2D String Matching and Pattern Identification

Advance Problem Solving, M.Tech CSE 2018

Project Report

Group members:
Devesh Tewari, 2018201039
Tarun Munjal, 2018202007

International Institute of Information Technology,
Hyderabad

Contents

Abstract

Pattern Matching addresses the problem of finding all occurrences of a pattern string in a text string. Pattern matching algorithms have many practical applications. Computational molecular biology and the world wide web provide settings in which efficient pattern matching algorithms are essential. New problems are constantly being defined. Original data structures are introduced and existing data structures are enhanced to provide more efficient solutions to pattern matching problems.

2D String Matching refers to finding all occurrences of a 2D pattern string in a 2D text string. *The Two Dimensional Pattern Matching Problem* is formally defined as follows:

Let the Alphabet be Σ.
INPUT: Text array *T[1..n, 1..n]* and *Pattern array P[1..m, 1..m]*
OUTPUT: All locations *(i, j)* in *T* where there is an occurrence of the pattern i.e.
$T[i+k, j+l] = P[k+1, l+1]$, $0 \leq k,l < m$.

In this project, we implemented four pattern matching algorithms in two dimensions. The four algorithms implemented are:

1. Naive 2D pattern matching
2. Baker Bird 2D matching algorithm
3. Baeza-Yates Régnier algorithm
4. An algorithm using Hash values

These algorithms are discussed in detail in the subsequent chapters.

Chapter 1

The Naive approach

The first approach we discuss for 2D pattern matching is the simplest Naive approach. This is the most simplest case. At the end of this chapter, we will analyze the Time and Space Complexity of this algorithm. Later we will realize why this approach is worst among all other approaches discussed in this report.

1.1 About the Algorithm

This approach is also called a brute-force approach because it searches for the pattern array in all sub-matrix of the text array. The idea is to consider all possible patterns in the test array with size as that of pattern array and match each corresponding element.

Consider a text array *T[1..n, 1..n]* and pattern array *P[1..m, 1..m]*. The algorithm starts by matching the pattern array starting from T[0, 0]. If the elements *T[0, 0]* and *P[0, 0]* match, we proceed forward to check the next element of both the test array as well as the pattern array for a match, and we keep on proceeding this way until there is a mismatch between the corresponding elements of test array and pattern array. Whenever there is a mismatch at *T[i, j]*, we say that the pattern array and text array do not match at *(i, j)* and we move to the next element of text array. When this happens, we will restart this process starting from next element of text array (e.g. *T[0,1]*) and the first element of the pattern array i.e. *P[0, 0]*.

We say that it's a match at *(i, j)* if all the text array elements and pattern array elements are matched. In other words, $T[i+k, j+l] = P[k+1, l+1]$, $0 \leq k,l < m$.

After finding a match at position *(i, j)* we do not stop the algorithm and consider all possible patterns that can match. The algorithm at the end must return all the matches found.

This algorithm requires no pre-processing and is the most simplest approach. Though this is the simplest one, it's running time can be very large because of brute-force matching when the size of pattern array and text array is large. Let's analyze the running time of this algorithm.

1.2 Analyzing the algorithm

In this section, we will analyze the time and space complexity for this algorithm. We assume, text array *T[1..n, 1..n]* and pattern array *P[1..m, 1..m]*.

1.2.1 Time Complexity

The algorithm runs for each possible text sub-matrix of size as that of pattern. So, first we need number of such sub-matrices. This will be equal to $(n - m + 1)^2$. So, the algorithm will have these many iterations in the text array.

Now for each of these iterations, we are matching each character until there is a mismatch. The worst case will arise when all the text sub-matrices match the pattern. In this case each iteration will require m^2 steps.

Both of the above facts imply that the worst case steps required by the algorithm will be equal to $(n - m + 1)^2.m^2$. Which is O($n^2.m^2$).

The worst case time complexity is too large. But for random inputs, the time complexity would be very less because there will be mismatches mostly.

1.2.2 Space Complexity

This algorithm does not require any extra space as it only searches for pattern in the text array and pattern array and does not have any additional data structure.

So the space complexity for the algorithm is O(1).

Chapter 2

Baker Bird Algorithm

The first linear-time algorithm for two dimensional pattern matching with bounded alphabets was obtained independently by Bird and Baker, who convert the 2D pattern matching problem into a 1D string matching problem. The Bird / Baker algorithm solves 2D pattern matching in $O(n^2 + m^2)$ time and space.

A pattern array may be viewed as a sequence of strings, for example its columns. To locate columns of the pattern array within columns of a text array searching for several strings is required . The Bird and Baker's solution uses well-known Aho-Corasick algorithm of pattern matching for a set of patterns and Knuth Morris And Pratt Algorithm for string matching. Aho-Corasick allows construction of associated output actions in every state of the AC pattern matching machine i.e for every element present in the text array we have an output action.

2.1 Algorithm

1. **Machine-Building Step :** Consider PA as set of it's rows , then given a number to each distinct row we build a data structure Trie(tree + failure function) using Aho-Corasick algorithm of size(m*m).

2. **Column-generating step :** Given Trie , build TA' and PA' column by column, where each element of TA' and PA' is a transition state in Trie.

3. **Row-matching step :** Given TA' and PA', we find match of last row of PA' within TA'. For each row of TA' apply KMP to find row(pm) , where pm is the last row in PA'.

Let Pattern array be PA and text array be TA.

