STRING MATCHING ALGORITHMS AND ITS APPLICATIONS

In this paper, we are going to describe about different string matching algorithms and their real-world applications. For each algorithm, we described their algorithm, time complexities, and space complexities. We did analysis of different types of string matching algorithms on random text taken from Wikipedia and search for pattern in it. Then we conclude that which algorithm is best in which cases. We have also implemented the code of the algorithms and experimented it in different conditions.

INTRODUCTION

String matching is a special case of pattern matching, where the pattern is described by a finite sequence of symbols (or alphabet ∑). It consists of finding one or more generally all the occurrences of a short pattern P=P[0]P[1]…P[m-1] of length m in a large text or sequences T=T[0]T[1]…T[n-1] of length n, where m, n>0 and m <= n. Both P and T are built over the same alphabet ∑. ∑ may be usual human alphabet (A to Z), or other applications may use binary alphabet (∑ = {0,1}), or DNA alphabet (∑ = {A,C,G,T}) in bioinformatics. For Example:

If we have DNA sequences like: "AACGTAACGTTCAACGTAATTTAACGTAACCGT". We want to find pattern "AACGT" in these sequences. So, we got four patterns at index 0,5,12,22.

If we have binary file sequences like: "00011010110100110011101". We want to find a file in this binary sequence whose pattern is "1101". So, we got three patterns at index 3,8,18.

Now, String Matching Algorithm can be divided into two matching algorithms: Exact String Matching and Approximate String Matching. We will discuss these matching in detail. Then we will be able to answer the question which of these algorithms is the best. To achieve this, we implement, test, and compare the efficiency of each algorithms. The comparison will be executed in different situations: small and large alphabet, the pattern might appear zero time, once, a few times or many times in the text depending to its length. Then we will see its applications in real world in detail.

EXACT STRING MATCHING ALGORITHMS

Exact string matching algorithms is to find one, several, or all occurrences of a defined string (pattern) in a large string (text or sequences) such that each matching is perfect. This means all alphabets of patterns must be matched to corresponding matched subsequence. These are further classified into four categories:

1. Algorithms based on character comparison.
2. Deterministic finite automation (DFA) method
3. Algorithms based on Bit – parallelism method
4. Hashing-string matching algorithms.

ALGORITHMS BASED ON CHARACTER COMPARISON

1.NAIVE ALGORITHM

This algorithm could be considered the simplest string matching algorithm, since it performs character comparisons between the scanned text substring and the complete pattern from left to right. It requires no pre-processing phase and no extra space.

Algorithm:

Slide the pattern over text one by one and check for a match. If first alphabet match is found, then check for next alphabet match until we get mismatch or complete match.

Time complexities:

a) Best case: O(n) b) Worst case: O(m*(n-m+1))

Where 'm' is length of pattern and 'n' is length of text.

Space complexities:

O(1), since no extra space is required.

2.KMP (KNUTH MORRIS PRATT) ALGORITHM

Since Naïve approach doesn't work well in many situations. KMP algorithm is better as compared to naïve approach. 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 next window. We take advantage of this information to avoid matching the characters that we know will anyway match. for example: Consider a text **"**AAAAABAAABA" and we want to find pattern "AAAA" in it.
We compare the first window of text with pattern:
Text = "**AAAA**ABAAABA"
Pattern = "**AAAA**". we find a match. This is same as Naïve algorithm.
In next step, we compare next window of text with pattern.
Text= "**AAAA**ABAAABA"
Pattern = "**AAA**A" [pattern shifted one position]
This is where KMP does optimization over naïve. In this second window, we only compare fourth A of pattern with fourth alphabet of current window of text to decide whether current window matches or not. Since we know first three alphabets will anyways match, we skipped matching first three alphabets. Now how many alphabets we need to be skipped. To know this, we pre-process pattern and prepare an integer **array LPS [ ]** that tells us count of alphabets to be skipped.
LPS [i**]** = Longest Proper Prefix of pattern[0….i] which is also Suffix of pattern[0…i].

