Group 44:
Ricardo Rei nº 78047
Miguel Carvalho nº 78052

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 the *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 that obtained 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.

2. BACKGROUND:

We formalize the string matching problem as follow. We assume that a text T is an array
T [0..n-1] of length n and a pattern P is and array P [0..m-1] of length m where n ≥ m. We also assume that every element presented in P and T belongs to a finite alphabet Σ. For example, in the DNA sequence Σ = {A, T, C, G}.
In figure 1 have an alphabet Σ = {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.

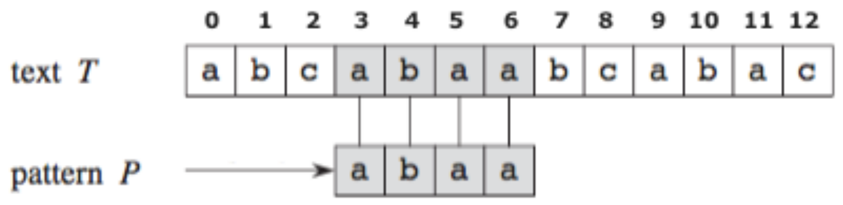


Figure 1 - example of a string matching problem.

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.

3. DATA STRUCTURE AND ALGORITHMS:

3.1 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.

```c
typedef struct
{
        int size;
        int occupied;
        char * str;

} DynamicArray;
```

Figure 2 - Dynamic Array C structure.

```c
void readString(DynamicArray * array)
{
        char new = getchar();

        while (new != '\n')
        {
                /* we need more memory when the size is equal to the occupied spaces */
                if (array->occupied == array->size)
                {
                        array->size = 2*array->size;
                        array->str = (char *) realloc(array->str, array->size);
                }

                array->str[array->occupied++] = new;
                new = getchar();
        }
}
```

Figure 3 – Function that stores the Input.

3.2 Naive Approach:

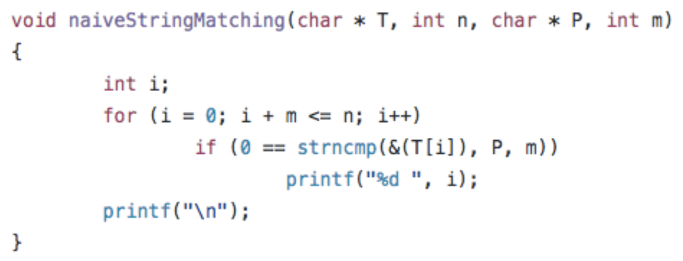
The solution provided by the naive approach consists in starting at a shift s = 0 check, character by character, if s is a valid shift. For each shift s, 1 out of 3 things can happen:

(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).



```c
void naiveStringMatching(char * T, int n, char * P, int m)
{
        int i;
        for (i = 0; i + m <= n; i++)
                if (0 == strncmp(&(T[i]), P, m))
                        printf("%d ", i);
        printf("\n");
}
```

Figure 4 - Our implementation of the Naive Algorithm in C

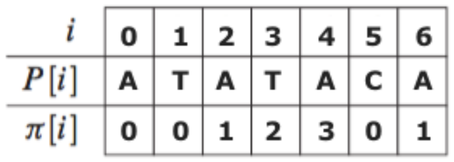
3.3 Knuth Morris Pratt Algorithm:

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 π table using the prefix function, and a searching part which consists in finding the valid shifts. The first has a complexity of Θ (M) and the second one a complexity of Θ (N), which together gives us a liner complexity of Θ (N + M).

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

$$\pi: \{0,1,\dots,m\} \rightarrow \{0,1,\dots m-1\} such that:$$

$$\pi[q] = max\{k: k < q \wedge P_k is a prefix of P_q\}$$



i	0	1	2	3	4	5	6
$P[i]$	A	T	A	T	A	C	A
$\pi[i]$	0	0	1	2	3	0	1

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

```c
int * computePrefixFunction(char * P, int m)
{
        int * pi = (int *) malloc(sizeof(int) * m);
        int q, k = 0;
        pi[0] = k;

        for (q = 1; q < m; q++)
        {
                while ( k > 0 && P[k] != P[q] )
                        k = pi[k-1];

                if (P[k] == P[q])
                        k++;

                pi[q] = k;
        }

        return pi;
}
```