An example of the AC pattern matching machine for searching three strings of a pattern array of size (3 3) is shown in Figure 2.1. The first column containing string "abc" is related to final state 5 in Figure 2.2; the second column containing string "cba" is related to final state 8; the last column of "aab" to final state 3.

Transition diagram of an AC pattern matching for the example of pattern array PA is shown above .

$$PA = \begin{array}{|c|c|c|}\hline a & c & a \\ \hline b & b & a \\ \hline c & a & b \\ \hline \end{array}\ ,\ TA = \begin{array}{|c|c|c|c|c|c|c|}\hline b & b & a & b & b & a & b \\ \hline a & a & c & a & c & b & a \\ \hline b & b & b & a & c & a & c \\ \hline a & c & a & b & b & a & b \\ \hline c & a & a & c & a & b & a \\ \hline b & b & b & b & a & c & c \\ \hline a & c & c & a & b & a & b \\ \hline \end{array}\ ,\ |PA| = (3 \times 3),\ |TA| = (7 \times 7)$$

Figure 2.1: Hash values for text array's row 2

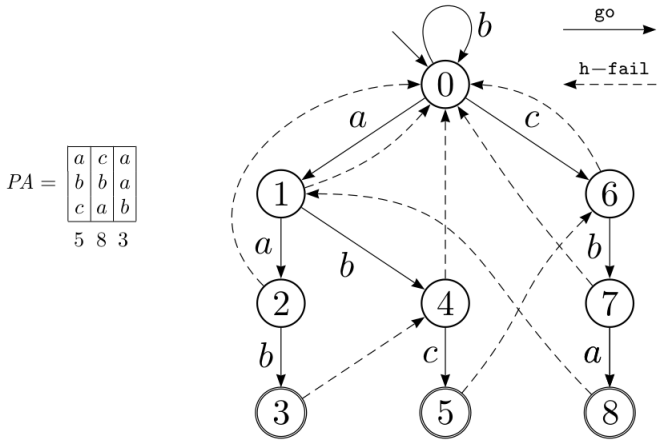

Figure 2.2: The transition diagram of an AC pattern matching machine for the example pattern array

By using the automaton built we now construct a text array prime TA' and pattern array PA' which contains all the transitions in column order. Once array TA and PA' are generated, the bottom row of PA have to be located (i.e. the last string, when we read PA horizontally) within array TA.

Original text array TA and resulting array TA after processing by AC pattern matching machine are below.

Now to search for a string of final states associated with pattern array PA we use KMP algorithm (Knuth-Morris-Pratt).

Set of final states F of AC automaton is *F = 3, 5, 8*. Recall that columns of PA' have final states labeled 5, 8, 3, in that order. Occurrences of string "583" in the pre-processed array of states TA' (see Figure 2.3) gives 2D occurrences of the pattern array in the original text array TA. Result of matching for the pattern array are shown in Figure 2.4.

$TA=$

b	b	a	b	b	a	b
a	a	c	a	c	b	a
b	b	b	a	c	a	c
a	c	a	b	b	a	b
c	a	a	c	a	b	a
b	b	b	b	a	c	c
a	c	c	a	b	a	b

$TA'=$

0	0	1	0	0	1	0
1	1	6	1	6	4	1
4	4	7	2	6	1	6
1	5	8	3	7	2	7
6	1	1	5	8	3	8
7	4	4	7	2	5	6
8	5	5	8	3	1	7

Figure 2.3: Text array TA before and after preprocessing based on the run of the AC pattern matching machine in columns

	a	c	a			
	b	b	**a**	**c**	**a**	
	c	a	**b**	**b**	**a**	
		a	**c**	**a**	**b**	
		b	*b*	*a*		
		c	*a*	*b*		

0	0	1	0	0	1	0
1	1	6	1	6	4	1
4	4	7	2	6	1	6
1	5	8	3	7	2	7
6	1	1	**5**	**8**	**3**	8
7	4	4	7	2	5	6
8	5	*5*	*8*	*3*	1	7

Figure 2.4: 2D occurrences of pattern array PA in TA (left) and the relation to the result of KMP pattern matching in array TA of states of the AC pattern matching machine (right)

2.2 Analysis

2.2.1 Time Based

Steps:

1. Aho-Corasick takes O(m) time for each row , where m is the number of elements in each row .For all rows of Pattern *PA[1..m,1..m]*, overall time taken by this step is Time(O(m^2)).

2. We have to traverse all the elements of TA to make TA‘ and all the elements of PA to make PA‘ in column order. Therefore if *TA[1..n,1..n]* and *PA[1..m,1..m]* then time taken will be O(n*n + m*m). Therefore , Overall Time(O(n^2 + m^2)). Given TA‘ and PA‘, we find match of last row of PA‘ within TA‘. For each row of TA‘ apply KMP to find row(pm) , where pm is the last row in PA‘.

3. We apply KMP algorithm in this step at each row of TA‘. KMP pre-processing takes Time(O(m)) , where m is the number of elements in a pattern to search for. KMP takes Time(O(n)) where n is the number of elements in text. Hence for every row Total time , Time(O(n(n+m)).

Therefore , Total time : Time(O((m^2) + (n^2+m^2) + (n(n+m)))) i.e O(n^2 + m^2).

2.2.2 Space Ordered

Steps:

1. Building an automaton takes O(m^2) space , where m is the number of elements in the pattern matrix.

2. It takes O(n^2) space , where n is the number of elements in present in each row in text matrix , to build TA‘ where we store all transitions. O(m) to store the last row of pattern matrix.

