

MSc in Computer Science and Engineering

**Advanced Algorithms 2016-2017**

**Text Searching:**

In this project students had to implement and compare several online algorithms for matching

small patterns against a larger text. This problem is recurrent in computer science specially in bioinformatics applications, therefore we will use DNA sequences to explore this problem.

1. **INTRODUCTION:**

The string matching problem consists in finding one, several or all occurrences of a pattern P in a larger text T.

There are several solutions that solve this problem and in this project we explore 3 of them. The first one is the *Naive algorithm,* this is the simplest one and consists in trying to match P with every substring\* of T with the same size as P. As the name indicates this is a naïve approach and in 1977 Donald Knuth, Vaughan Pratt and James H. Morris published a better algorithm called *Knuth-Morris-Pratt algorithm* which we will examine in second. The third algorithm we are going to examine is *Boyer-Moore algorithm* developed by Robert S. Boyer and J. Strother Moore also in 1977. Both *Knuth-Morris-Pratt* and *Boyer-Moore* preprocess the pattern to obtain a better performance than what we had with the naïve approach.

In this project we present the three algorithms mentioned before and also a dynamic data structure to store strings. The final goal is to be able to say which one is better and in what situations for the DNA sequence problem. To achieve this, we implement, test and compare the solutions. The comparison will be executed with randomly generated strings T, highly repetitive strings T and the pattern might appear or not. To measure the effectiveness of the algorithm we are going to count the number of comparisons and the execution time. In section 2 we introduce the problem and introduce some nomenclature. In section 3 we devote a section to our dynamic structure and we devote a section to each algorithm. Finally, we compare the three algorithms in Section 4 and conclude in Section 5.

1. **BACKGROUND**:

We formalize the string matching problem as follow. We assume that a text T is an array T [0..n] of length n and a pattern P is and array P [0..m] of length m where n  m. We also assume that every element presented in P and T belongs to a finite alphabet . For example, in the DNA sequence = {A, T, C, G}.

In figure 1 have an alphabet = {a, b, c} and as we can see the pattern P occurs in T starting in position T [3], we say that P occur in T with a shift 3. If P appears in T with shift s we say that s is a valid shift, otherwise we call s an invalid shift. Our goal is to find every valid shift in T for a pattern P.

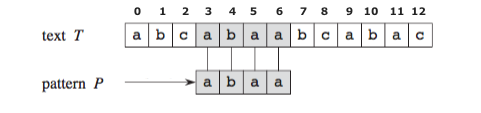


Figure 1 - example of a string matching problem.

The concept of suffix and prefix are also important in this context. We say that a string x with size m is a prefix of another string y with size n and n  m, if y starts with x. In the example of figure 1 the string ab is a prefix of P. A suffix is the opposite. We say that a string x with size m is a suffix of another string y with size n and n  m, if y ends with x. In the example of figure 1 the string aa is a suffix o P.

1. **DATA STRUCTURE AND ALGORITHMS:**
   1. **Dynamic Array:**

A dynamic array is a variable-size data structure that allows elements to be added or removed. In the context of this project, since we don’t know the initial size of text T or pattern P, this structure is suitable for our problem.

Dynamic arrays are similar to an array with 2 extra operations, add and remove, and they share many advantages like locality of reference and data cache utilization, compactness and random access.

Comparing with linked lists dynamic arrays have faster indexing (constant vs linear) and typically faster iteration due to improved locality of reference however they require linear time to insert or delete at an arbitrary location, since all elements must be moved, while linked lists can do this in constant time, but for this project, since we are interested in iterating over the elements in order and access specific indexes of a string the dynamic array advantages overcome the disadvantage of a linear worst case for insert, and if we analyze dynamic arrays in and amortized sense we see that insert at the end is constant (as shown in the theoretical lessons).

Our implementation of this structure is based on the C structure presented on figure 2, and we populate the dynamic array with the function from figure 3. The *readString* function is going to read element by element until it finds the special character ‘\n’. While reading, it is storing the input and counting how many elements it has stored. If at some point the number of elements stored reaches the size of the string allocated, we duplicate the size by reallocating memory.

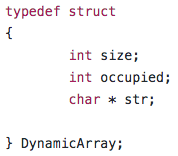


Figure 2 - Dynamic Array C structure.

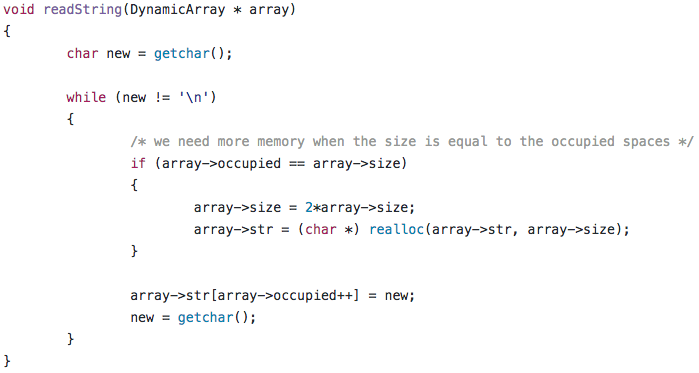


Figure 3 – Function that stores the Input.

* 1. **Naive Approach:**

The solution provided by the naive approach consists in starting at a shift s = 0 check, character by character, if s is a valid shift. For each shift s 1 out of 3 things can happen:

1. We find a character in P that differ from a character in the substring T [s ... s + m], which means that shift s is invalid and we move to the next shift = s+1.
2. We reach the end of P, then we have found a match between T [s … s + m] and P. We print s and move to the next.
3. Shift s reach the value n-m+1, which means s is not a valid shift anymore and the algorithm terminates.