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.

```c
void KMP_matcher(char * T, int n, char * P, int m)
{
        int * pi = computePrefixFunction(P, m);
        int i, q = 0, count = 0;

        for (i = 0; i < n ; i++)
        {
                /* ++count is always true, this expression serves only to increment the counter of comparations */
                while ( q > 0 && P[q] != T[i] && ++count)
                        q = pi[q-1];

                if (++count && P[q] == T[i])
                        q++;

                if (q == m)
                {
                        printf("%d ", i - m +1);
                        q = pi[q-1];
                }

        }

        printf("\n%d \n", count);
        free(pi);
}
```

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.

Note: the code submitted in mooshak considers ´\0´in the end of the pattern.

Group 44:
Ricardo Rei nº 78047
Miguel Carvalho nº 78052

3.4 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.

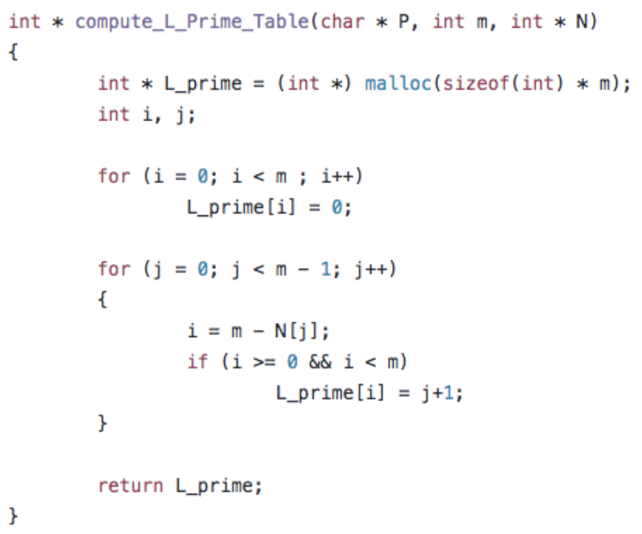
(1) The Right to Left scan is as simple as checking for an occurrence of P by scanning the characters from right to left rather than left to right.

(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 ($|\Sigma \vee$) 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 $\Sigma \vee$ 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.

(3) 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:

- **Table N**: For a string P, $N[j]$ is the length of the longest suffix of the substring P[0..j] that is also a suffix of the full string. To compute this table, we made a cycle for i starting at 0 until $m - 1$and at each iteration we use another cycle for j starting at i and decrementing until, 0 or a mismatch between $P[j]$ and $P[m - 1 - i + j]$, after this inner cycle $N[j]$ is the equal to i – j.

- **Table L':** For a string P, $L'[i]$ is the largest index j less than m such that: $N[j] = |P[i..m - 1]| = m - i$. This table can be computed using the Z-Based Boyer-Moore algorithm.



```
int * compute_L_Prime_Table(char * P, int m, int * N)
{
        int * L_prime = (int *) malloc(sizeof(int) * m);
        int i, j;

        for (i = 0; i < m ; i++)
                L_prime[i] = 0;

        for (j = 0; j < m - 1; j++)
        {
                i = m - N[j];
                if (i >= 0 && i < m)
                        L_prime[i] = j+1;
        }

        return L_prime;
}
```

Figure 8 - Z-Based Boyer-Moore Algorithm to compute table L' in linear Time implementation in C.

- **Table l’:** For a string P, $l'[i]$ equals the largest $j \leq \vee P[i..m-1]|$, which is $m-i$, such that $N[j] = j+1$. This table can be computed using a cycle for i starting at 2 until $m+1$ and at each iteration if $N[i-1] = i$, $l'[m-i]$ is equals to i, otherwise its equal to $l'[m-i+1]$. At last $l'[m-1]$ is equal to 1 if $N[0] = 1$, 0 otherwise.
  Note that with this algorithm , l' is computed in linear time.

