

Ute Abel
Proseminar "Algorithmen"
Prof. Brandenburg
Sommersemester 2001

String matching

TABLE OF CONTENTS:

| | | |
|-----|---|----|
| I | Introduction | 3 |
| II | Algorithms and methods to solve the string matching problem | 03 |
| | 1.1 The Brute-Force-Algorithm | 03 |
| | 1.2 String matching with finite automata | 04 |
| | 1.3 The Knuth-Morris-Pratt Algorithm | 05 |
| | 1.4 The Boyer-Moore-Algorithm | 08 |
| | 2.1 Other algorithms and improvements | 10 |
| III | Bibliography | 11 |

I. Introduction

Operations on strings often make it necessary to decide if and where a certain substring is contained in a sequence of characters. In more formal terms: Given a text T of length N over an alphabet Σ^* and a pattern P of length M, is there an occurrence of P in T? This problem crops up very frequently in text editing processes, during information retrieval, in recent times also in the analysis of genetic sequences, and in data processing, where you mostly operate on binary strings.

In the following I'm going to present some of the best-known algorithms and methods that have been developed to solve the string matching problem; the four main topics I will discuss include the Brute-Force-Algorithm, string matching with finite automata, the Knuth-Morris-Pratt-Algorithm, which is quite similar to the finite automaton matcher, and the Boyer-Moore-algorithm, perhaps the most sophisticated one.

II Algorithms and methods to solve the string matching problem

1.1 The Brute-Force-Algorithm

The most obvious approach to the string matching problem is the Brute-Force-Algorithm, which is also called the "naive" algorithm. It requires no pre-processing on pattern or text.

The idea: pattern and text are compared character by character; in case of a mismatch, the pattern is shifted one position to the right and the comparison is repeated, until a match is found or the end of the text is reached.

The following code fragment is a Java implementation of the Brute-Force Algorithm

```
public static int BruteForcematch (String text, String pattern) {for (int i = 0; i < n-m; i++) {  
    int j = 0;  
    while ( (j<m) && (T[i+j] == P[j]) )  
        j++; if (j == m)  
            return i;  
}  
return -1;
```

The algorithm works with two pointers; a "text pointer" i and a "pattern pointer" j.

For all (N-M) possibly valid shifts, pattern and text are compared;

while text and pattern characters are equal, the pattern pointer is incremented.

If a mismatch occurs, i is incremented, j is reset to zero and the

comparing process is restarted. In case a match is found, the algorithm returns the position of the valid shift; if not, it returns - 1.

The running time of the Brute-Force-algorithm is $O(n \cdot m)$ in the worst case; this would happen if it is used on e.g. a sequence of n 0's followed by a 1 to search for a pattern of m 0's and a 1 (assume $m < n$).

In this case, up to m characters would have to be compared for all $n-m$ examined shifts, which sums up to the mentioned worst case complexity of roughly $O(n \cdot m)$.

Although this easy pattern matching algorithm is still widely used, and although many of the more complicated algorithms are not much faster for pattern matching on standard English text, the disadvantages of the Brute Force-algorithm are obvious: It is a rather inefficient method for binary text strings or other text strings on small alphabets, uses many small shifts some of which appear to be negligible, and the text pointer i has to be "backed up" each time a mismatch occurs, which means that the input text has to be buffered.

1.2 String matching with finite automata

String matching with finite automata is a more efficient approach to the problem. As already indicated by the name, this method uses a finite automaton to scan for occurrences of the pattern in the text.

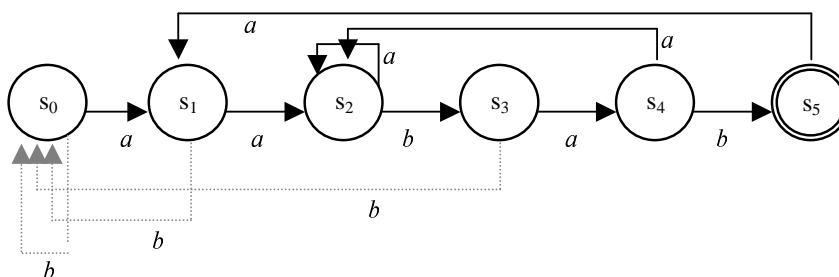
Although most of the participants of the Proseminar should have heard about finite automata in the Technical Computer Science lecture, I will start by repeating the definition.

A finite automaton is a 5-tuple $(S, s_0, A, \Sigma, \delta)$, where

- S is a finite set of states
- s_0 is the start state
- $A \subseteq S$ is a distinguished set of accepting states
- Σ^* is a finite input alphabet
- δ is a function from $S \times \Sigma^*$ into S , called the transition function of the automaton

In order to use finite automata for the string matching problem, the machine has to be built according to the pattern P ; the resulting automaton will have $(m+1)$ states of which the last one is the only accepting state.

Let's have a look at the construction of the finite state automaton for the pattern "aabab"



A short example with a small alphabet of only two characters; in the first step, we construct the "spine" of the automaton with the transition steps as they will be executed in case of a match; then we add the directed edges for the mismatch cases.

To compute the transition function for the latter ones, we can make use of this formula which determines the longest suffix of the "false start" which is also a prefix of the pattern P:

$$\begin{cases} \delta(i,a) = \max \{k \leq i \mid P[1..k] \text{ is suffix of } P[1..i]a\} & \text{if maximum exists} \\ \delta(i,a) = 0 & \text{if no suffix can be found} \end{cases}$$

To put it in a nutshell: the pattern is checked for self-repetitive parts; in some mismatch cases, the comparison cannot be restarted from state s_0 , as the last characters of the present shift position could also be a part of another occurrence of the pattern in the text. In our example, this would be the case for all directed edges on the upper part of the illustration.

In all other cases, the machine is reset to the start state.

When we use this machine now to scan through the text, each character will only be examined once, and this is where the complicated pre-processing pays off:

the search will be executed in a running time of approximately $O(n)$.

We won't analyse string matching algorithms which simulate the behaviour of finite automata in greater detail, as we'll have a much closer look at the Knuth-Morris-Pratt-algorithm, which is quite similar.

1.3 The Knuth-Morris-Pratt-Algorithm

This algorithm was found by Knuth & Pratt and independently also by Morris in 1976; they all were inspired by Cook's theorem that there could be a method to solve the string matching problem with a runtime of $O(n+m)$ in the worst case.

When analysing the Brute-Force-Algorithm, we judged it to be ineffective, because we had to backup the text pointer after each mismatch; i.e., some text characters were examined twice or more, although, by the information about the pattern we had, we knew that several shifts could have been skipped. Using finite automata helped us to eliminate some of these needless shifts, but the pre-processing, especially the computation of the transition function, can still be improved.

The Knuth-Morris-Pratt-algorithm works much like the finite automaton matcher: pattern and text characters are compared in a left-to-right scan. If a mismatch occurs, the algorithm searches for the largest suffix of the "false start" that is also a prefix of the pattern and thereby determines how far the pattern can be shifted to the right without missing a possible match.

The data we need to find the next shifting position is stored in an auxiliary "next" – table which is computed in a pre-processing step by comparing the pattern with itself;

the entries contain the information which character of the pattern should be compared to the text character in the "mismatch"-position next. This "next"-table is superior to the helper function we used for the finite automaton matcher; the old two-dimensional table with $m+1$ can be replaced by a one-dimensional table with only m entries.

Here is a short description of the algorithm we use to compute the next-table:

We slide a template of the pattern along P and search for the largest prefix of P which is a suffix of $P[1, \dots, j]$. We compare the pattern with itself at each possible shift position; while the characters match,

both the pointers are incremented; when a mismatch happens after more than one comparison, we compute $\text{next}[j]$ of $j-1$; if the mismatch occurs already in the first position, $\text{next}[j]$ is set to zero, i is incremented, and we check the next shift of the pattern against itself.

The main algorithm shows nearly the same structure:

```

while (i < n) {
    if (pattern.charAt(j) == text.charAt(i)) {
        if (j == m - 1)
            return i - m + 1;           // match
        j++;
    } else if (j > 0) {
        j = fail[j - 1];
    } else {
        i++;
    }
    return -1; }                       // no match

```

As long as we haven't reached the end of the text, pattern and text are compared.

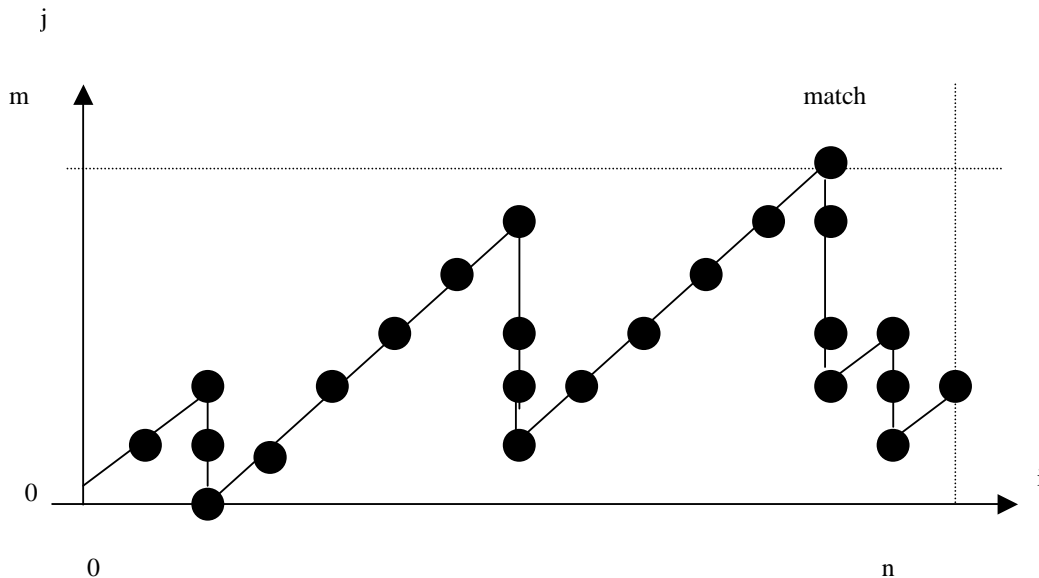
While pattern and text characters match, i and j are incremented; when the match in full is found, the algorithm returns the position of the valid shift. For the mismatch cases, there is a case distinction:

If the mismatch occurs already in the first position, the pattern is shifted one position to the right and the comparison is restarted; if not, the algorithm calls the helper function to determine the next possibly valid shift.

Should the end of the text be reached before we found a match, the algorithm will return -1 .

At the beginning of this chapter, I claimed that the Knuth-Morris-Pratt has a linear worst case boundary; still, we just saw that there are two nested loops in the algorithm which could theoretically speaking be executed up to n and up to m times, and this would make a worst case complexity of $O(n \cdot m)$. Now I'm going to give you an idea of why this is not so.

Below is a graphical illustration of the way j is typically modified during the execution of the KMP-algorithm.



Whenever text and pattern character match, j is incremented; the points on the downward leading lines signify the mismatch cases, when j is decremented according to the "next"-table. The crucial thing you should recognise from this graph is; j can only be decremented as many times as it was incremented before, there can be only as many points on the falling lines as on the rising lines, which means the total number of points is limited by $2n$. For the algorithm, this signifies that we have a runtime of $O(2n) \approx O(n)$ in the worst case, not considering the auxiliary function. There is a similar graph for the pre-processing step, and so, we would have an overall runtime of $O(2n) + O(2m) \approx O(n+m)$ in the worst case.

This linear worst case boundary is one of the major advantages of the KMP algorithm; another good reason to use this algorithm is that, like the finite automaton matcher, it doesn't require buffering of the scanned text. How important this can prove is best seen in the example of a data transfer from the Internet; not having to store the data before you scanned it for possible viruses, for example, is obviously very useful. Disadvantages of the KMP are that this method requires a high amount of pre-processing, and that it is still not the most efficient solution for larger alphabets like you have in standard (English) text. We're now going to encounter a faster algorithm for these kinds of text string.

1.4 The Boyer-Moore-Algorithm

The Boyer-Moore algorithm was found by Boyer & Moore in 1974.

The first speciality about it is; the pattern is scanned from right to left when proceeding through the text; which is also called the "looking glass heuristic".

BM works with two different pre-processing strategies to determine the smallest possible shift; each time a mismatch occurs, the algorithm computes both and then chooses the largest possible shift, thus making use of the most efficient strategy for each individual case.

The first strategy is the "bad character" heuristic.

As the name says, this strategy concentrates on the "bad character" in the text which causes the mismatch. If it is not contained in P at all, the pattern can be shifted past it; if it is somewhere in the pattern, you search for the rightmost appearance of the "bad character" in the pattern and match it against the text.

Let's first have a look at the auxiliary function for the "bad character" heuristic:

```
public static int[] buildLastFunction (String pattern) {
    int[] last = new int[128];           // assume ASCII character set
    for (int i = 0; i < 128; i++) {
        last[i] = -1;                   // initialise array
    }
    for (int i = 0; i < pattern.length; i++) {
        last[pattern.charAt(i)] = i;    // implicit cast to integer ASCII code
    }
    return last;
}
```

For each character of the alphabet (in this case the ASCII set), we determine its rightmost occurrence in the pattern and write our result into an array. Then, each time a mismatch occurs, we can look up the value of "last" for our bad character and find out how far the pattern can be shifted to the right.