The problem with this approach is the complexity. The naive approach is a O (N x M).

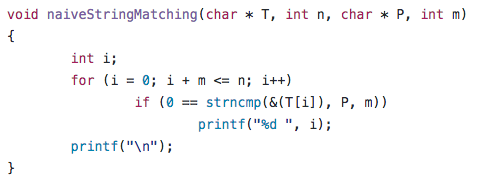


Figure 4 - Our implementation of the Naive Algorithm in C

* 1. **Knuth Morris Pratt Algorithm:**

The Knuth Morris Pratt algorithm is a linear time algorithm. This algorithm is based on the idea that unnecessary comparisons can be avoided because the word itself embodies sufficient information to determine where the next match could begin. To do this the KMP algorithm is divided in two parts, a preprocessing part where we build the  table using the prefix function, and a searching part which consists in finding the valid shifts. The first has a complexity of  (M) and the second one a complexity of  (N), which together gives us a liner complexity of  (N + M).

Given a pattern P [1 ... m], the prefix function for pattern P if defined as:

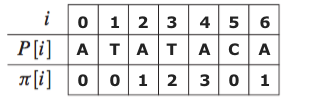


Figure 5 - Example of the of Prefix Function for a pattern P

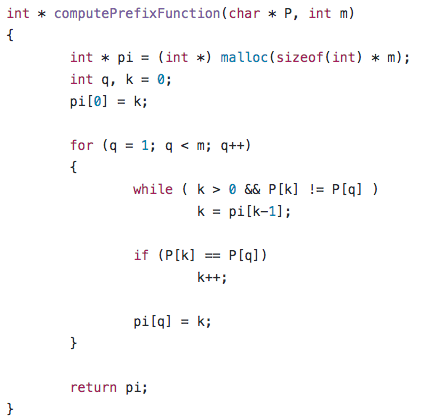


Figure 6 - Implementation of the Prefix function in C

After the preprocessing stage the searching stage is very similar. Both stages match a string against the pattern: KMP-matcher (the searching function) matches T against P while the Prefix function matches P against itself.

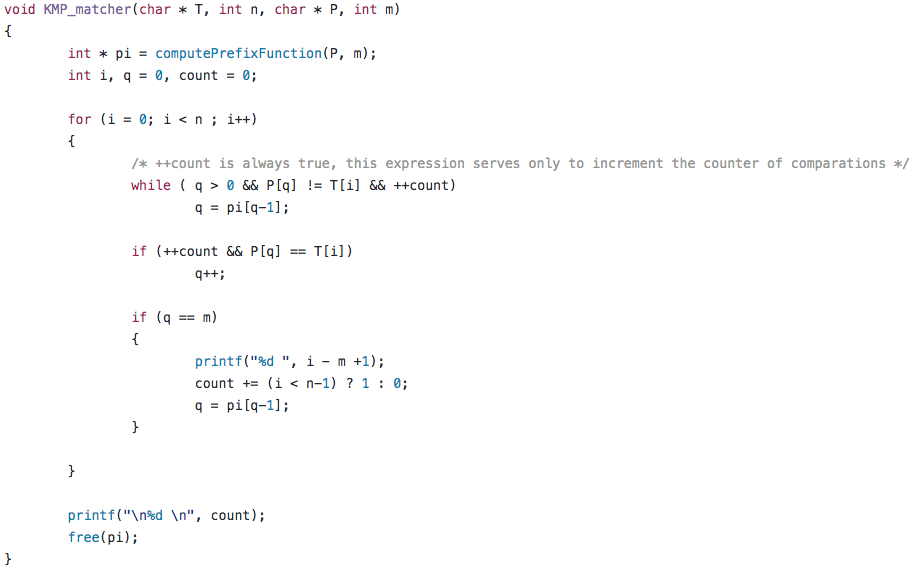


Figure 7 - KMP-matcher function implementation in C, variable count and its manipulation serves only for the purpose of this project where we needed to count the number of comparisons.

ALTERAR IMAGEM DO KMP PARA CONTAGENS COM SENTIDO

* 1. **Boyer Moore Algorithm:**

Boyer Moore algorithm uses 3 clever ideas to obtain a method that typically examines fewer than m + n characters, and in O (M + N) for some cases. The three ideas are: Right to Left scan, The Bad Character shift rule and The Good Suffix shift rule.

1. The Right to Left scan is as simple as checking for an occurrence of P by scanning the characters from right to left rather than left to right.
2. The Extended Bad Character Rule says that when a mismatch occurs at position i of P and the mismatched character in T is x then shift P to the right so that the closest x to the left of position i in P is bellow the mismatched x in T.

This rule requires O (|) space to maintain the lookup table (table R) in memory and O (M) to build that table. For the DNA sequence the space issue is not a problem since is size 4, and even in big sequences the probability of finding the first four letters in the beginning of the sequence is high which makes the worst case for building table R a very rare one.

1. The Good Suffix Shift aligns the matched part of T, T’, with the rightmost occurrence of that character sequence in the pattern that is preceded by a different character (including none, if the matched suffix is also a prefix of the pattern) than the matched suffix T’ of the pattern – if there is such occurrence.

To do this Good Suffix rule uses 2 tables: L’ and l’, and to build those tables we need a third table N. We define next these tables: