Therefore, the answer to the SME problem is yes and if we continue our search then the answer to the SMC problem is position 9 and 17 [2].

It is however not always this "easy" to work with string pattern recognition. It is often a lot bigger patterns and even bigger texts to be searched through. If we were to search for "World's greatest teacher: Gordana" on Google, we would get approximate 169 000 results in about 0, 18 seconds. This is a good example of how string pattern matching together with other algorithmic techniques is being used on a very high level.

In the domain of string processing, string searching is a very important subject. They are a basic component used in implementations of practical software that exists under most of the operating systems we have today. String searching algorithms have an important role in theoretical computer science since they provide challenging problems. The algorithms also punctuate methods that serve as paradigms in other fields of computer science e.g., system or system design [1].

3. RELATED WORK

The paper “String Matching: First Algorithms” by Paco Gómez [2] describes what string matching is and what problems exist in the area. The paper gives a clear view of string matching in general and proves several easy to understand examples of how pattern matching is being used. Two of the most famous string searching algorithms, the Rabin-Karp and the Naïve String-Matching algorithm are explained and proven to work. Since this paper is also explaining string pattern matching, this report will be very useful.

Christian Charras and Thierry Lecroq have written an extensive report called “Exact String Matching Algorithms” [1] on several algorithms. They have declared advantages and disadvantages on all the algorithms. They have also explained all the algorithms in a pedagogical way which make them easy to understand. For every algorithm, they have also provided code-examples so that the algorithms can be tested in Java. This report contains a lot of valuable knowledge, which can be highly usable in this paper.

The paper that Petteri Jokinen, Jorma Tharhio and Esko Ukkonen wrote, “A Comparison of Approximate String Matching Algorithms” [4], is similar to this report. The paper is about experimental comparison of the running time if approximate string matching algorithms. The experiment is conducted on seven different algorithms. The result is presented and explained and according to the paper there is no "The best algorithm". The result of this paper is very useful when doing a general evaluation between string searching algorithms.

Andrew Hume and Daniel Sunday have written a report on “Fast String Searching” [3]. The paper focuses on improving the Boyer-Moore algorithm. They describe two algorithms that should perform 4.5 times faster than the original algorithm (in some ranges and compilers). It is also explained where these algorithms comes to good use. This is quite relevant since the algorithms they used will be looked at in this paper as well.

4. METHOD

The method used to evaluate different string searching algorithms in a general view was to study as many of the different algorithms as possible. Google and Wikipedia have been the primary tools for finding information. The search terms were a combination between "String searching algorithms", "String matching algorithms" and "String pattern matching". The main goal is to find as much information about the different algorithms as possible and try to explain them.

5. RESULTS

There was a lot of information to be processed in the field of String searching algorithms. It has been a lot of research done on how the algorithms work and how they are best put to use. In this section, five algorithms will be presented. The results will include why an algorithm is chosen, some of the execution times and the advantages and disadvantages of the algorithms. Finally, a general overview of the algorithms will be presented.

When talking about string pattern matching algorithms, there are several terms that need to be explained to understand the whole process. Most algorithms have both a preprocessing phase followed by a searching phase. In the preprocessing phase the algorithm often conducts some of the work before doing the actual search, e.g., dividing the text to be searched into different windows. These windows are used to determine if it is even possible to find a certain pattern within the current area. The
The algorithms that will be presented are the Brute Force algorithm, the Knuth-Morris-Pratt algorithm, Boyer-Moore algorithm, the Karp-Rabin algorithm and the Horspool algorithm. They are some of the most common algorithms used when searching for strings in texts. Some of them are still used today and some are considered ancient but play a big role in the field of string pattern matching.

5.1 Brute force algorithm

The brute force algorithm is probably one of the simplest algorithms for string searching there is. It is categorized as a naïve algorithm. The algorithm checks all the positions in the text between 0 and \( [m .. n] \) to see if the pattern starts there or not. After checking a position, the algorithm shifts one step to the right.

There is no preprocessing time phase, and a constant extra space in addition to the pattern and the text. During the searching phase, character comparison can be done in any desired order. The expected amount of text character comparisons is \( 2n \). The searching phase is \( O(nm) \) [1].

The biggest advantage this algorithm has is that it is very easy to implement. It is however one of the most primitive algorithms and can be very slow compared to other algorithms.

5.2 Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm originally comes from the Morris-Pratt algorithm which is very similar. Since the Knuth-Morris-Pratt algorithm is considered to be better and more updated the Morris-Pratt algorithm has been discarded.

The Morris-Pratt algorithm reminds a bit of the brute force algorithm, except that is has a preprocessing phase. Another difference is that this algorithms is also an index search type algorithm. If we look at the brute force algorithm, it is possible to improve the length of the shifts and at the same time remember parts of the text that match the pattern. This increases the speed of the algorithm since it saves comparisons between characters of the text and characters of the pattern. The Knuth-Morris-Pratt algorithm uses patterns to pre-compute the number of skips and then searches in the same way as the brute force algorithm. If a mismatch is found KMP uses the pre-computed numbers of shifts to determine how far to shift [6].

The algorithm follow these 5 steps [5]:

1. \( N = \text{Length of the text.} \)
   \( M = \text{length of the pattern.} \)
   \( P = \text{prefix function of pattern.} \)
   \( Q = \text{number of characters matched.} \)

2. Define the variable: \( q = 0 \), the beginning of the match.

3. Compare the first character of the pattern with first character of text.
   If match is not found, substitute the value of \( P[q] \) to \( q \).
   If match is found, then increment the value of \( q \) by 1.

4. Check whether all the pattern elements are matched with the text elements.
   If not, repeat the search process.
   If yes, print the number of shifts taken by the pattern.

5. Look for next match

The computation of the first phase will take \( O(n) \) time and the searching phase will take \( O(n) \) time.

Some of Knuth-Morris-Pratt strengths is that the running time is optimal \( O(n+m) \), and that the algorithm never has to move backwards in the text that is being searched. This makes it ideal for processing very large files.

The strongest weakness that Knuth-Morris-Pratt has is that it does not work well as the size of the alphabet increases. This will lead to a higher chance of a mismatch.

5.3 The Boyer-Moore Algorithm

One of the most classic string searching algorithms is the Boyer-Moore Algorithm. It has been considered as a benchmark algorithm for a long time. It is said to be very efficient when working with practical string search literature [3]. This algorithm processes the pattern but not the actual text. This makes it well suited if the pattern is much smaller than the text. The algorithm gathers information during the preprocessing phase which it uses to skip sections of the text it is searching in. If the pattern length increases, the execution time decreases [7]. Just like the Knuth-Morris-Pratt algorithm, the Boyer-Moore is also categorized as an index search algorithm.

The algorithm contains three special parts to carry out the task. This is the Good suffix rule, the bad character rule and the right to left scan.

The right to left scan works in the following way: For any alignment of the pattern with the text, the Boyer-Moore algorithm scans for an occurrence of the pattern by scanning its character from right to left. If there is a mismatch, then the shift is decided based upon the two other rules.

The bad character rule works in the following way: Assume that a mismatch is going to occur between the character \( p[i]=a \) of the pattern and the character \( t[i+j]=b \) of the text during an attempt at position \( j \). Then, \( p[i+1 .. m] = t[i+j+1 .. j+m] = u \) and \( p[i] \neq t[i+j] \). If \( b \) is not contained anywhere in \( p \), then shift the pattern \( p \) completely past text \( [i+] \).
5.4 The Karp-Rabin algorithm

Karp-Rabin is a string searching algorithm that conducts its searches with help of hashing. Hashing is when an algorithm maps data of variable length to data of a fixed length [8]. It is also categorized as a finite set of pattern search algorithm. Unlike the brute force algorithm, Karp-Rabin does not check every position of the text if the pattern occurs. Instead it checks if the content of the "window" resembles the pattern. To do this resemble check, the hashing function is used.

If the string matching search with hashing is going to be helpful, the hashing function hash should fulfill some certain properties. It should be efficiently computable and highly discriminating for strings.

If we have a pattern \( p \) of length \( L \), let \( \text{hash}(h) \) be defined as follows:

\[
\text{hash}(p[0 .. m-1]) = (p[0]*2^{p[1]} + w[1]*2^{p[2]} + ... + p[m-1]*2^0) \mod q
\]

where \( q \) is a large number.

Finally, \( \text{rehash} (a, b, h) = ((h-a*2^{p[1]})+2*b) \mod q \).

The expected number of character comparisons of the Karp-Rabin algorithm is \( O(n+m) \). The biggest advantages of this algorithm is that in most cases the time complexity can be as low as \( O(n) \). One of the disadvantages of this algorithm is that in the worst-case scenario, the runtime would be \( O(n - m + 1)m) \).

5.5 Horspool algorithm

This Algorithm is a simplification of the Boyer-Moore algorithm. As described earlier, the Boyer-More algorithm uses the bad character rule. This rule is not very efficient when working with small alphabets compared to the length if the pattern. Therefore, when working with the ASCII table and ordinary searches in a text editor the bad shift rule works a lot better. Using this rule on its own can make a very efficient algorithm. Horspool (the name of the person coming up with this idea) proposed to only use the bad shift rule of the rightmost character of the window to compute the shifts in the Boyer-Moore algorithm [1].