Algorithm:

We start comparison of pattern[j] with j=0 with alphabet of current window of text. We keep matching alphabets text[i] and pattern[j] and keep incrementing I and j while pattern[j] and text[i] keep matching. When we see a mismatch: We know that alphabets pattern[0..j-1] match with text[i-j+1…i-1] (Note that j starts with 0 and increment it only when there is a match). We also know that LPS[j-1] is count of alphabets of pattern[0…j-1] that are both proper prefix and suffix. So, we can conclude that we do not need to match these LPS[j-1] characters with text[i-j…i-1] because we know that these characters will anyway match.

Time Complexities:

Pre-processing time: Θ(m)
Matching time: Θ(n)

Space complexities:

Θ(m), to construct LPS array.

3.BOYER MOORE ALGORITHM

Boyer Moore is considered the basic and the best algorithm for single pattern matching algorithms. BM algorithm matches pattern suffix from right to left and it maintains two heuristics in the case of mismatch. The first, called bad character heuristic and the second, called good suffix heuristic, KMP algorithm does pre-processing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does pre-processing for the same reason. It pre-processes the pattern and creates different arrays for both heuristics.

Algorithm:

Compare pattern to text, starting from the rightmost characters (alphabets) in pattern. If matches then we slide to next position to compare next characters (alphabets) in pattern. When mismatch occur, you slide the pattern by more than one. At every step, it slides the pattern by max of the slides suggested by the two heuristics**.** So, it uses best of the two heuristics at every step**.**

BAD CHARACTER HEURISTICS:

The character of the text which doesn't match with the current character of pattern is called the Bad Character. Upon mismatch, we shift the pattern until –

1) The mismatch become a match
2) Pattern P move past the mismatch character.

In this case, Pre-processing steps proposed shift by i-j, with O(1) lookup time and O(n) space.

GOOD SUFFIX HEURISTICS:

Let **s** be substring of text **T** which is matched with substring of pattern **P**. Now we shift pattern till:

1) Another occurrence of s in P matched with s in T.

2) A prefix of P, which matches with suffix of t 3) P moves past 's'.

In this case, it requires two tables: one for use in the general case, and another for use when either the general case returns no meaningful result or a match occurs. These tables will be designated L and H respectively. Both table are constructible in O(n) time and use O(n) space. The alignment shift for index i in P is given by n-L[i] or n-H[i]. H should only be used if L[i] is zero or match has been found.

Time Complexities:

Pre-processing time: Θ(m+k)

Matching time: a) best case: Ω(n/m) b) Worst case: O(mn)

Where m is length of pattern, n is length of text.

Space Complexities:

Θ(m), to store shift during pre-processing.

4.USING TRIE DATA STRUCTURE

All the above discussed algorithm pre-processes the pattern to make matching algorithm faster. In this case we will pre-processes the text. After pre-processing text (building suffix tree of text), we can search pattern very efficiently as compared to above algorithms. But this algorithm is good for fixed text or text which is not change frequently. A suffix Tree for a given text is a compressed trie for all suffixes if the given text.

Build a Suffix Tree for a given text

1) Generate all suffixes of given text.

2) Consider all suffixes as individual words and build a compressed trie.

Algorithm

Starting from the first character of the pattern and root of Suffix Tree, do following for every character. For the current character of pattern, if there is an edge from the current node of suffix tree, follow the edge. If there is no edge, then no pattern will be found and return. If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern and hence pattern will be found. Every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes.

Time Complexities:

Pre-processing time: Θ(n)

Matching time: Θ(m+k)

Where m is length of pattern, n is length of text and k is the occurrence of pattern.

Space Complexities:

Depends on text. Space required to make suffix tree.

DETERMINISTIC FINITE AUTOMATON METHOD

1.AUTOMATON MATCHER ALGORITHM

In FA based algorithm, we pre-process the pattern and build a 2D array that represents a Finite Automata. Construction of the FA is the main tricky part of this algorithm. Once the FA is built, the searching is simple. We can make FA using idea similar to LPS array construction in KMP algorithm.

CONSTRUCTION OF FA:

1) Fill the first row. All entries in first row are always 0 except the entry for pattern[0] character. For pattern[0] character, we always need to go to state 1.

2) Initialize LPS as 0. LPS for the first index is always 0.

3) Do following for rows at index i = 1 to M. (M is the length of the pattern)

a) Copy the entries from the row at index equal to LPS.

b) Update the entry for pat[i] character to i+1.

c) Update LPS "LPS = TF[LPS][pat[i]]" where TF is the 2D array which is being constructed

Algorithm:

Since we constructed Finite Automaton 2D array using above process. Now, in search, we simply need to start from the first state of the automata and the first character of the text. At every step, we consider next character of text, look for the next state in the built FA and move to a new state. If we reach the final state, then the pattern is found in the text.

Time Complexities:

Pre-processing Time: O(m*k)

Matching Time: O(n)

Where m is length of pattern, n is length of text and k is No_Of_Characters.

Space Complexities:

O((M+1)*(No_Of_Characters)) , to build Finite Automaton 2-D Matrix.

ALGORITHM BASED ON BIT PARALLELISM METHOD

Bit parallelism uses the essential parallelism of the bit manipulations inside computer words to perform many operations in parallel.

1.AHO-CORASICK ALGORITHM

It is a kind of dictionary-matching algorithm that locates elements of a finite set of strings (the "dictionary") within an input text. It matches all the strings simultaneously. For example:
Text = “ahishers”. We want to search the pattern “he”, “she”, “hers”, “his”. Then AC algorithm will simultaneously process all query and give the result. If we use linear time searching like KMP, then we need to one by one search all pattern in text. This makes KMP expensive as compared to AC algorithm. This algorithm formed the basis of the original Unix command fgrep. The algorithm constructs a finite state machine that resembles a trie with additional links between the various internal nodes.

Preprocessing:

Build an automaton of all words in input array. The automaton has mainly three functions:
GO TO: This function simply follows edges of Trie of all word in input array. It is represented as 2D array g[][] where we store next state for current state and character.
FAILURE: This function stores all the edges that are followed when current characters doesn’t have edge in trie. It is represented as 1D array f[] where we store next state for current state.
OUTPUT: Stores index of all words that end at current state. It is represented as 1D array o[] where we store indexes of all matching words as a bitmap for current state.

Algorithm:

Matching steps is to traverse the given text over built automaton to find all matching words. We first Build a Trie of all words. This part fills entries in goto g[][] and output o[].Next we extend Trie into an automaton to support linear time matching. This part fills entries in failure f[] and output o[].We build Trie. And for all characters which don’t have an edge at root, we add an edge back to root. For a state s, we find the longest proper suffix which is a proper prefix of some pattern. This is done using Breadth First Traversal of Trie. For a state s, indexes of all words ending at s are stored. These indexes are stored as bitwise map (by doing bitwise OR of values).This is also computing using Breadth First Traversal with Failure.

Time Complexities:

O(n+m+z), where n and m are length of text and pattern, z is total number of occurrence of words. If we use KMP then it takes O(n*k + m), where k is number of words to be search.

Space Complexities:

Space needed to store goto, failure and output functions.

HASHING STRING MATCHING ALGORITHM

1.RABIN KARP ALGORITHM

The Rabin-Karp algorithm uses a totally different approach to solve the string matching problem. This method is based on hashing techniques.

Algorithm

Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So, Rabin Karp algorithm needs to calculate hash values for following strings:

1.Pattern itself.

2.All the substrings of text of length m.

Rehashing of text window done using the following formula:

$$hash(text[s+1....s+m]) = d\,(hash(text[s....s+m-1]) - txt[s]*h) + txt[s+m])\bmod q$$

where, hash(text[s….s+m-1]): Hash value at shift s
hash(text[s+1……s+m]): Hash value at next shift
d: Number of characters in alphabet
q: A prime Number
h: d^(m-1)
m: Length of pattern

Time Complexities:

Pre-processing time: O(m)
Compute Hashing in: O(1)
Matching time: a) Best and Average case: O(n+m) b) Worst case: O(nm)

Space Complexities:

O(1), since pre-processing involves hashing of Pattern which does not require extra space.

APPROXIMATE STRING MATCHING ALGORITHMS

Approximate string matching (also known as fuzzy string searching) is the technique of finding substrings of a text string that match a pattern approximately (rather than exactly). More specifically, the approximate string matching approach is stated as follows: Let a given alphabet Σ, text T[1…n] and pattern P[1...m], we are supposed to find all the occurrences of pattern in the text whose edit distance to the pattern is at most K. The edit distance between two strings is defined as minimum number of character insertion, deletion and replacements needed to make them equal. Some of the well-known edit distances are: Levenshtein edit distance and Hamming edit distance.

Hamming Distance: is the number of positions with mismatching characters between two strings of equal length. So, its perform substitution only. We call the approximate string matching algorithm with d Hamming distance string matching with k mismatches.

Levenshtein Distance: is the minimum number of character insertions, deletions and substitutions that required transforming of one string to the other. We call the approximate string matching algorithm with d Levenshtein distance string matching with k differences.

The reasons for introducing approximate string matching are: low quality of text, heterogeneousness of databases, spelling errors in the pattern or text, finding DNA subsequence's after mutation, etc. Some of the important approximate string matching algorithms are:

1.BRUTE-FORCE ALGORITHM

This algorithm could be considered the simplest string matching algorithm, since it performs character comparisons between the scanned text substring and the complete pattern from left to right. It requires no pre-processing phase and no extra space to count the number of mismatches found.

Algorithm:

compute the edit distance to P for all substrings of T. If more than k has been found, means we have not found pattern and shifts exactly one position to the right to compare next text window.

Time Complexities:

Matching time: $O(n^3m)$

Space Complexities:

O(1), since no pre-processing is required.

2.SELLERS ALGORITHM (DYNAMIC PROGRAMMING)

This is better approximate matching algorithm as compared to brute-force. It used concept of dynamic programming.

Algorithm:

For each position j in the text T, and each position i in the pattern P, go through all substrings of T ending at position j, and determine which one of them has the minimal edit distance to the i first characters of the pattern P. Compute the minimal distance as E(i, j).After computing E(i, j) for all i and j, we can easily find a solution to the original problem: it is the substring for which E(m, j) is minimal (where m being the length of the pattern P).Computing E(m, j) is very similar to computing the edit distance between two strings. we can use the Levenshtein distance computing algorithm for E(m, j), the only difference being that we must initialize the first row with zeros, and save the path of computation, that is, whether we used E(i − 1,j), E(i,j − 1) or E(i − 1,j − 1) in computing E(i, j). In the array containing the E(x, y) values, we then choose the minimal value in the last row, let it be $E(x_2, y_2)$, and follow the path of computation backwards, back to the row number 0. If the field we arrived at was $E(0, y_1)$, then $T[y_1 + 1] ... T[y_2]$ is a substring of T with the minimal edit distance to the pattern P.

Time Complexities:

Running time: O(mn) ,to create DP table using Tabulation.

Backward working phase: O(n+m)

Space Complexities:

O(m*m) , to create DP Table.

3.SHIFT-OR ALGORITHM (BITAP ALOGRITHM)

This algorithm is based on bit-parallelism method. The bitap algorithm is perhaps best known as one of the underlying algorithms of the Unix utility agrep. To perform fuzzy string searching using the bitap algorithm, it is necessary to extend the bit array R into a second dimension. Instead of having a single array R that changes over the length of the text, we now have k distinct arrays $R_{1...k}$. Array R_i holds a representation of the prefixes of pattern that match any suffix of the current string with i or fewer errors. An "error" may be an insertion, deletion, or substitution.

Algorithm

The algorithm searches a pattern in a text (without errors) by parallelizing the operation of a nondeterministic finite automaton that looks for the pattern. It is treat mismatches by counting k differences using a counter of size log2, specifically, the bigger the number of bits needed to represent individual states, the smaller the length of patterns that are considered.

Time Complexities:

Running time: O(mn), where m is length of pattern and n is length of Text.

Space Complexities:

O(m*k) where m is length of pattern and k is number of words to be search in parallel.

UPDATED AND HYBRID STRING MATCHING ALGORITHMS

In the last decade, more than 50 new algorithms have been proposed for the string matching approach. These algorithms are either a kind of variations of the previously described algorithms or a hybrid form that combines the features of these algorithms. Some of them are discussed below in brief:

UPDATED STRING MATCHING ALGORITHM

1.FAST SEARCH ALGORITHMS

These are a family of algorithms that consists from three different variants of the Boyer-Moore algorithm. The general base of these algorithms that at the end of each attempt the shift is computed with the bad character rule only if the first comparison of the attempt is a mismatch and the shift is computed using the good suffix rule otherwise. The first algorithm is Fast-Search (FS) algorithm that compares the pattern with the current window characters from right to left at each attempt the pattern is compared with the current window characters from right to left. Then the shift is computed using the Horspool bad-character rule if and only if a mismatch occurs during the first character comparison, otherwise the algorithm uses the good-suffix rule. The second algorithm from this family is the Backward-Fast-Search (BFS) algorithm. The algorithm benefits from combining the standard good-suffix rule with the bad-character rule to get the backward good suffix rule. Finally, the Forward-Fast-Search (FFS) algorithm 2004, which preserve the same structure as the Fast-Search algorithm, but it uses a look-ahead character to determine larger shift advancements called forward good-suffix rule.

HYBRID STRING MATCHING ALGORITHM

1.HYBRID MULTITHREADED ALGORITHM

This algorithm based on two well-known multiple pattern matching algorithms Wu-Manber and Aho-Corasick. Where the algorithm benefits from wu-manber power in matching long patterns and Aho-Corasick for short patterns. It divide the patterns between the two algorithms to keep the workloads balanced for optimal performance. Additionally, multiple threads are used to maximize the performance of the hybrid algorithm.

2.FRANEK JENNINGS SMYTH ALGORITHM

It is a hybrid algorithm that combines the linear worst-case time complexity of Knuth-Morris-Pratt Algorithm and the sublinear average behaviour of Quick-Search algorithm. Each attempt of the search is divided into two phases. In the first phase, as with the Quick-Search approach, the FJS algorithm first compares the rightmost character of the pattern with its corresponding character in the text, if a mismatch occurs, a Quick-Search shift is used, when a match is found the FJS algorithm invokes the second step. Otherwise another Quick-Search shift occurs. The second phase of the algorithm consists in a Knuth-Morris-Pratt pattern-matching starting from the leftmost character and, if no mismatch occurs, then whether or not a match is found, a Knuth-Morris-Pratt shift is performed followed by a return to the first step.

EXPERIMENTAL ANALYSIS

We have implemented the above algorithm in c++ language. We experimented different types of algorithms on random text taken from Wikipedia. The text name is Machine Learning. We have made some of the patterns to search via these algorithms and did analysis.

Input Text: Text of Machine Learning from Wikipedia consists of above 12,000 characters.
Input Patterns: "the", "Machin", "machine learning", "artificial intelligence", "computers".

We did analysis on the basis of length of different patterns. Corresponding lengths are 3,6,16,23,9. Below BM is Boyer Moore, DFA is Deterministic Finite Algorithm, KMP is Knuth Morris Pratt Algorithm, RK is Rabin Karp algorithm. AC is Aho-corasick Algorithm.

Algorithm	Length(3)	Length(6)	Length(9)	Length(16)	Length(23)
Naïve(Brute)	**0.173**	**0.197**	**0.15**	**0.145**	**0.176**
KMP	**0.153**	**0.173**	**0.113**	**0.109**	**0.088**
BM	**0.198**	**0.066**	**0.04**	**0.046**	**0.024**
RK	**0.343**	**0.351**	**0.349**	**0.346**	**0.353**
DFA	**0.131**	**0.183**	**0.243**	**0.331**	**0.541**

NOTE: Each time is in Seconds.

- AC algorithm searches all the pattern simultaneously and take very less time as compared to total time taken by each algorithm. **It takes 0.267 seconds.**

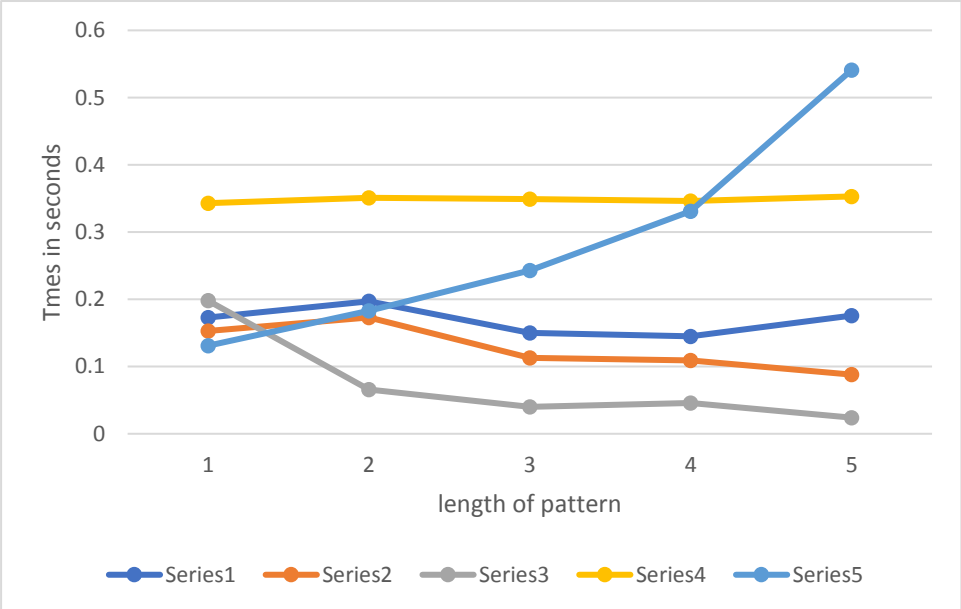

FIGURE: Plot of different algorithms with different pattern length
Series1: Brute force Series2: KMP Series3: BM Series4: RF Series5: DFA

CONCLUSION

- From above result, we can conclude that Deterministic finite algorithm is many times better as compared to algorithm based on characters comparison.
- Among characters comparison algorithm:
- we can conclude that the best algorithm in majority of cases is Boyer-Moore. More the pattern is long, more its advantage become significant, the reason to this can be easily explained by the fact that it could skip more characters in this case, its complexity is sublinear: O(N/M). But we also notice that when the pattern is very small, the advantage of BM algorithm disappears completely.
- Knuth-Morris-Pratt and the naive algorithm obtain similar results, but the result of KMP is always a bit better. KMP could be a good choice if the length of pattern is very short. Rabin-Karp obtains very good results in general but due to collision phenomena in hashing it sometimes end up with bad time complexity. So, it depends on nature of input text. But generally, Rabin Karf results are a lot better than KMP and the naive solution and it is definitely the best choice in the situations where Boyce-Moore is not adapted.
- In general, for algorithm based on characters comparison, we also notice that the results with a bigger alphabet is always better than a small one. The reason is, because with a small alphabet we have more similarity between the pattern and substrings of the text even there are not matches, but these similarities require more comparisons.
- In case of Approximate algorithm, we see how much a pattern is approximately equal to text window. And main optimization is based on minimizing edit distance.
- Some algorithm based on bit-parallelism have also a better advantage over single step algorithms that it processes the whole query simultaneously and take less time as compared to all algorithms.
- The query for which text is not changing frequently, it is best implemented using suffix tree because in this case matching time depends on length of pattern and hence fast.
- Several Many algorithms are discovered based on updated and hybrid of two or more string matching algorithms, which improves the complexity of the program.

APPLICATIONS

Applications in information technology includes web search engines, spam filters, natural language processing, computational biology (search of pattern in DNA sequence), feature detection in digital images and so on. We will describe some of its applications.

1.SPELL CHECKER

In spell checkers, we build a “trie” of pre-defined set of patterns. This trie is used for the string matching means if any such pattern occurs then it shows the occurrence by reaching to its final states. Spell checker basic module is shown in figure below:

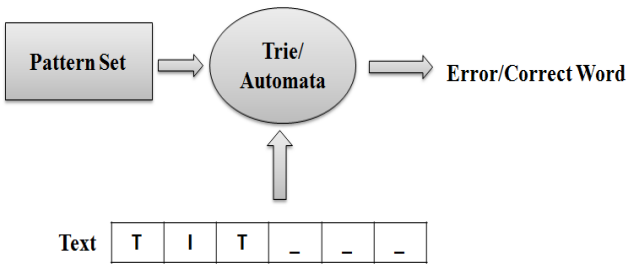

Figure: Spell Checker

2.SPAM FILTERS/SPAM DETECTION SYSTEM

Unsolicited and unwanted emails called spam that engages lots of network bandwidth. This will cause great financial loses. All spam filters use the concept of string matching to identify and discard the spam. Spam filter searches suspected signature patterns in the content of email by applying string matching. All content based filters are worked on string matching. Spam filter basic structure is shown in figure:

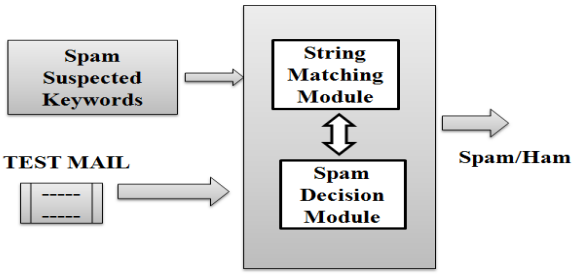

Figure: Spam Filter

3.INTRUSION DETECTION SYSTEM

In Intrusion Detection System, data packets that contain intrusion related keywords are found by applying string matching strategy. All the malicious code is stored in the database and every incoming data is compared with stored data. If match found then alarm is generated. It is based on exact string matching algorithms where we should capture each and every intruded packet and they must be detected. The Intrusion detection system modal is shown in figure:

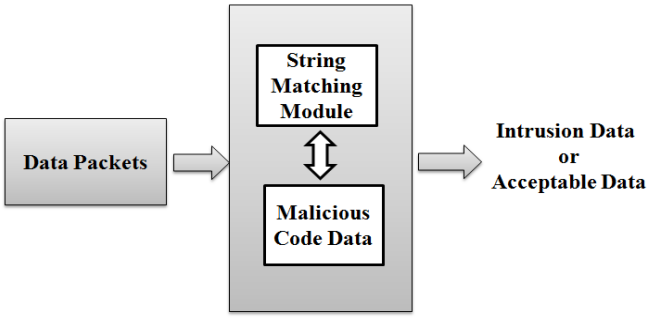

Figure: Intrusion Detection System Model

4.SEARCH ENGINE/CONTENT SEARCH IN LARGE DATABASE

Most of the data are available on internet in the form of textual data. Due to the large quantity of uncategorized text data, it becomes really difficult to search a particular content. Web search engines help us to solve this problem by organizing the required text / data as efficiently as possible. To categorize these data string matching algorithms are used. Categorization is done based on search keywords. Figure shows the basic model of Search Engine.

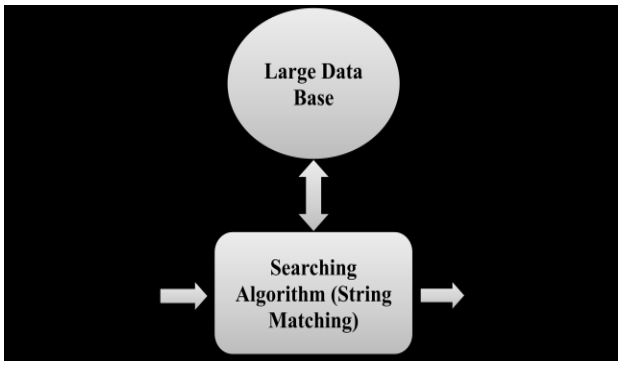

Figure: Search Engine Module

5.COMPUTATIONAL BIOLOGY/DNA SEQUENCING

Bioinformatics is the application of information technology and computer science to biological problems, in perspective to the issues involving genetic sequences and in order to find the DNA patterns, string matching module and DNA analyser both works with collaboration for finding the occurrence of the pattern set. Figure shows the Bioinformatics DNA Sequencing Module.

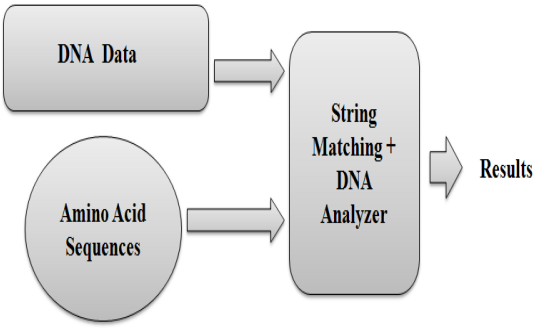

Figure: DNA Sequencing Module

6.PLAGIARISM DETECTION

Copy someone work and claim it as own is called as Plagiarism. So, with the use of string matching we can compare the texts and detect the similarities between them. On the basis of these similarities declare whether it is original work or taken from somewhere else. Figure shows the Plagiarism detection technique.

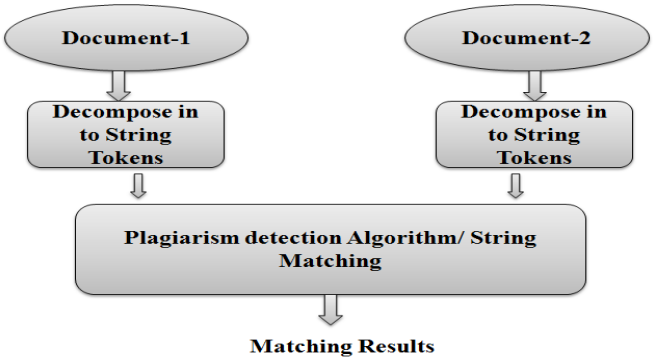

Figure: Plagiarism Detection System

7.INFORMATION RETRIEVAL

In text mining task designed to extract previously unknown information by analysing large quantities of text. String matching plays very vital role here like as information extraction, topic tracking, question answering etc. Figure shows the basic structure of information retrieval system.

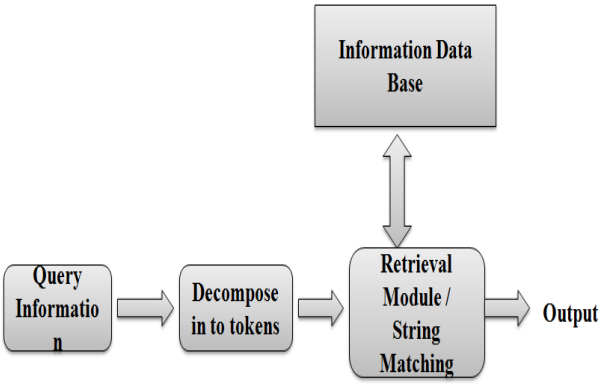

Figure: Information Retrieval Modal