Multiple Pattern Matching Algorithm using Pair-count

Raju Bhukya1, DVLN Somayajulu 2

1 Dept of CSE, National Institute of Technology,
Warangal, A.P, India. 506004.

2 Dept of CSE, National Institute of Technology,
Warangal, A.P, India. 506004.

Abstract

Pattern matching occurs in various applications, ranging from simple text searching in word processors to identification of common motifs in DNA sequences in computational biology. The problem of exact pattern matching has been well studied and a number of efficient algorithms already exist. However these exact pattern matching algorithms are of little help when they are applied to finding patterns in DNA sequences. Pattern matching in a DNA sequence or pattern searching from a large data base is a major research area in computational biology. To extract pattern from a large sequence it takes more time, in order to reduce searching time we have proposed an approach that reduces the search time with accurate retrieval of the matched pattern from the given sequence of any size of a file. Executing patterns from a large DNA or protein data is a computationally intensive task. As performance plays a major role in extracting patterns from a given DNA sequence or from a large database independent of the size of the sequence. More efficient approaches related to multiple pattern matching techniques are becoming more important for finding the functional as well as the structural properties of the proteins and genes. One of the major problems in genomic field is to perform pattern comparison on DNA and protein sequences. In the current approach we explore a new technique which avoids unnecessary comparisons in the DNA sequence and gives the accurate retrieval of the pattern called a multiple pattern matching algorithm using pair count. The proposed technique gives very good performance related to DNA sequence analysis for querying of publicly available genome sequence data. By using this method the number of comparisons gradually decreases and comparison per character ratio of the proposed algorithm reduces accordingly when compared to the some of the existing popular methods. The experimental results show that there is considerable amount of performance improvement due to this the overall performance increases.

*Keywords***:** *Count, Index, Pair, Sequence*

1. Introduction

Genetic algorithms are based on ideas from population genetics. These algorithms are powerful tools for solving complex pattern-matching problems, especially when the matching is incomplete or inexact or when it occurs on repetitive patterns separated by unmatched patterns, as it can be in searches for long DNA sequences that take into possible alterations, from single deletions or insertions to crossovers.

Exact string matching consists of finding one or, more generally, all of the occurrences of a pattern in a target. Text-Based applications must solve two kinds of problems, depending on which string, the pattern or the target, is given first. In computational biology the application of computer technology is used to the management of biological information. Computers are used to gather, store, analyze and integrate biological and genetic information which can then be applied to gene based drug discovery and development. The problem of string matching is to find all occurrences of pattern '*P*' of size '*m*' in the text string '*T*' of size '*n*'. Researchers have been focused this sphere of research, various techniques and algorithms have been purposed and designed to solve this problem. Exact String matching algorithms are widely used in bibliographic search, question answering application, DNA pattern matching, text processing applications and information retrieval from databases.

Every human has his/her unique genes. Genes are made up of DNA and therefore the DNA sequence of each human is unique. However the DNA sequences of all humans are 99.9% identical, which means there is only 0.1% difference. DNA is contained in each living cell of an organism, and it is the carrier of that organism's genetic code. The genetic code is a set of sequences, which define what proteins to build within the organism. Since organisms must replicate and reproduce tissue for continued life, there should be some means of encoding the unique genetic code for the proteins. The genetic code is the information which will be needed for biological growth and reproductive inheritance. As DNA is the basic blue print of life it can be viewed as a long sequence over the four alphabets *A, C, G* and *T*. It contains genetic instructions of an organism and is mainly composed of nucleotides of four types. Adenine *(A),* Cytosine *(C),* Guanine *(G),* and Thymine *(T)*. The pattern itself may not be exactly known, because it may involve insertion, deletion, or replacement of the symbols. The amount of DNA extracted from the organism is increasing exponentially. The DNA constitutes the heritable genetic information in nuclei, plasmids, mitochondria, and chloroplasts that forms the basis for the developmental programs of all living organisms. Determining the DNA sequence is therefore useful in basic research studying fundamental biological processes, as well as in applied fields such as diagnostic or forensic research. Because DNA is key to all living organisms, knowledge of the DNA sequence may be useful in almost any biological subject area. In medicine it can be used to identify, diagnose and potentially develop treatments for genetic diseases. Similarly, genetic research into plant or animal pathogens may lead to treatments of various diseases caused by these pathogens. When we know a particular sequence is the cause for a disease, the trace of the sequence in the DNA and the number of occurrences of the sequence defines the intensity of the disease. As the DNA is a large database we need an efficient algorithm to find out a particular sequence in the given DNA. We have to find the number of repetitions and the start index and end index of the sequence, which can be used for the diagnosis of the disease and also the intensity of the disease by counting the number of pattern matching strings, occurred in a g ene database. The biologists often queries new discoveries against a collection of sequence databases such as GENBANK, EMBL and DDBJ to find the similarity sequences. As the size of the data grows it b ecomes more difficult for users to retrieve necessary information from the sequences. Hence more efficient and robust methods are needed for fast pattern matching techniques. The string matching can be described as: given a specific strings *P* generally called pattern searching in a large sequence/text *T* to locate *P* in *T*. if *P* is in *T*, the matching is found and indicates the position of *P* in *T*, else pattern does not occurs in the given text. Pattern matching techniques has two categories and is generally divides into single pattern matching and multiple pattern matching algorithms.