3. KMP take O(m) space for preprocessing.

Hence , Overall Space required is O(m^2 + n^2 + m^2 + m) , i.e. O(n^2 + m^2)

Chapter 3

Baeza-Yates Régnier Algorithm

The Baeza-Yates Régnier Algorithm considers the pattern as m superposed strings Pi of length m. Like Bird's algorithm it decompose the search in a one dimensional (say horizontal) multi-string search. However, the key idea is to perform the horizontal search only on n/m rows of the text, possibly followed by vertical searches when matches occur. These rows cover all possible positions where the pattern may occur.

The algorithm uses dictionary searching to locate a two-dimensional pattern in a two-dimensional text. The idea is to treat the pattern as a dictionary of ordinary strings and perform a search in only n/m rows of the text to find all candidate positions (see Figure 3.1). Obviously, no pattern occurrence will be missed.

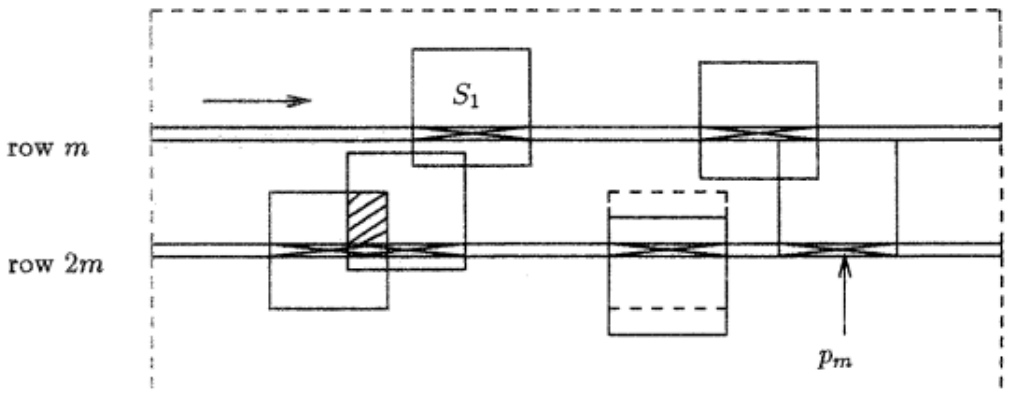

Figure 3.1: Potential matches for the checking phase of the 2D algorithm

Suppose there is a match of one of the strings from the dictionary, that is one complete row, say l^{th} ,of the two-dimensional pattern has been found at position x, y in text. Then the *m-l* rows above and *l-1* rows below the current row in the two-dimensional text are to be searched

for a 2D occurrence, starting from the column *x-m+1*. In the algorithm, the search is performed using Aho-Corasick algorithm.

3.1 Algorithm

Steps:

1. Consider PA , then given a number to each distinct row we build an automaton named Trie(tree + failure function) using Aho-Corasick algorithm of size(m*m).
2. In each every n/m^{th} row , for every pattern that appears , we check for the vertical search function for above and below rows.
3. If we found exact match of above and below strings also , then we have found the Pattern.

3.2 Analysis

3.2.1 Time Ordered

Steps:

1. Aho-Corasick takes O(m) time for each row , where m is the number of elements in each row of Pattern. Overall time : Time(O(m^2)).
2. For each n/m row , we run horizontal search that takes O(n) time , where n is the number of elements in a row .If found a match of i^{th} string in pattern at column j, then for above i rows and below m-i rows , we apply horizontal search starting from *m-i+1* row and j^{th} column till $j + m^{th}$ column.4
   Time: O(n/m(n(m(m))) , i.e. O(n^2m)
3. Output the match , O(1)

Total time : O(nm^2)

But this algorithm , has an average case complexity O(n^2/m + m^2) , which is efficient than baker-bird.

3.2.2 Space Ordered

Space required is for automaton build using Aho-Corasick algorithm i.e. O(m^2).

Chapter 4

A Hashing Algorithm

The Naive String Matching algorithm slides the pattern one by one. After each slide, it one by one checks characters at the current shift and if all characters match then prints the match.

Like the Naive Algorithm, this Hashing algorithm also slides the pattern matrix one by one. But unlike the Naive algorithm, this algorithm matches the hash value of the pattern's rows with the hash value of current row of text of length m, (see Figure 4.1, here m = 2) and if the hash values of a row match then only it starts matching subsequent rows. In the next section we describe this algorithm in detail.

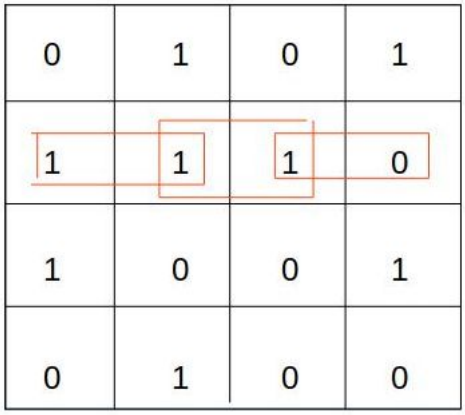

Figure 4.1: Hash values for text array's row 2

4.1 Algorithm's working

The hashing algorithm requires pre-processing of the hash values of rows. The pre-processing is required to compute the hash values and to store them in a data structure.

The Hashing algorithm needs to calculate hash values for following strings.