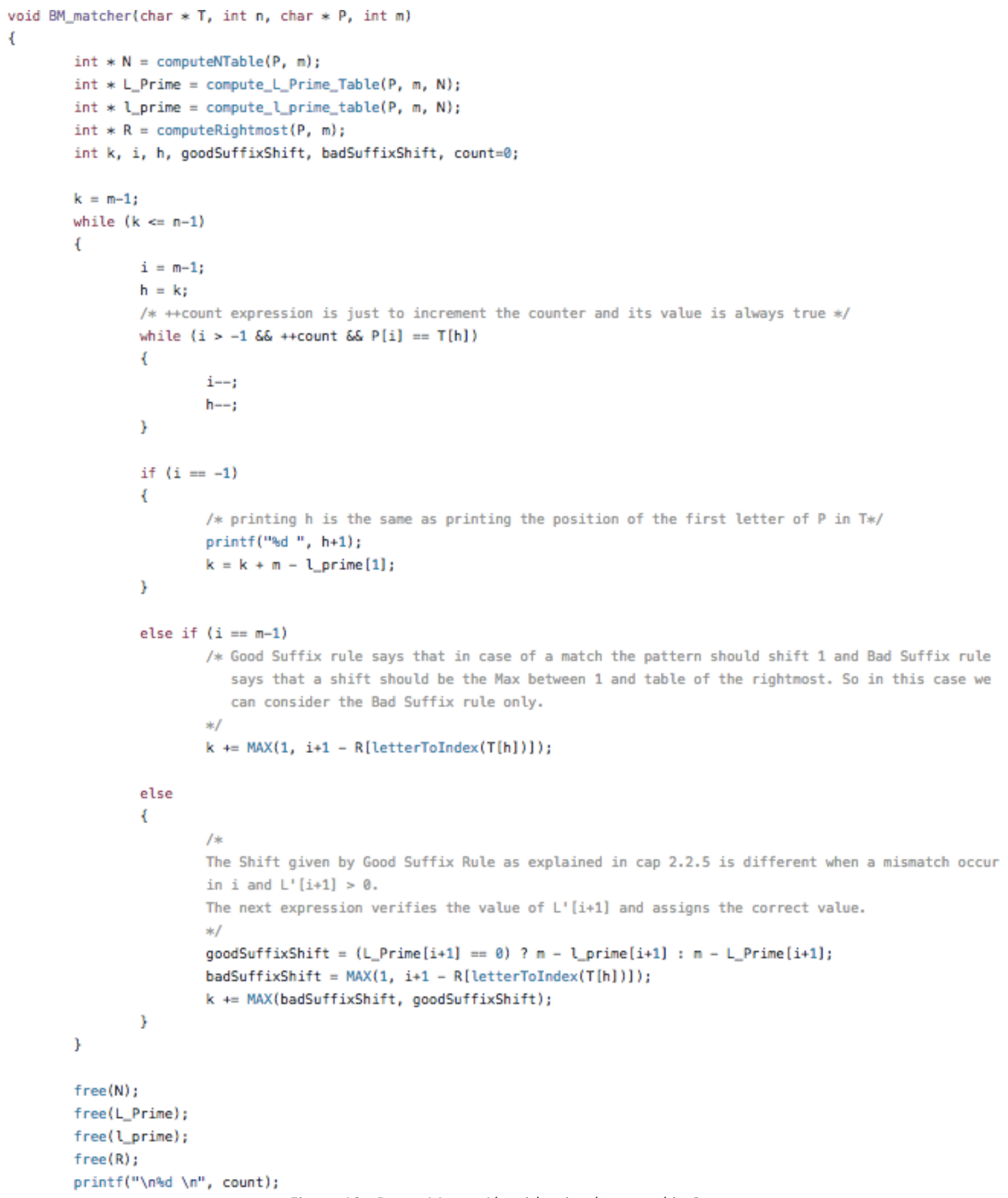
$$N = 1 \quad 2 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0$$
$$L' = 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 2 \quad 8$$
$$l' = 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 1$$

Figure 9 - Tables for AACTGTCAA pattern.

Having computed L' and l' for each position in P the Good Suffix Rule says: if, during the search stage a mismatch occurs at position i ($i < m-1$) of P and $L'[i+1] > 0$ then shift P by $m - L'[i+1]$ to the right, otherwise shift P by $m - l'[i+1]$. When an occurrence of P is found shift P by $m - l'[1]$.
One special case remains. When the first comparison is mismatch (i.e., P[i with $i = m-1$ mismatches) then P should be shifted by 1.

Finally, after the preprocessing phase Boyer Moore algorithm will always shift by the largest amount given by either of the rules.

The algorithm implemented in C is shown in figure 10.



```c
void BM_matcher(char * T, int n, char * P, int m)
{
	int * N = computeNTable(P, m);
	int * L_Prime = compute_L_Prime_Table(P, m, N);
	int * l_prime = compute_l_prime_table(P, m, N);
	int * R = computeRightmost(P, m);
	int k, i, h, goodSuffixShift, badSuffixShift, count=0;

	k = m-1;
	while (k <= n-1)
	{
		i = m-1;
		h = k;
		/* ++count expression is just to increment the counter and its value is always true */
		while (i > -1 && ++count && P[i] == T[h])
		{
			i--;
			h--;
		}

		if (i == -1)
		{
			/* printing h is the same as printing the position of the first letter of P in T*/
			printf("%d ", h+1);
			k = k + m - l_prime[1];
		}

		else if (i == m-1)
			/* Good Suffix rule says that in case of a match the pattern should shift 1 and Bad Suffix rule
			   says that a shift should be the Max between 1 and table of the rightmost. So in this case we
			   can consider the Bad Suffix rule only.
			*/
			k += MAX(1, i+1 - R[letterToIndex(T[h])]);

		else
		{
			/*
			The Shift given by Good Suffix Rule as explained in cap 2.2.5 is different when a mismatch occur
			in i and L'[i+1] > 0.
			The next expression verifies the value of L'[i+1] and assigns the correct value.
			*/
			goodSuffixShift = (L_Prime[i+1] == 0) ? m - l_prime[i+1] : m - L_Prime[i+1];
			badSuffixShift = MAX(1, i+1 - R[letterToIndex(T[h])]);
			k += MAX(badSuffixShift, goodSuffixShift);
		}
	}

	free(N);
	free(L_Prime);
	free(l_prime);
	free(R);
	printf("\n%d \n", count);
```

Figure 10 - Boyer-Moore Algorithm implemented in C.

Note: count variable serves only to count the number of comparisons.

4. EXPERIMENTAL ANALYSIS:

The tests we present next were run in a MacBook Air, Processor 1,4 GHz Intel Core i5 with a memory of 4 GB 1600 MHz DDR3. This laptop runs a OS X El Capitan Version 0.11.6.
With these tests the goal was to understand empirically the influence of text size, pattern size, and the type of the text (highly repetitive vs low regularity), to do this we evaluate the execution time of the algorithms and we compared the number of comparisons between Knuth-Morris-Pratt and Boyer-Moore.

(1) In the first test a set of files with texts with low regularity of 1MB, 2MB and 4MB size, and a pattern of size 10, were generated. Since the text is randomly generated and the pattern too, even with size 10, the pattern doesn't appear much (varying between 0 times and 6 times).
It was expected a linear behavior for the 3 algorithms since a pattern so small shouldn't influence the complexity of the algorithms. We also expected to see Knuth-Morris-Pratt and Boyer-Moore with slightly better execution time since the pre-processing phase can be neglected.

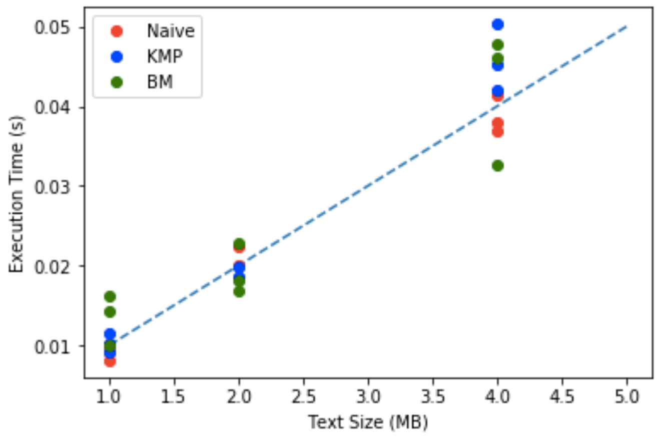


Figure 11 - Execution Time with a small pattern and a varying, low regularity, text size.

The results presented in figure 11, to our amazement, show that the Naive Approach was never the worst in terms of execution time.