Below, I listed up a simplified version of the algorithm in full which only uses the "bad character" heuristic.

```
public static int BMmatch (String text, String pattern) {
    int[] last = buildLastFunction(pattern);
    int n = text.length();
    int m = pattern.length();
    int i = m-1;
    if (i > n-1)
        return -1;                       // no match if pattern is longer than text
    int j = m - 1;
```

```

do {
    if (pattern.charAt(j) == text.charAt(i))
        if (j == 0) {
            return i; // match
        } else {
            i--; j--; // left-to-right-scan
        }
    } else { // bad-character heuristic
        i = i + m - Math.min(j, 1 + last[text.charAt(i)]);
        j = m - 1;
    } while (i <= n - 1);
return -1; } // no match

```

The first thing that is checked is: is the pattern longer than the text? In this case it would be clear that there can't be a match. We set text and pattern pointer to the starting point, which is the rightmost character in the pattern. Now we compare. When j equals $m - 1$, we know we found the match in full, and we return the position of the valid shift. If not, j and i are decremented and we continue the comparison.

In case text and pattern character don't match, the auxiliary function is called, we determine the rightmost occurrence of the bad character in the pattern and modify i and j accordingly; if we have examined all valid shifts and no match has been found, we know the pattern doesn't appear in the text and return -1.

The second strategy is the "good suffix" heuristic. The name should already remind of the Knuth-Morris-Pratt -algorithm, where we tried to find the largest suffix of the "false start" that is also a prefix of the pattern.

Here again, the pattern is checked for self-repetitive parts. The few modifications in the pre-processing are due to the fact that this Boyer-Moore-version of the Knuth-Morris-Pratt strategy has to be adapted to the right-to-left-scans of the main Boyer-Moore-algorithm.

The combination of these two different strategies reduces the number of shifts and character comparisons very effectively. The Boyer-Moore algorithm achieves an average-case-running time of $O(n/m)$ for pattern matching on standard text; only a small fraction of the text characters will actually be examined during the search. (Note that both pre-processing functions operate only on P , not on T !). On the other hand, the worst-case-complexity of this algorithm is $O(n \cdot m)$, just as bad as the worst case boundary of the naive algorithm; it is inadvisable to use the algorithm for small alphabets and highly self-repetitive patterns.

III Other algorithms and improvements

Of course, the algorithms and methods we just mentioned are not the only ones that deal with pattern matching; there are several other algorithms and a number of varieties and improvements to the ones I discussed.

Notably the Boyer-Moore-algorithm has often been used as a starting point to find an even better solution to the string matching problem; works of this kind are the Apostolico & Giancarlo variant, and Cole's linear worst case bound for Boyer-Moore.

The "strong suffix rule" is an improvement which works both for the KMP algorithm and the "good suffix" strategy of the Boyer-Moore-algorithm; it's a technique to avoid that the same mismatch happens twice, and eliminates a few more shifts.

Horspool has shown that a simplified version of the Boyer-Moore-algorithm which uses only the bad-character-heuristic will achieve about the same running time for the average case.

The Rabin-Karp-algorithm finally is a totally different approach to our string matching problem; it works with hash tables.

IV Bibliography

Here is an overview of the literature I have used; to each book I added a few comments to simplify the choice for further reading.

| Source | Comment |
|--|---|
| R. Sedgewick: Algorithmen Addison-Wesley Bonn, 1991 | gives a brief description of the algorithms; not too detailed and quite easy to understand. Best source to start with; also available in English |
| U. Schöning: Algorithmen - kurz gefaßt Spektrum Heidelberg u.a. 1997 | also a rather short source; harder to understand than the Sedgewick text, but well-structured and more precise. One of the few authors I found who discusses the "finite automaton matcher " separately |
| T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen Spektrum Verlag Heidelberg, 1990 | very detailed descriptions of the algorithms; not always well explained |
| M. T. Goodrich, R. Tamassia: Data Structures and Algorithms in Java John Wiley & Sons, Inc., 1998 | contains a Java implementation of KMP, Boyer-Moore and Brute Force; few pieces of information about the history of the algorithms, good illustrations; best source for readers who want more than just a short introduction to string matching algorithms |
| T. H. Cormen, C.E. Leiserson, L. Rivest: Introduction to Algorithms The MIT Press Cambridge, Mass. u.a., 1990 | the most formalistic text I found; contains proof of correctness and worst-case boundaries for each algorithm well illustrated by numerous examples |
| G. A. Stephen: String searching algorithms World Scientific, Singapore u.a., 1994 | lengthy explanations, much information about the history of the string matching algorithms; just for additional reading |
| Dan Gusfield: Algorithms on strings, trees and sequences Cambridge University Press, 1997 | only advisable for advanced readers; after a short introduction, G. lists up several possibilities to improve KMP and Boyer-Moore in detail |