1. Every row of pattern itself.
2. All the sub-strings in a row of text of length m.

It maintains two matrices, *hash-text[1 .. n, 1 .. n-m+1]* and *hash-pattern[1..m]* which stores the above hash values.

hash-text[i, j] stores the hash value of *T[i, j..j+m-1]*.

hash-pattern[i] stores the hash value of *P[i, 1..m]*.

This algorithm now applies the Knuth Morris Pratt (KMP) algorithm on each column of text-hash to match the hash values of the pattern-hash. KMP algorithm is linear time and does the job efficiently.

The KMP algorithm also requires pre-processing. In the pre-processing stage, it computes the LPS (largest proper prefix which is also a suffix) array, *lps[1..m]* for the matrix hash-pattern. *lps[i]* stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern *hash-pattern[1..m]* .

Since we need to efficiently calculate hash values for all the sub-strings of size m of text for each row, we must have a hash function which has following property. Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say *hash(text[i, s+1 .. s+m])* must be efficiently computable from *hash(text[i, s .. s+m-1])* and *text[i, s+m]* i.e., *hash(text[i, s+1 .. s+m]) = rehash(text[i, s+m], hash(i, text[i, s .. s+m-1])* and *rehash* must be O(1) operation.

We have considered the case where the input text is in binary i.e. either 0 or 1. Our implementation of this algorithm computes the hash values as integer which is the decimal equivalent of the input. e.g. if m = 3 and *T[i, j..j+m-1] = [1,0,1]* , then it's hash value will be 5 (decimal equivalent of 101). The computation of these hash values is optimized by using bit-wise operators where ever possible.

Since an integer is of 32 bits, the pattern array must have dimension less than 32.

The rehash is computed as follows:
Consider *hash(text[i, s+1 .. s+m]) = rehash(text[i, s+m], hash(i, text[i, s .. s+m-1]))*
To compute rehash, firstly we left shift the integer hash(i, text[i, s .. s+m-1] by 1 i.e. multiply it by 2. Then the m^{th} bit from left is reset and the rightmost bit of *text[i, s+m]* is added to it.
e.g. to compute *hash(text[i, s+1 .. s+m])* , if *text[i, s+m] = 1* and *hash(i, text[i, s .. s+m-1]) = 11 (1011 in binary)* . Then we first left shift the hash by one and make it *22 (10110)* . Then we reset the m^{th} bit from left and make it *6 (00110)* . In the final step we add text[i, s+m] to it and make it *7 (0111)* which is our required hash i.e. *hash(text[i, s+1 .. s+m]).*

4.2 Analyzing the Algorithm

We will divide the analysis of this algorithm into two parts, the time complexity and the space complexity. The below sections does the analysis.

4.2.1 Time Complexity

Let us divide the algorithm into two parts for obtaining it's time complexity.

The first part is the pre-processing. i.e. computing and storing the hash values for the text array as well as the pattern array. The second part is the KMP algorithm which is run on each column of hash-text where the pattern is hash-pattern.
In the first part, for computing the hash value for i^{th} row of text, we first compute the first hash i.e. hash-text[i, 1] in O(m) because we need to traverse elements in text[i, 1..m]. Now all other hash values in this row are calculated in O(1) by using rehash. So, for every text's row we require O(m + (n - m)) = O(n) time. Hence for all the rows of text array, we require O(n^2) time.

For the text-array, we need to compute hash for each row which requires O(m^2) time. So the time to compute hash-text and hash-array is O(n^2 + m^2).

For the second part, we need to apply KMP algorithm. It's pre-processing for the hash-pattern will take O(m) time. We need to apply KMP algorithm to each column of text-hash. Since there are n - m + 1 columns and n rows in text-hash and KMP algorithm runs in linear time, the time complexity for the second part is O((n - m + 1)n + m).

Adding the time complexities for first and second part, the total time complexity for the algorithm is O(n^2 + m^2).

4.2.2 Space Complexity

This algorithm requires two additional 2D arrays for storing the hash values. They are *hash-text[1 .. n, 1 .. n-m+1]* and *hash-pattern[1..m]* . So the space complexity is O(n(n - m + 1) + m) which is roughly O(n^2 + m) when n is very larger than m.

Chapter 5

Experimental Analysis

In this chapter, we will be doing experimental analysis on all the algorithms described in previous chapters.

The algorithms will be provided with different types of inputs, namely random, binary and the worst case input for naive algorithm. All of these types will be tested on different text array sizes and pattern array sizes.

We will then analyze the results and based on that we will compare these algorithms. Detailed graph plotting will be used for this task as it will help us visualize the time complexities of the algorithms for different types and sizes of input.

These tests are run on *HP-Pavillion-Notebook* with *intel® Core™ i5-5200U CPU @ 2.20GHz* and *8 GB RAM*. Plotly tool is used to plot the graphs.

5.1 Naive Algorithm's worst case

In this section, we will prove why naive algorithm is the worst among all other algorithms. The Naive's worst case occurs when all the elements of the text array are equal and there is a match in all positions. The Naive Algorithm runs in $O(n^2.m^2)$ which is a multiple of m^2. Figure 5.1 illustrates this fact. In this figure, it plots the graph for a fixed size text array and varies the pattern size.

The other algorithms on the other hand, do not have much effect on their running times when we keep the test array size fixed and vary the pattern size. This is illustrated in Figure 5.2, Figure 5.3 and Figure 5.4 for the Baker Bird Algorithm, Baeza-Yates Régnier Algorithm and Hashing Algorithm respectively. There is hardly and change in their running times on varying the pattern size.

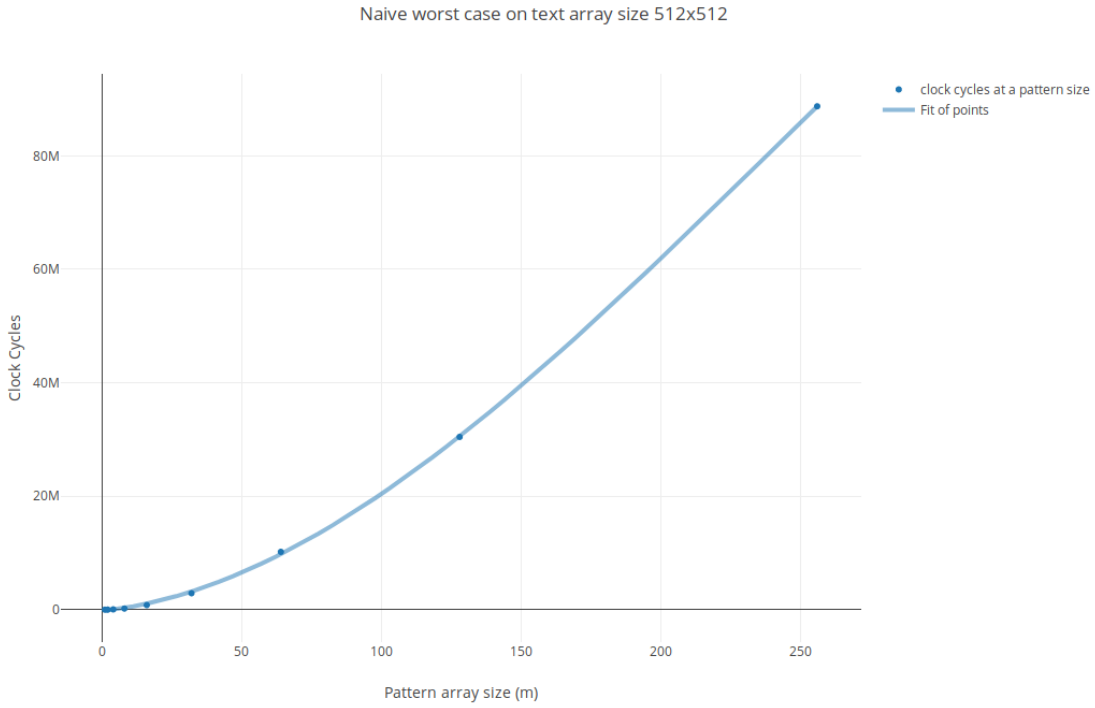

Figure 5.1: Naive's worst case

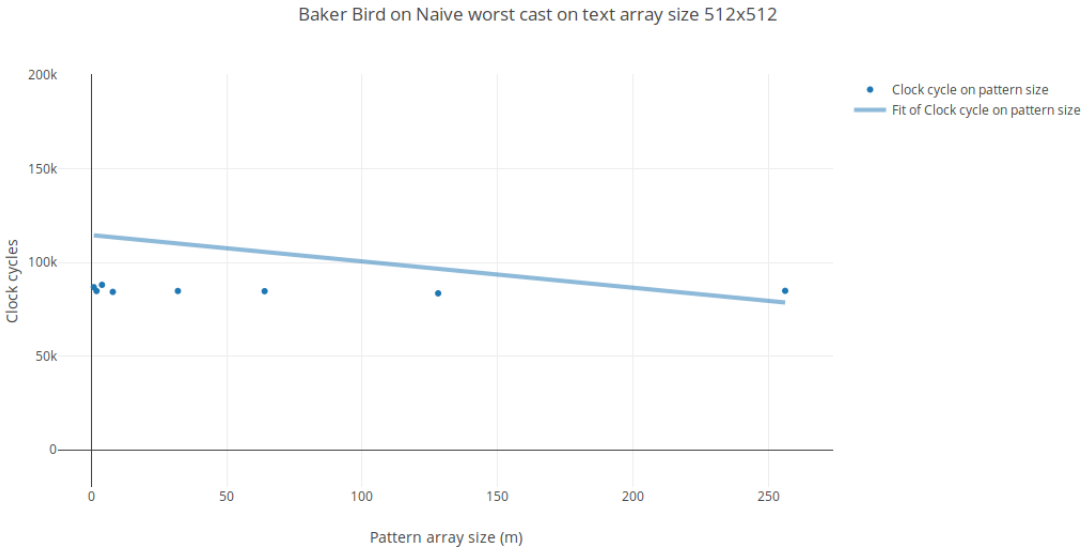

Figure 5.2: Baker Bird run on Naive's worst case

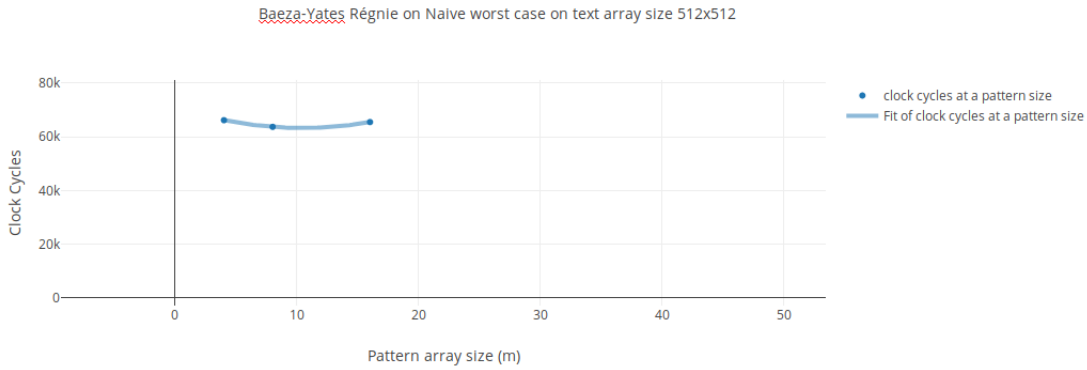

Figure 5.3: Baeza-Yates Régnier Algorithm run on Naive's worst case

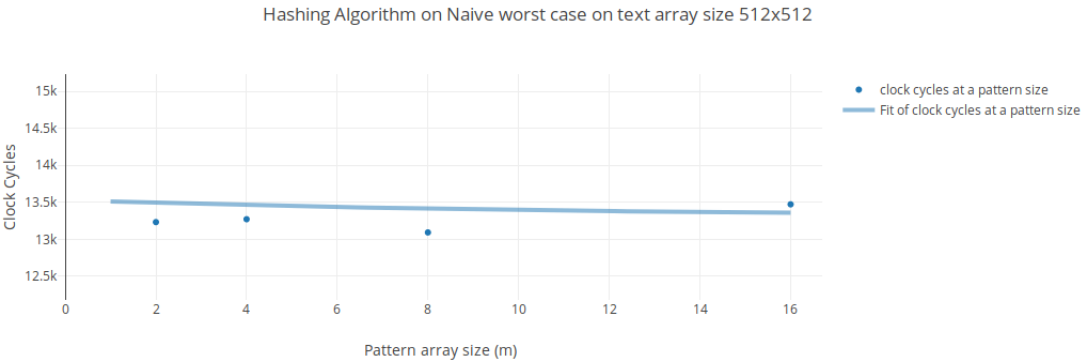

Figure 5.4: Hashing Algorithm run on Naive's worst case

5.2 Binary Input

We consider the binary inputs because the Hashing algorithms runs very efficiently on binary inputs. But it has a constraint that the pattern size must be less than 32.

We will test the running time for this algorithm on different text size less than 1025. The pattern size will be half the text size, but it will be atmost 16 to satisfy the above constraint. Figure 5.5 illustrates how efficient this algorithm is.

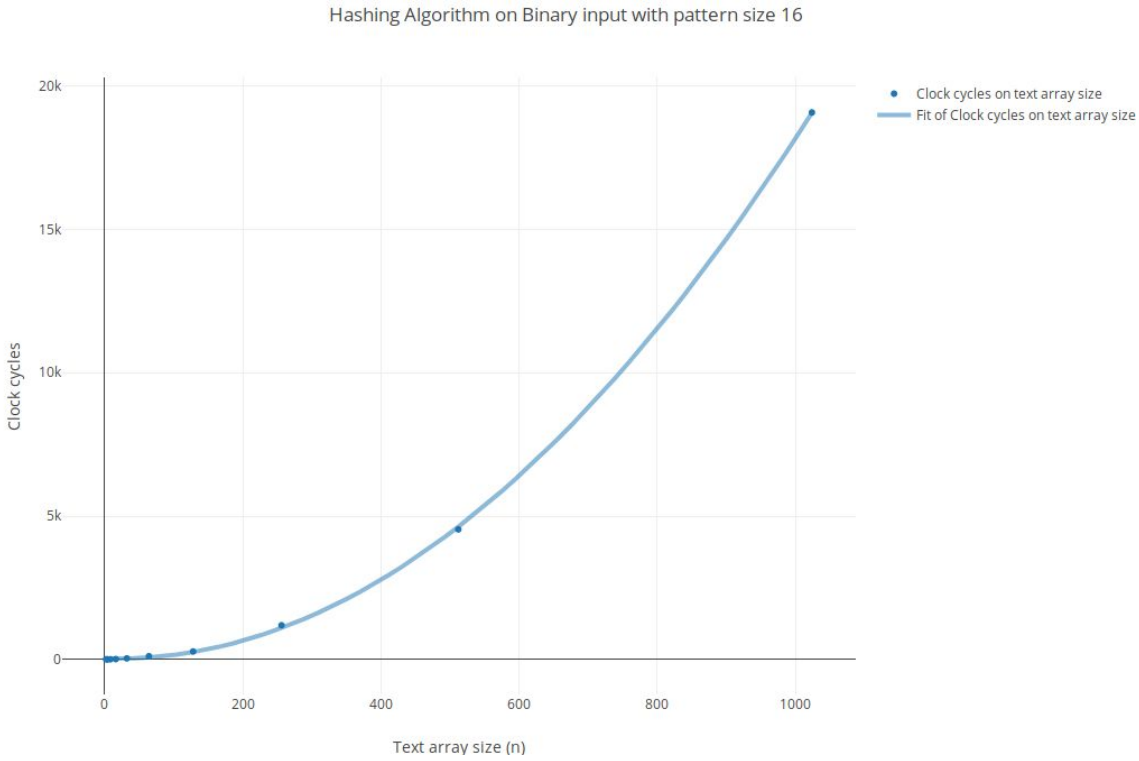

Figure 5.5: Hashing Algorithm Analysis

5.3 Random Input

In this section, we will run all the algorithms on random input text array and a random pattern array present at any location *(i, j)* in the text array. This analysis is not so god for the Naive approach because it will be almost it's best case. While this analysis is great for all other algorithms.
Figure 5.6 to Figure 5.9 illustrates how the algorithms perform on random input.

From the above experiments, we see that Baeza-Yates Régnier Algorithm performs better than Baker Bird Algorithm in terms of it's running time. Also, we can see that the Hashing algorithm is the fastest by a great margin.

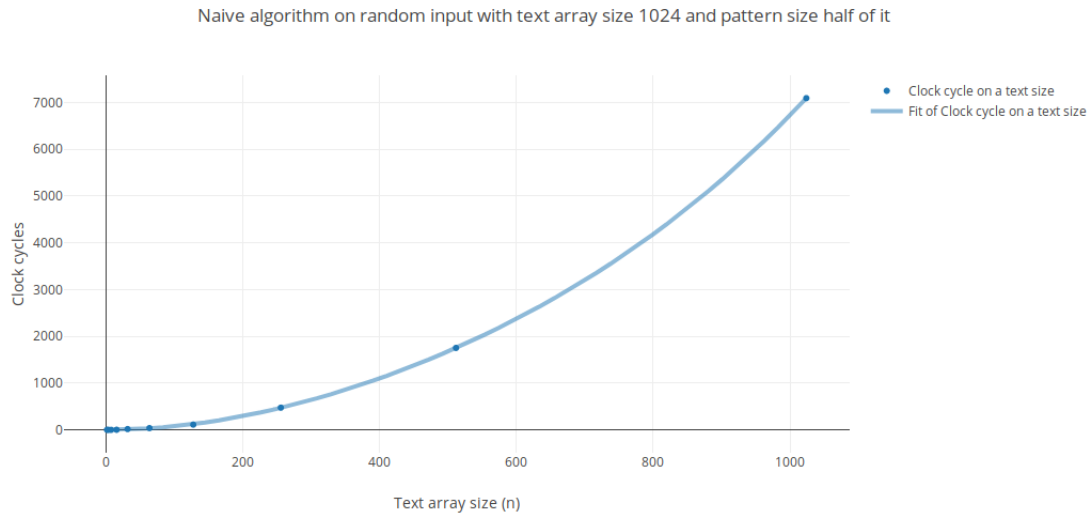

Figure 5.6: Naive Algo on random input

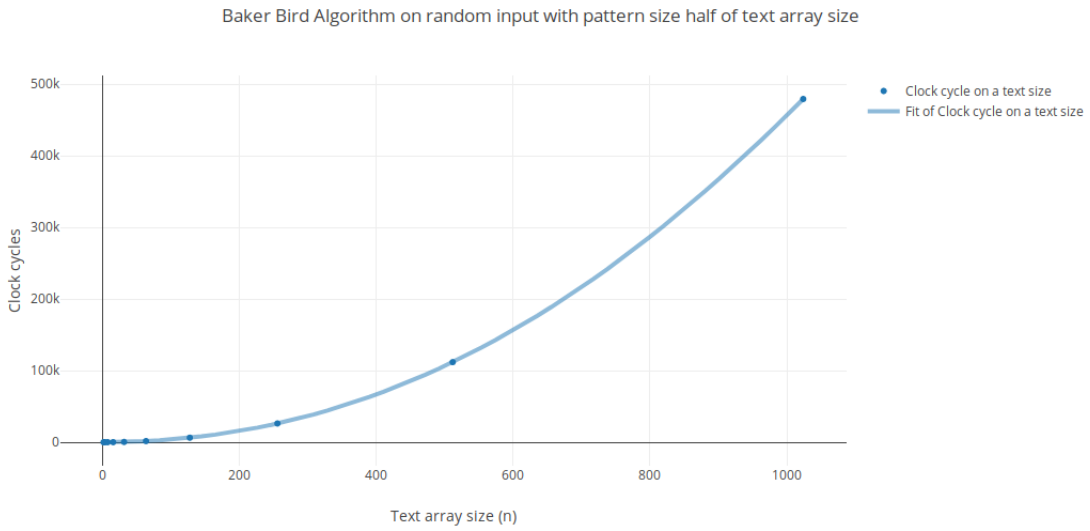

Figure 5.7: Baker Bird Algo on random input

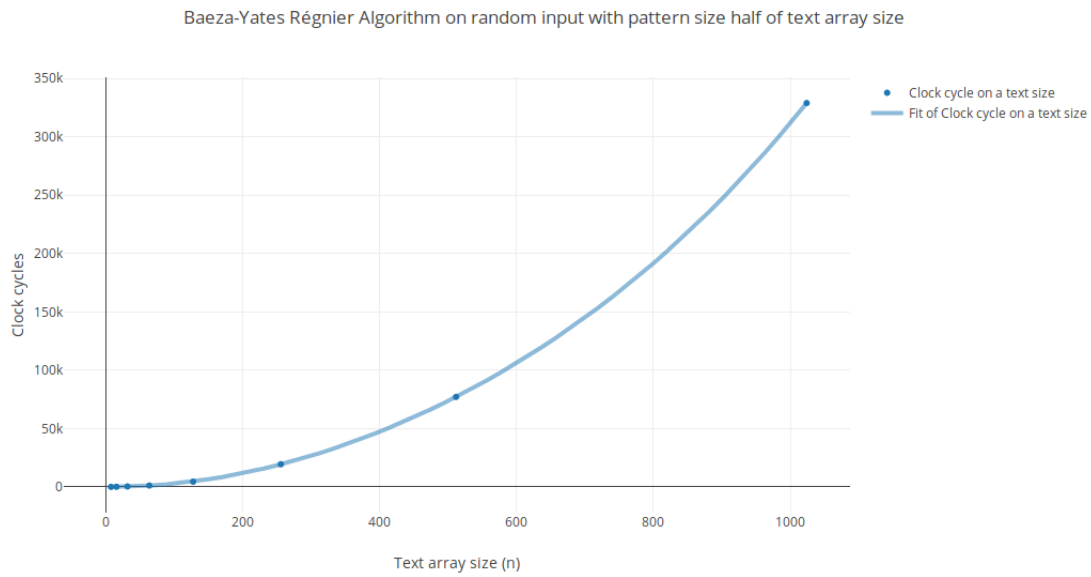

Figure 5.8: Baeza-Yates Régnier Algo on random input

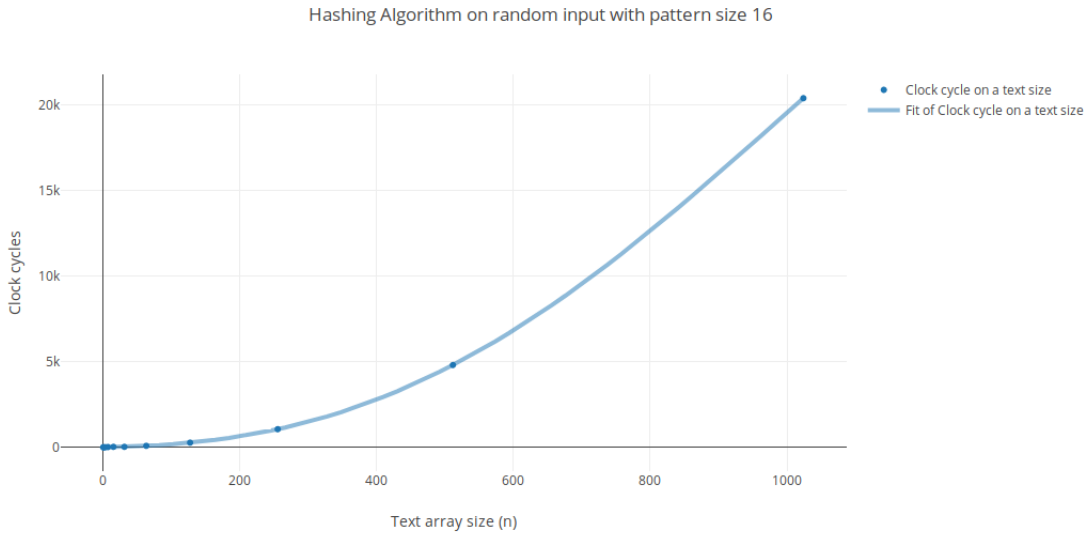

Figure 5.9: Hashing Algo on random input

5.4 Conclusion

In this section we will analyze the results from above section and infer the 'goodness' of the algorithms.

We saw in Section 5.1 that when the Naive's worst case is given as input to all these algorithms, and we changed only the pattern size, there was an exponential increase in the running time of Naive's algorithm, while there was hardly and change in other algorithm's running time. This implies that Naive approach can be very time consuming for large pattern array sizes.

So now we compare the other three Algorithms, Baker Bird Algorithm, Baeza-Yates Régnier Algorithm and Hashing Algorithm. Looking at the running time of these algorithm's in Section 5.2 and Section 5.3, we can conclude that the Hashing algorithm is better than the other two. But, it can be run only on binary inputs and pattern array size less than 32. So if we have binary inputs and pattern array size less than 32, the hashing algorithm should be used, otherwise,
