Advanced Problem Solving,
International Institute of Information Technology,
Hyderabad

2-D STRING MATCHING AND PATTERN IDENTIFICATION

Arushi Agarwal (2019201015)

Jyoti Gambhir (2019201032)

Abstract

String matching is an important problem in text processing and is commonly used to locate one dimensional pattern (string) in a text. The expansion of imaging, graphics and multimedia required the use of pattern matching to higher dimensions, leading to the two and multi-dimensional pattern matching problem. The main problem of two dimensional pattern matching is the huge number of comparisons which are needed to find the occurrence of the two-dimensional pattern in the two-dimensional text.

TABLE OF CONTENTS:

1. PROBLEM STATEMENT

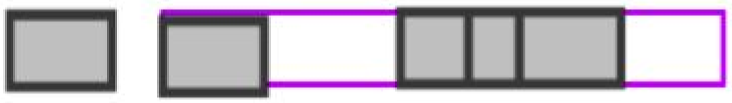
The two dimensional pattern matching problem is defined as:

Let q be an alphabet, given a text array *T (n * n)* and pattern array *P (m * m)*, report all locations (i, j) in T where there is an occurrence of P, i.e.

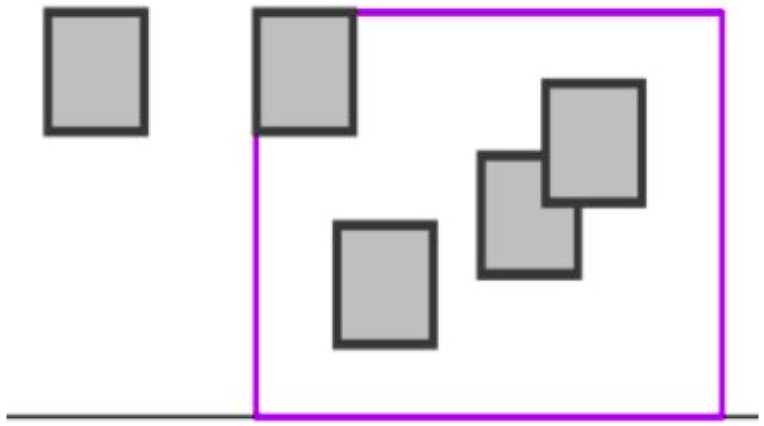
T (I + k, j + l) = P (k, l) for 0 <= k, l <= n and m <= n

When considering string matching in text processing, we could specify its main usage as allocating one-dimensional pattern (string) in the specific text. With the breakthrough in image processing, graphical imaging and multimedia, a need to work in higher levels of pattern matching dimensions was noticed. Two and multi-dimensional pattern matching was introduced.

However, this unique approach faced multiple difficulties in implementation since it required handling extremely large numbers of comparisons to locate similarities and specified occurrences in both two-dimensional patterns and text. In most suggested solutions, a relapse to the one-dimensional pattern down from the two-dimensional problem is carried out. These solutions, though, require pre-processing phase for structure re-constructing before text searching could be applied. There are other algorithms that require the pre-processing phase, but do not necessarily convert the 2D data to one dimensional.



1D String Matching



2D String Matching

2. PROPOSED SOLUTION

To solve the problem of 2D pattern matching we have used the existing 1D pattern matching algorithms, like the one proposed by *Knuth Morris Pratt(KMP)*, the one by *Rabin Karp(RK)*, and the one one given by *Aho Corasick*. These algorithms have then been extended to approach the two Dimensional texts and patterns. Following 2D pattern matching algorithms have thus been implemented:

—>**Naive(Brute Force)**

—>**Zhu Takaoka**

—>**Baker-Bird Algorithm**

—>**Baeza-Yates Regnier Algorithm**

The above (except for Naive) work in **O(n2+m2)** time to find all occurrences of the given 2D pattern in the 2D text, and are based on combinations of the above mentioned 1D pattern matching algorithms, to achieve the task efficiently.

2.1 ONE DIMENSIONAL PATTERN MATCHING ALGORITHMS

2.1.1 RABIN KARP

Rabin-Karp checks the pattern by moving window one by one, but without checking all characters for all cases, it finds the hash value. Only when the hash value matches, does it try to compare the strings for equality. This procedure makes the algorithm more efficient.

Since we need to efficiently calculate hash values for all the substrings of size m of text, there is a need for a hash function which has the 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(txt[s+1 .. s+m]) must be efficiently computable from hash(txt[s .. s+m-1]) and txt[s+m] i.e.,

hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1]))

,and rehash must be O(1) operation. This is called ***Rolling Hash*** Function.

Let's consider an example:

```
Input:
Text: ABAABCDBBABCDDEBCABC
Pattern: ABC

Output:
Found At: 4
Found At: 10
Found At: 14
```

The Algorithm:

```
rabinKarpSearch(text, pattern, prime)
Input: The main text and the pattern. Another prime number of find hash
location
Output: location where patterns are found
Begin
  patLen := pattern Length
  strLen := string Length
  patHash := 0 and strHash := 0, h := 1
  maxChar := total number of characters in character set

  for index i of all characters in pattern, do
    h := (h*maxChar) mod prime
  done

  for all character index i of pattern, do
    patHash := (maxChar*patHash + pattern[i]) mod prime
    strHash := (maxChar*strHash + text[i]) mod prime
  done

  for i := 0 to (strLen - patLen), do
```

```
    if patHash = strHash, then
      for charIndex := 0 to patLen -1, do
        if text[i+charIndex] ≠ pattern[charIndex], then
          break the loop
      done

      if charIndex = patLen, then
        print the location i as pattern found at i position.
    if i < (strLen - patLen), then
      strHash := (maxChar*(strHash – text[i]*h)+text[i+patLen]) mod prime,
    then
    if strHash < 0, then
      strHash := strHash + prime
  done
End
```

The average and best case running time of the Rabin-Karp algorithm is **O(n+m)**, but its worst-case time is **O(nm)**. Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same, as then the hash values of all the substrings of txt[] match with hash value of pat[].

2.1.2 KNUTH MORRIS PRATT

Knuth Morris Pratt (KMP) for pattern matching checks the characters from left to right. When a pattern has a sub-pattern that appears more than once in the sub-pattern, this algorithm finds longest proper prefix which is also suffix to find the starting index for comparison on mismatch.

The KMP matching algorithm uses *degenerating property* (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

Let's consider an example:

```
Input:
Text: ABAABCDBBABCDDEBCABC
Pattern: ABC

Output:
Found At: 4
Found At: 10
Found At: 14
```

The Algorithm:

findPrefix(pattern, m, prefArray):

```
Input: The pattern, the length of pattern and an array to store prefix location
Output: The array to store where prefixes are located
Begin
  length := 0
  prefArray[0] := 0

  for all character index 'i' of pattern, do
    if pattern[i] = pattern[length], then
      increase length by 1
      prefArray[i] := length
    else
      if length ≠ 0 then
        length := prefArray[length - 1]
        decrease i by 1
      else
        prefArray[i] := 0
  done
End
```

kmpAlgorithm(text, pattern):

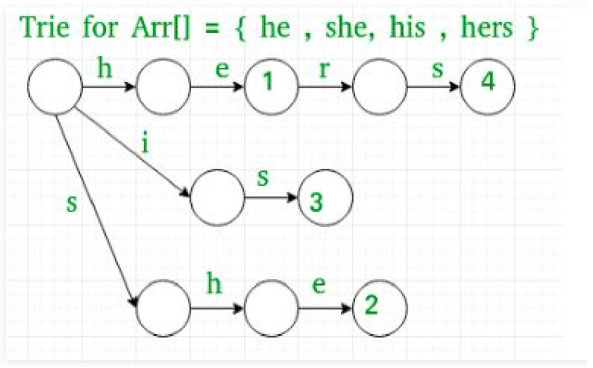
```
Input: The main text, and the pattern, which will be searched
Output: The location where patterns are found
Begin
  n := size of text
  m := size of pattern
  call findPrefix(pattern, m, prefArray)

  while i < n, do
    if text[i] = pattern[j], then
      increase i and j by 1
    if j = m, then
      print the location (i-j) as there is the pattern
      j := prefArray[j-1]
    else if i < n AND pattern[j] ≠ text[i] then
      if j ≠ 0 then
        j := prefArray[j - 1]
      else
        increase i by 1
  done
End
```

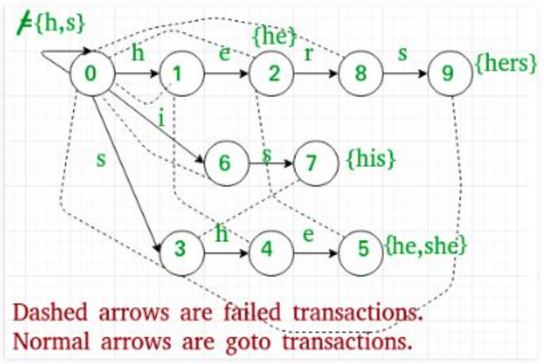
2.1.3 AHO CORASICK

AC algorithm is a classical and suitable solution for exact string matching and it is widely used for multi pattern matching algorithm. AC algorithm contains two main stages: Finite state machine construction stage and matching stage. The algorithm constructs a finite-state machine that resembles a trie with additional links between the various internal nodes. These extra internal links allow fast transitions between failed string matches (e.g. a search for cat in a trie that does not contain cat, but contains cart, and thus would fail at the node prefixed by ca), to other branches of the trie that share a common prefix (e.g., in the previous case, a branch for attribute might be the best lateral transition). This allows the automaton to transition between string matches without the need for backtracking.

When the string dictionary is known in advance (e.g. a computer virus database), the construction of the automaton can be performed once off-line and the compiled automaton stored for later use. In this case, its run time is linear to the length of the input plus the number of matched entries i.e *O(n+m).*

Consider the following example for construction of Trie and Automata for Arr[ ]={ he,she,his,hers }




{h,s}
{he}
h
e
r
s
0
1
2
8
9
{hers}
i
s
6
7
{his}
h
e
3
4
5
{he,she}
Dashed arrows are failed transactions.
Normal arrows are goto transactions.

Let’s consider an example for string matching:

Input:

Text: ABAABCDBBABCDDEBCABC

Pattern: ABC

Output:

Found At: 4

Found At: 10

Found At: 14

The Algorithm:

Input: The list of all patterns, and the size of the list
Output: Generate the transition map to find the patterns

```
buildTree(patternList, size):
Begin
  set all elements of output array to 0
  set failure link NULL
  set all links of child_node to NULL
  end := 0      //at first there is only one state.

  for all patterns ‘i’ in the patternList, do
    word := patternList[i]
    current := root
    for all character ‘ch’ of word, do
      if current->child_node[ch] := NULL then
```

```
        current->child_node[ch] := newstate

      current:= current->child_node[ch]
    done
    current->output := current->output OR (shift left 1 for i times)
  done

  for all characters ch, do
    if root->child_node[ch] is not equal to NULL then
      root->child_node[ch] := root
      insert root->child_node[ch] into a Queue q.
  done

  while q is not empty, do
    newState := first element of q
    delete from q.
    for all possible character ch, do
      if newState->child_node[ch] is not equal to NULL then
        failure := newstate->fail
        while failure->child_node[ch] == NULL, do
          failure := failure->child_node[ch]
        done

        newState->child_node[ch]->fail = failure
        newState->child_node[ch]->output
:=newState->child_node[ch]->output OR failure->output
        insert newState->child_node[ch] into q.
    done
  done
  return state
End
```

getNextState(presentState, nextChar):

```
Input: present state and the next character to determine next state
Output: the next state
Begin
  answer := presentState
  ch := nextChar

  while answer->child_node[ch] = -41, do
    answer := answer->fail
  done
  return answer->child_node[ch]
End
```

Input: List of patterns, size of the list and the main text
Output: The indexes of the text where patterns are found

```
patternSearch(patternList, size, text):
Begin
  call buildTree(patternList, size)
  presentState := root

  for all indexes of the text, do
    if presentState->output.size := 0
      ignore next part and go for next iteration
    for all patterns in the patternList, do
      if the pattern found using output array, then
        print the location where pattern is present
    done
  done
End
```

2.2 TWO DIMENSIONAL PATTERN MATCHING ALGORITHMS

2.2.1 NAIVE ALGORITHM

This is the Brute Force approach to finding a 2D pattern in the given 2D text. We check each and every character of the pattern with each and every character of the text, without leveraging the degenerative property of text, or including the idea of rolling hash in the approach.

The Worst Case Time Complexity of Naive Algo is *O(n*n*m*m)*, where the text matrix os of size n*n and the pattern matrix is of size m*m.

The algorithm:

```
Begin:
for each row in text matrix:
do
        set flag as true
        for each char in row of text matrix:
        do
                for each row in pattern matrix
                do
                        for each char in row of pattern matrix:
                        do
                                if pattern[row][char] not equals text[row][char]:
                                do
                        set flag as false
                    done
                done
            done
            if flag is true
                pattern found

        done
done
End
```

2.2.2 ZHU TAKAOKA ALGORITHM

The general scheme of this algorithm is to use the hash function method proposed in the RK algorithm *vertically*. We first translate the two-dimensional arrays of characters of TEXT and PATTERN to one-dimensional numbers TEXT' and PATTERN' respectively; we then search TEXT' for the occurrences of the PATTERN' using the KMP technique row by row.

*Time complexity is O(n * n + m * m)* for either of the worst case and average case since the number of multiplications during the preprocessing stage for the hash function calculation is *O(n * n + m * m)* and the largest number of comparisons made in KMP 1 procedure is O((n - m) * n).

Here Rolling Hash Function is used. The hash function is defined as follows:
h(k) = k mod q

Where q is a large prime.

The number k corresponding to the M-character section TEXT [i]. . . TEXT[i + M - 1] is:

$$k = \mathrm{ord}(TEXT[i]) * d^{M-1} + \mathrm{ord}(TEXT[i+1]) * d^{M-2} + \ldots + \mathrm{ord}(TEXT[i+M-1])$$

where ord (x) is the order of character x. Shifting one position to the right in the text simply corresponds to replacing k by

```
(k - ord(TEXT[i]) * d^(M-1))
  * d + ord(TEXT[i + M]).
```

The Algorithm:

INPUT: The text and pattern are stored in the TEXT[l. . n,1 . . n] and PATTERN[l. . m,
1 . . m], respectively.

OUTPUT: Location at which pattern occurs.

```
Begin
texthash();  //text is hashed using the Hash Function explained above
row := m;
dm := 1.
for j := 1 to m - 1
do
        dm' := (d * dm) mod q;
read(PATTERN);
pathash(); //pattern is hashed using the Hash Function explained above
repeat
        KMP (PATTERN',TEXT', found,column);
        if (found = false) and (row < n) then change(row);
        row := row + 1;
until(found = true) or (row > n);
if found = true
        then writeln
        ('The beginning of the pattern is found at',row - n + 1, column - n + 1)
else writeln
        ('not found')
End
```

*Time complexity is O(n * n + m * m)*

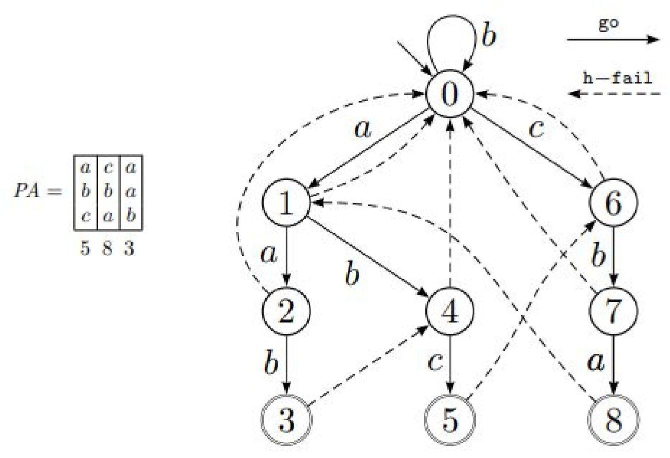
*Space complexity is O(n * n + m * m)*

2.2.3 BAKER-BIRD ALGORITHM

The Bird and Baker’s solution uses standard Aho-Corasick algorithm of pattern matching for a set of patterns. This allows construction of deterministic machine with associated output actions in every state of the Aho Corasick pattern matching machine and therefore direct conversion of text array TA to TA′ by writing state identifiers into the original text array TA. Let us consider an example of the idea presented above. Let PA be pattern array and TA be text array, respectively.

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

AC machine obtained on PA is as follows:

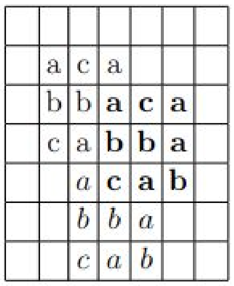


Each string in TA (horizontally) is iterated over AC machine, once the pointer reaches the end node i.e. strings for pattern and text match, state identifier is updated in TA'. Resulting TA' formed is as follows:

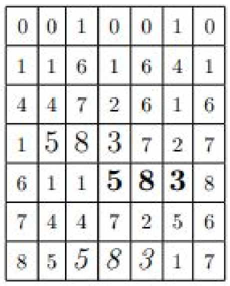
$$TA= \begin{array}{|c|c|c|c|c|c|c|}\hline b & b & a & b & b & a & b \\ a & a & c & a & c & b & a \\ b & b & b & a & c & a & c \\ a & c & a & b & b & a & b \\ c & a & a & c & a & b & a \\ b & b & b & b & a & c & c \\ a & c & c & a & b & a & b \\ \hline\end{array} \qquad TA'= \begin{array}{|c|c|c|c|c|c|c|}\hline 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 6 & 1 & 6 & 4 & 1 \\ 4 & 4 & 7 & 2 & 6 & 1 & 6 \\ 1 & 5 & 8 & 3 & 7 & 2 & 7 \\ 6 & 1 & 1 & 5 & 8 & 3 & 8 \\ 7 & 4 & 4 & 7 & 2 & 5 & 6 \\ 8 & 5 & 5 & 8 & 3 & 1 & 7 \\ \hline\end{array}$$

Now this updated TA' is sent column by column to and compared with the pattern through KMP.

Occurence of Pattern in Text will be visualized as:



	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

The Algorithm:

INPUT: The text and pattern are stored in the TEXT[l. . n,1 . . n] and PATTERN[l. . m, 1 . . m], respectively.

OUTPUT: Location at which pattern occurs.

1. Row Matching Step:
   Give a number to each distinct row.
   Build an n × n array R.
   Computation of R (Aho-Corasick) :
   - Build the data structure (trie + failure function) with the rows of PA. Time O(| Σ |m2).
   - For each row of T, run the AC search algorithm. Time O(n) for each row; overall O(n*n).

2. Column-matching step:
   Given R, find all occurrence positions.
   We need to check if all rows of P appear vertically.
   - Let P ′ = r(p1)r(p2)· · · r(pm).
   - For each column of R, run the KMP algorithm with P ′ as the pattern.

*Total Time Complexity: O(m*m + n*n)*
*Total Space Complexity: O(n*n + m)*

2.2.4 BAEZA-YATES RÉGNIER ALGORITHM

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 occurrence positions in such a way that no occurrence is missed. Suppose there is a match of one of the strings from the dictionary, i.e. one complete row, say lth, of the two-dimensional pattern has been found at position x, y, where $m \le x \le n$, $m \le y \le n$.

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, similarly to Bird and Baker algorithm.

The Algorithm:

INPUT: The text and pattern are stored in the TEXT[l. . n,1 . . n] and PATTERN[l. . m, 1 . . m], respectively.

OUTPUT: Location at which pattern occurs.

```
for row = 1 to m
do
    addString(dictionary,PA[1..m, row])
end

createAC(dictionary)

nextline = false
for row = m − 1 to row ≤ n, step m
do
    for i = 1 to i ≤ m
    do
        if ACsearch(row, i)
        then
            for j = 1 to j < i
            do
                if not ACsearch(row − i + j, j)
                then
                    nextline = true ;
                    break
            end
            for j = 1 to j ≤ m − i + 1
```

```
                              do
                                        if not ACsearch(row + j, j)
                                        then
                                                  nextline = true ;
                                                  break
                                        end
                                        if not nextline
                                        then printMatch(i, j)
                                        else
                                                  nextline = false
                                        else
                                                  break
                                        end
          end
end
```

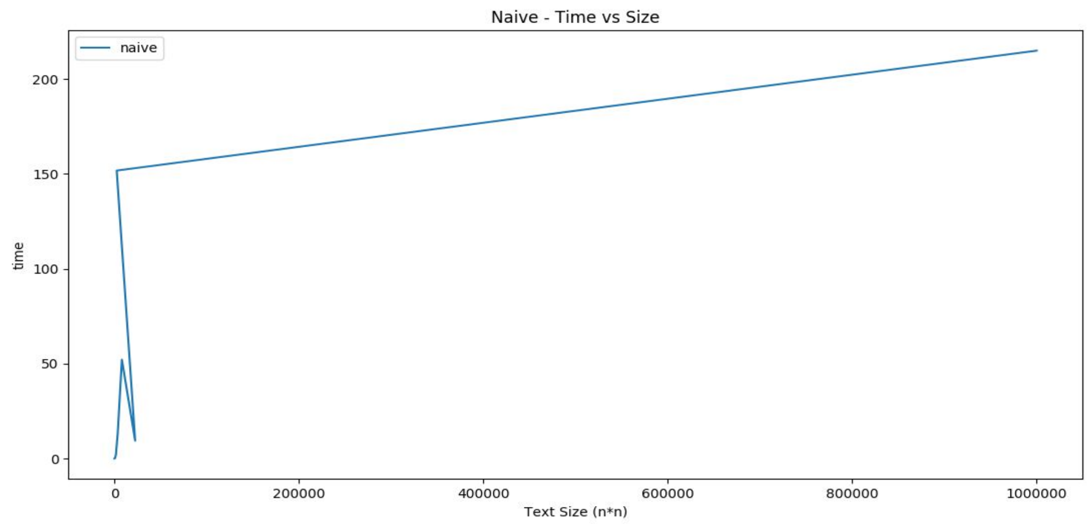
*Total Time Complexity: O(m*m + n*n)*
*Total Space Complexity: O(n*n + m)*

3. EXPERIMENTAL RESULTS

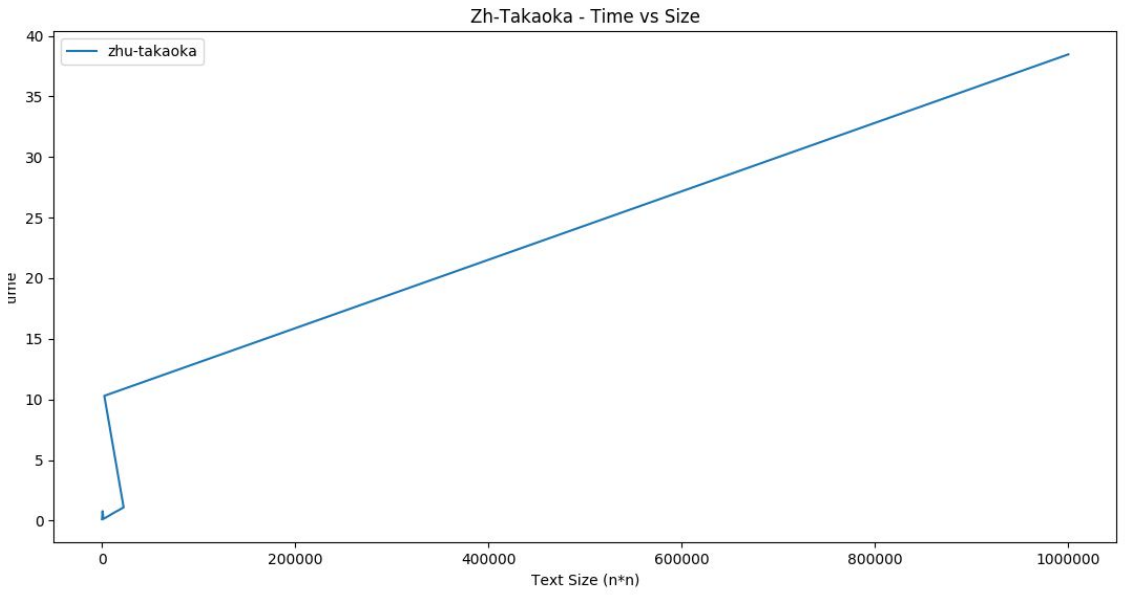
We have plotted our observations of the running times of the various algorithms in the form of a graph. First, we have a graph for each of the four algorithms that'll show the variation in running time against the different sizes of square text matrices (n*n).

Then one final graph is plotted for all the four algorithms; a clear observation can be made from the graph regarding their efficiency (in terms of time taken to find all occurrences of pattern) as the input size increases.

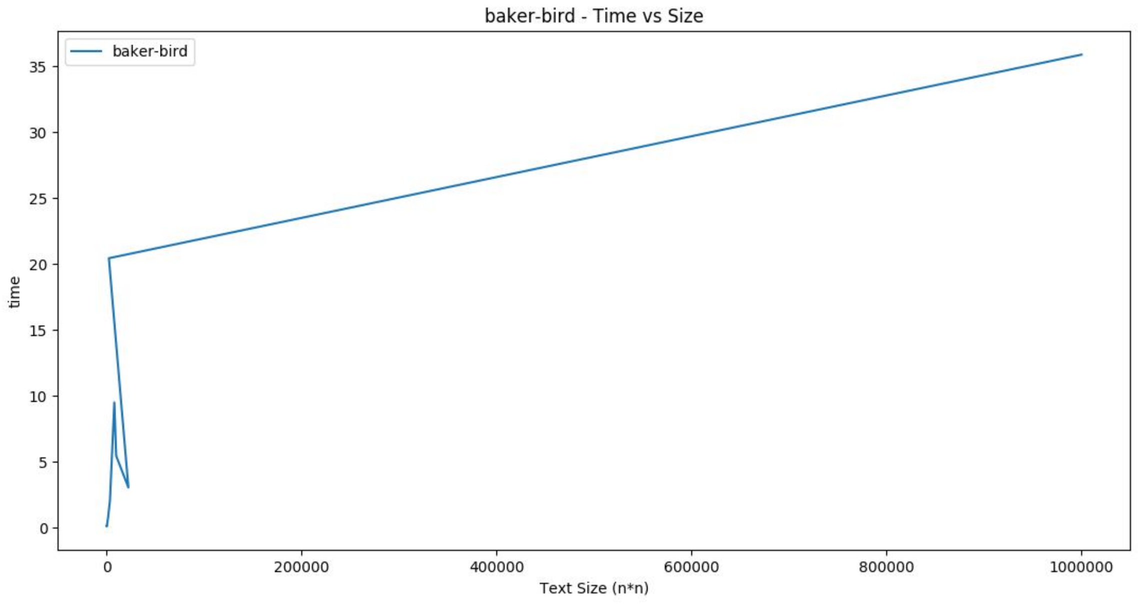
3.1.1 NAIVE



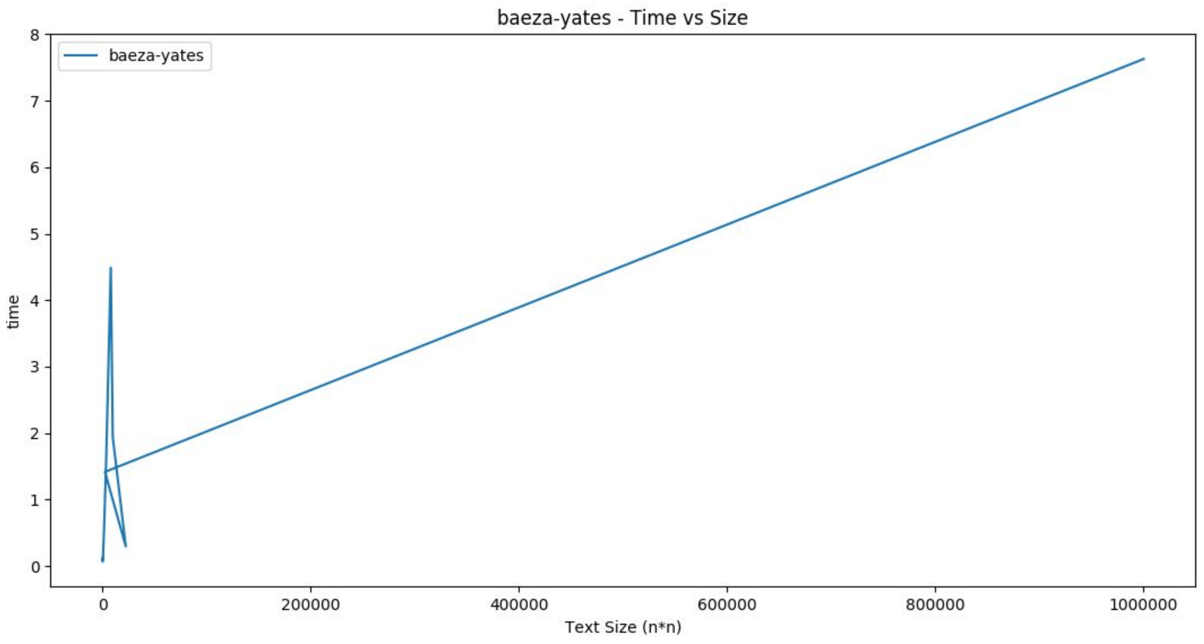
3.1.2 ZHU-TAKAOKA



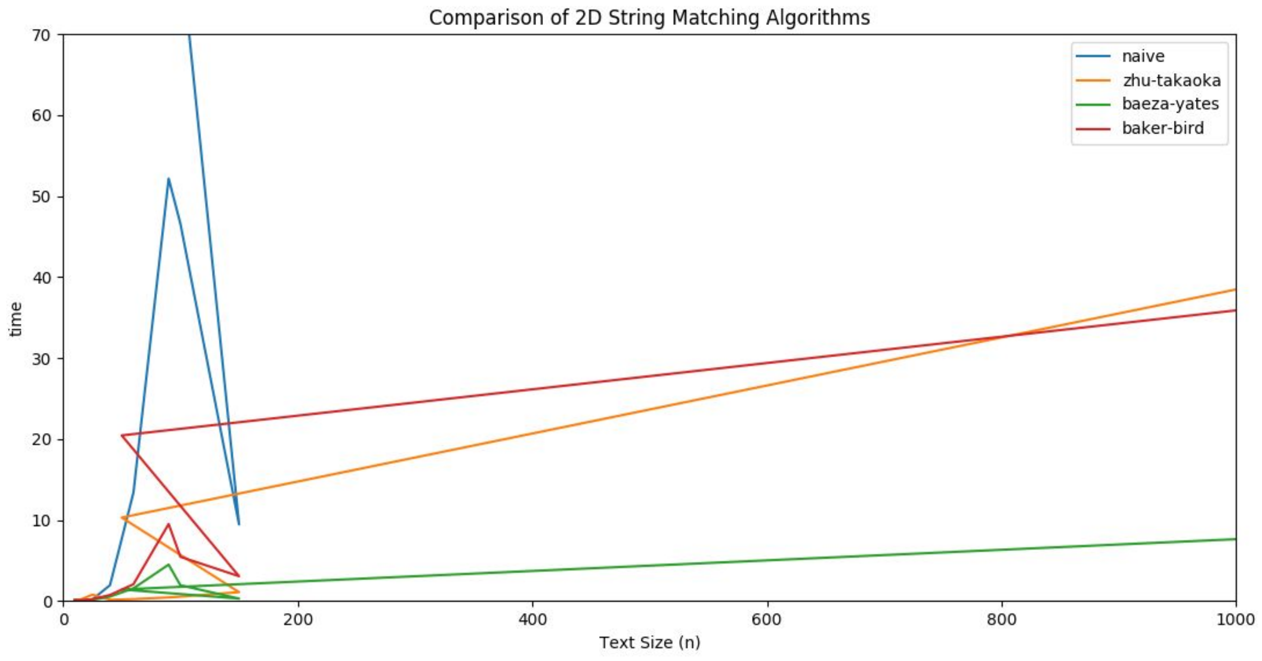
3.1.3 BAKER-BIRD



3.1.4 BAEZA-YATES REGNIER



3.1.5 COMPARISON OF ALL FOUR



4. CONCLUSION & REFERENCES

Following conclusions can be drawn from the above experimental results:

- ❏ Naive algorithm with time complexity O(m*n*m*m) performs the worst in all cases.
- ❏ Although all the three algorithms, namely Zhu-Takaoka, Baker-Bird, Baeza-Yates Regnier, have the same time complexity of O(n*n+m*m), but with increase in size of the text matrix, Baeza-Yates outperforms the other two giving the best performance.

Thus, we can conclude that for larger input sizes Baeza-Yates Regnier would give best worst case time complexity.

REFERENCES

[1]https://www.researchgate.net/publication/322765507_Efficient_Algorithm_for_Two_Dimensional_Pattern_Matching_Problem_Square_Pattern

[2]https://ieeexplore.ieee.org/document/6216622

[3]https://www.uio.no/studier/emner/matnat/ifi/INF3800/v13/undervisningsmateriale/aho_corasick.pdf

[4]https://www.tutorialspoint.com/Rabin-Karp-Algorithm

[5]https://www.tutorialspoint.com/Knuth-Morris-Pratt-Algorithm

[6]http://www.stringology.org/papers/Zdarek-PhD_thesis-2010.pdf

[7]http://ocw.snu.ac.kr/sites/default/files/NOTE/4047.pdf

[8]https://www.tutorialspoint.com/Aho-Corasick-Algorithm