- Single and Multiple pattern matching algorithms

In a standard problem, we are required to find all occurrences of the pattern in the given input text, known as single pattern matching. Suppose, if more than one pattern are matched against the given input text simultaneously, then it is known as, multiple pattern matching. Whereas single pattern matching algorithm is widely used in network security environments. Multiple pattern matching can search multiple patterns in a text at the same time. It has a high performance and good practicability, and is more useful than the single pattern matching algorithms. To determine the function of specific genes, scientists have learned to read the sequence of nucleotides comprising a D NA sequence in a p rocess called DNA sequencing. DNA comparison, pattern recognition, similarity detection and phylogenetic trees construction in genome sequences are the most popular tasks. From the biological point of view pattern comparison is motivated by the fact that all living organisms are related by evolution. This implies that the genes of species that are closer to each other should show signs of similarities at the DNA level.

Let $P = \{p1, p2, p3,..,pm\}$ be a set of patterns of *m* characters and $T=\{t=t1,t2,t3…tn\}$ in a text of *n* characters which are strings of nucleotide sequence characters from a fixed alphabet set called $\sum= \{A, C, G, T\}$. Let *T* be a large text consisting of characters in $\sum$. In other words *T* is an element of $\sum^*$. The problem is to find all the occurrences of pattern *P* in text *T*. Many existing pattern matching algorithms are reviewed and classified in two categories.

- Exact and Inexact string matching algorithm

Exact pattern matching algorithm will find that whether the probability will lead to either successful or unsuccessful search. The problem can be stated as: Given a pattern *p* of length *m* and a string/Text *T* of length n ($m \leq n$). Find all the occurrences of *p* in *T*. The matching needs to be exact, which means that the exact word or pattern is found. Some exact matching algorithms are Naïve Brute force algorithm, Boyer-Moore algorithm[3], KMP Algorithm[7]. Inexact/Approximate pattern matching is sometimes referred as approximate pattern matching or matches with *k* mismatches/ differences. This problem in general can be stated as: Given a pattern *P* of length *m* and string/text *T* of length *n*. ($m \leq n$). Find all the occurrences of sub string *X* in *T* that are similar to *P*, allowing a limited number, say *k* different characters in similar matches. The Edit/transformation operations are insertion, deletion and substitution. Inexact/Approximate string matching algorithms are classified into: Dynamic programming approach, Automata approach, Bit-parallelism approach, Filtering and Automation Algorithms. Inexact sequence data arises in various fields and applications such as computational biology, signal processing and text processing.

Pattern matching algorithms have two main objectives.

- Reduce the number of character comparisons required in the worst and average case analysis.
- Reducing the time requirement in the worst and average case analysis.

In many cases most of the algorithm operates in two stages. Depending upon the algorithm some of the algorithm uses pre-processing phase and some algorithm will search without it. Many Pattern matching algorithms are available with their own merits and demerits based upon the pattern length and the technique they use. Some pattern matching algorithm concentrates on pattern itself. Other algorithm compare the corresponding characters of the patterns and text from the left to right and some other perform the character from the right to left. The performance of the algorithm can be measured based upon the specific order they are compared. Pattern matching algorithms has two different phases.