we are left with Baker Bird Algorithm and Baeza-Yates Régnier Algorithm. The above results show that the running time of Baeza-Yates Régnier Algorithm is better, but it also has a constraint on pattern size. So, if the input is not binary and pattern array size less than 32, Baeza-Yates Régnier Algorithm should be used, and in all other cases, Baker Bird Algorithm has to be used.

Bibliography

[1] Jan Žďárek Two-dimensional Pattern Matching Using Automata Approach. A thesis submitted to the Faculty of Electrical Engineering, Czech Technical University in Prague,

[2] Theodore P. Baker A technique for extending rapid exact-match string matching to arrays of more than one dimension. SIAM J. Comput., 7(4):533-541, November 1978.

[3] Alfred Vaino Aho and Margaret John Corasick Efficient string matching: An aid to bibliographic search. Commun. ACM, 18(6):333-340, 1975.

[4] Richard S. Bird Two-dimensional pattern matching. Inf. Process. Lett., 6(5):168- 170, October 1977.

[5] Ricardo A. Baeza-Yates and Gonzalo Navarro Faster approximate string matching. Algorithmica, 23(2):127-158, 1999.

[6] Ricardo A. Baeza-Yates and Mireille Régnier Fast two-dimensional pattern matching. Inf. Process. Lett., 45(1):51-57, 1993.

[7] Shoshana Neuburger Pattern Matching Algorithms: An Overview Department of Computer Science The Graduate Center, CUNY September 15, 2009

[8] Ricardo Baeza-Yates Fast Algorithms for Two Dimensional and Multiple Pattern Matching (Preliminary version)

[9] Stack Overflow Rabin Karp Algorithm for 2D arrays
https://stackoverflow.com/questions/29991650/rabin-karp-algorithm-for-2d-arrays

[10] Geeks for Geeks Aho-Corasick Algorithm for Pattern Searching
https://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/