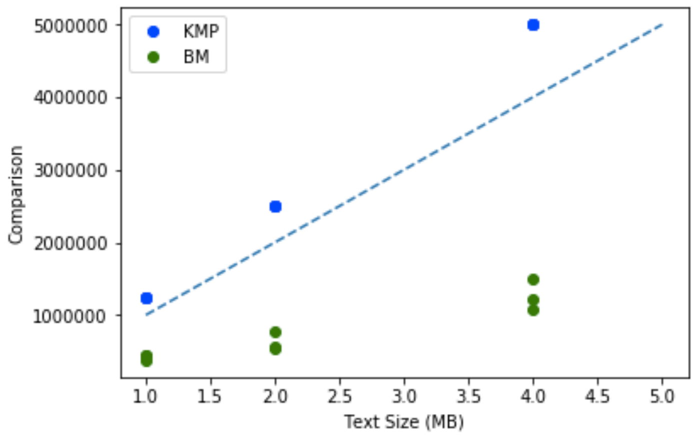


Figure 12 - Comparisons Number with a small pattern and a varying, low regularity, text size.

As showed in figure 12, Boyer-Moore always had less comparisons than Knuth-Morris-Pratt even in tests where it had worst execution time. We expected that less comparisons should revel a better execution time yet that's not true in all the cases.

(2) In the second test a text with low regularity of 1MB size was generated. After the text has been created we extracted parts of that text with different sizes to create the patterns, and for each pattern extracted we stored the text and the pattern in a file.
It was expected worst execution time from Naive Approach comparing with Knuth-Morris-Pratt and Boyer-Moore since Naive Approach is more influenced in terms of complexity with the size of the pattern.

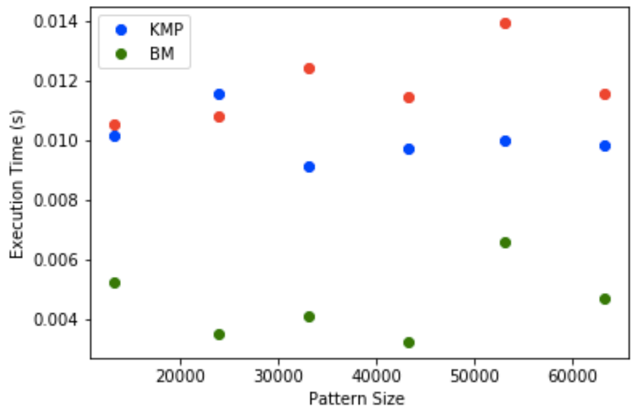


Figure 13 - Execution Time Influence of varying pattern size with 1MB low regularity text.

As showed by figure 13, with bigger patterns, Naive Approach is the worst algorithm in execution time. Yet we expected a bigger difference, but since the text has very low regularity the Naive approach doesn't need many comparisons to realize the pattern is not in a certain position. The worst case in which Naive Approach compares all the pattern and mismatches only on the last character never occurs.
In terms of number of comparisons, Knuth-Morris-Pratt and Boyer-Moore showed similar values to those presented in figure 12 for the a 1MB text.
If we compare the execution time and number of comparisons for the 1MB texts in figure 11 and figure 12 we see that the size of the pattern didn't influence much the Knuth-Morris-Pratt. In other hand it looks like the size of the pattern actually made Boyer-Moore get a better performance, but more tests should be done to support this intuition.

(3) For the third test a highly repetitive text was generated. After the text has been created we extracted parts of that text with different sizes to create the patterns. We ensured that the patterns only appeared once.
Our goal was to see how these algorithms scale with a highly repetitive text and different pattern sizes.
It was expected that Knuth-Morris-Pratt and Boyer-Moore to get high performances because of the calculated shifts.