The preprocessing phase of the Horspool algorithm is \( O(n+m\Sigma) \) and the searching phase has an average of \( 1/\Sigma \) and \( 2(\Sigma+1) \) (\( \Sigma \) being the alphabet).

5.6 End results

There are still a lot more of algorithms that has not been brought up due to the complexity of the algorithms. This paper has covered five of the most influential algorithms that we have in modern history. As we have seen they have some parts in common but are better suited for different tasks. You should choose the algorithm based on five things. How long the pattern is, how long the text is, the alphabet, if you expect to find the pattern or not and on the problem at hand.

If both the text and the pattern are going to be relative short, then it is probably easier to use the brute force algorithm since it is so small and easy to implement. If you are looking for a pattern in a very large text with the common alphabet, then the Knuth-Morris-Pratt algorithm may be of good use. In case the pattern matching is done on a very large alphabet or if it is a binary string, then the Boyer-Moore algorithm would probably be the best one suited for the task. When looking for multiple different patterns in a text, then you should probably use the Karp-Rabin algorithm. Finally,
if you are looking for string with a very large alphabet like the ASCII table or looking for strings in a text editor, then the Horspool algorithm is recommended.

Some algorithms can be improved by simply analyze the problem before the algorithm is set to work. One example is the Brute force algorithm. It may still possible to speed up the brute force algorithm even more without changing it. If we reduce the search space which is the set of candidate solutions with help of heuristics specific to the problem. To demonstrate this, the eight queens’ problem will be used. The eight queens problem consists of placing eight queens on a chessboard so there can be no queen attacking the other queens. There are 64 squares to place the queens on so in principle there is 64⁸ solutions. This equals to 281,474,976,710,656 possibilities to consider. Now, since all the queens are all alike and there can be no more than one queen placed on the same square, we can chose of a set of eight squares from the set of 64 squares. This will give us 4,426,165,386 possible solutions (64 choose 8 = 64!/56!/8!). This is almost 1/60,000 of the previous estimation if possibilities. By analyzing the problem we can gain a dramatic reduction in the possible solutions to the problem and therefore be able to use the algorithm more efficiently [8].

6. SUMMARY AND CONCLUSIONS
In this paper five of the most common string searching algorithms has been discussed. Every algorithm have been provided with an explanation of the semantics on how the algorithms work. The main limitations of this research have been the complexity of the algorithms. It can be very difficult to explain an algorithm without going in too deep into each algorithm. The paper has also explained why we need search algorithms and given examples of when to use them.

The algorithms that have been researched and explained is the Brute force, Knuth-Morris-Pratt, Boyer-Moore, Karp-Rabin and the Horspool algorithm. They all have their unique weaknesses and strengths. To choose the right algorithm one must first know what it is going to be used upon. It is also important to analyze the problem before to see if it is possible to make the algorithm come to better use.

As described in the methods section, the choice of algorithm depends on five things. The length of the pattern, the type of the alphabet, the length of the text that is being searched, if we expect to find the pattern and on what problem the algorithm is going to solve. For example, if the problem consists of finding a small pattern in a small alphabet, then the Brute Force algorithm would be a good choice since it is so easy to implement and works well for this kind of problem.

For future work more algorithms should be added and explained in a wider sense to give a better understanding of how they really work. There were several algorithms that would have been interesting to include in the evaluation but due to the time limit they had to be excluded. Some of these algorithms was the Colussi algorithm and the Shift-Or algorithm. These algorithms include a lot of mathematics (which most of the algorithms do) which I consider to be too advanced for this general evaluation. They do however still play an important role in the field of string searching algorithms.

The experiences from this work have been very positive. Knowing when to choose an algorithm for certain problems can reduce the overall problem and save a lot of time. It is good to have insight in string searching algorithms when working in the field of software engineering since string searching algorithms are being used for so many things. We use string searching algorithms every day when using a computer without even knowing that they are there. If we in the future want to be able to find files faster or get even better results from search aggregators, then we need to improve the string searching algorithms that we have today. Faster algorithms equal faster results, which mean that we can do our work faster. The overall experience when working with a computer would then be faster since there is such a big part that is constantly looking for strings.

In a world without string searching algorithms, this report would probably be a lot worse. It would not be possible to find all the misspelled words or find different phrases based upon a certain pattern. I would also have had to resort to read actual books since finding information on the internet would be impossible, I would also have had to mark all the important parts in the books and remember them. If I wanted to see if a book contained a certain word or pattern, I would be forced to do it manually and spend an enormous amount of time doing it.

7. REFERENCES