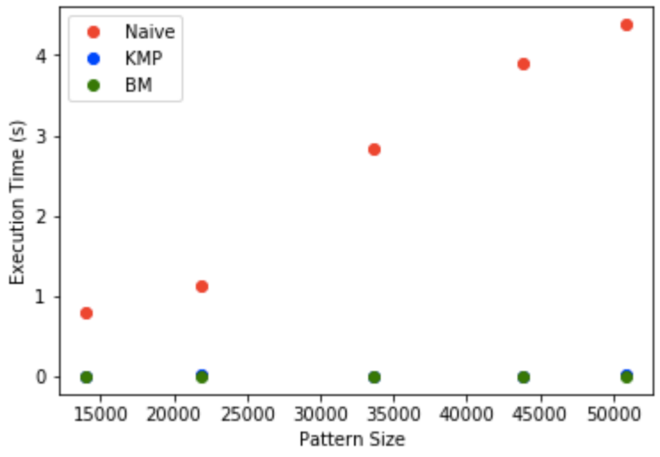


Figure 14 - Execution time in a repetitive text and a pattern that only appear once.

As showed in figure 14 the calculated shifts in the pre-processing phase allowed Knuth-Morris-Pratt and Boyer-Moore to obtain higher performances, actually the size of the pattern in these conditions did not influence much the execution time of these 2 algorithms. The Naive approach in contrast with the results presented in figure 13 showed a very poor performance in these conditions.

(4) For the forth test a set of highly repetitive texts with 1MB, 1.5MB, 2MB, 2.5M, 3MB and 4MB size were generated and for each text we took the most frequent pattern with size 1000. The goal was to observe how the size of the text influences the complexity when the pattern frequency is very high.

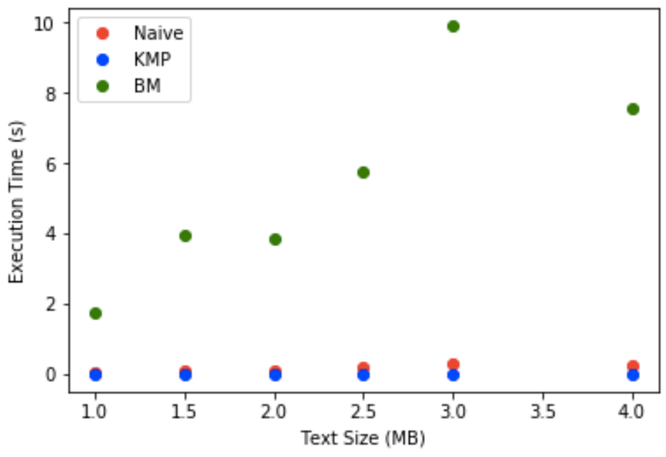


Figure 15 - Execution time in highly repetitive Text and a Pattern with high frequency.

In this conditions Boyer-Moore showed a very poor performance in terms of execution time comparing with the other algorithm.

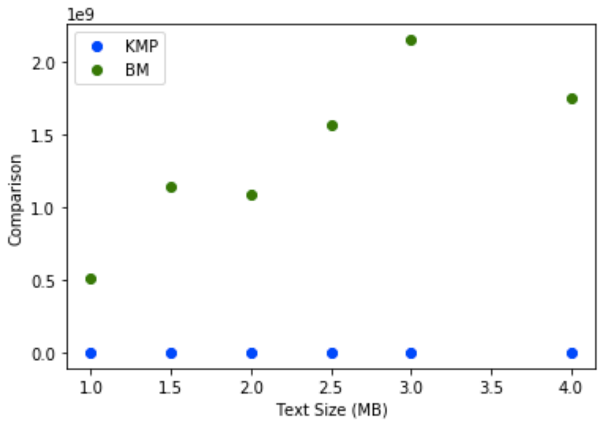


Figure 16 - Comparisons Number highly repetitive text and a pattern with high frequency.

In terms of comparisons, Boyer-Moore made a lot more than Knuth-Morris-Pratt.

5. DISCUSSION:

For this project the goal was to study and understand these 3 famous algorithms for the String Matching problem. From the studying we conclude that for low regularity texts and rare patterns (which is the DNA case) Boyer-Moore is the best algorithm. If the text is highly repetitive with a big pattern that appears a lot of times the best choice is Knuth-Morris-Pratt and finally for texts highly repetitive and small patterns that appear a lot the Naive Approach might be the simplest one and the results are satisfactory.

6. REFERENCES:

- Algorithms on Strings, Trees, and Sequences: Dan Gusfield 1997 Cambridge University Press
- Introduction to Algorithms: Thomas H. Cormen, Charles E. Leiserson, Rolnald L.Rivest and Clifford Stain 2009 MIT Press