- Pre-processing and searching phase

The pre-processing phase collects the full information and is used to optimize the number of comparisons. Whereas searching phase finds the pattern by the information collected in pre-processing.

2. Background and Related Work

In this paper we mainly focus on multiple pattern matching. Given a known pattern we wish to count how many times it occurs in the text, and to point out its occurrence positions. The outcome of this search can be further processed by the pattern discovery machinery, possibly to come back with new searches for more specific patterns. Since the search patterns of interest are in general complex, we leave out of this paper the search for exact strings, which are the most trivial search patterns. As methods were improved, matching set methods and approximate string matching techniques were developed. Improvements in computer speed and evaluation of more complex problems necessitated multi-dimensional matching methods. Multi-dimensional methods are generally built on previously developed string matching algorithms and applied to multi-dimensional patterns. These include matching patterns for tree charts, graphs, pictures, proteins, nucleic acids and molecular phylogeny. Multi-dimensional methods include tree methods, two dimensional methods used in computer graphics, and three dimensional methods used in analyzing protein structures.

String matching mainly deals with problem of finding all occurrences of a string in a given text. In most of the DNA applications it is necessary for the user and the developer to be able to locate the occurrences of specific pattern in a sequence. In Brute-force algorithm the first character of the pattern *P* is compared with the first character of the string *T*. If it matches, then pattern *P* and string *T* are matched character by character until a mismatch is found or the end of the pattern *P* is detected. If mismatch is found, the pattern *P* is shifted one character to the right and the process continues. The complexity of this algorithm is *O(mn).* The Bayer-Moore algorithm[3] applies larger shift-increment for each mismatch detection. The main difference the Naïve algorithm had is the matching of pattern *P* in string *T* is done from right to left *i.e.,* after aligning *P* and string *T* the last character of *P* will matched to the first of *T* . If a mismatch is detected, say *C* in *T* is not in *P* then *P* is shifted right so that *C* is aligned with the right most occurrence of *C* in *P*. The worst case complexity of this algorithm is $O(m+n)$ and the average case complexity is *O(n/m).* In IFBMPMA[12] the elements in the given patterns are matched one by one in the forward and backward until a mismatch occurs or a complete pattern matches .The KMP algorithm[7] is based on the finite state machine automation. The pattern *P* is pre-processed to create a finite state machine *M* that accepts the transition. The finite state machine is usually represented as the transition table. The complexity of the algorithm for the average and the worst case performance is *O(m+n).* In IBKPMPM[13] algorithm we first choose the value of *k* (a fixed value), and divide both the string and pattern into number of substring of length *k*, each substring is called as a partition. If *k* value is 3 we call it as 3-partition else if it is 4 then it is 4-partition algorithm. We compare all the first characters of all the partitions, if all the characters are matching while we are searching then we go for the second character match and the process continues till the mismatch occurs or total pattern is matched with the sequence. If all the characters match then the pattern occurs in the sequence and prints the starting index of the pattern or if any character mismatches then we will stop searching and then go to the next index stored in the index table of the same row which corresponds to the first character of the pattern *P*. In approximate pattern matching method the oldest and most commonly used approach is dynamic programming. In 1996 Kurtz[8] proposed another way to reduce the space requirements of almost *O(mn).* The idea was to build only the states and transitions which are actually reached in the processing of the text. The automaton starts at just one state and transitions are built as they are needed. The transitions those were not necessary will not be build. The Deviki-Paul algorithm[5] for multiple pattern matching requires a pre-processing of the given input text to prepare a table of the occurrences of the 256 member ASCII character set. This table is used to find the probability of having a match of the pattern in the given input text, which reduces the number of comparisons, improving the performance of the pattern matching algorithm. In the MSMPMA[18] technique the algorithm scans the input file to find the all occurrences of the pattern based upon the skip technique. By using this index as the starting point of matching, it c ompares the file contents from the defined point with the pattern contents, and finds the skip value depending upon the match numbers (ranges from 1 to m-1). Harspool[6] does not use the good suffix function, instead it uses the bad character shift with right most character .The time complexity of the algorithm is *O(mn).* Berry-Ravindran[2] calculates the shift value based on the bad character shift for two consecutive text characters in the text immediately to the right of the window. This will reduce the number of comparisons in the searching phase. The time complexity of the algorithm is *O(nm)* .Sunday[4] designed an algorithm quick search which scans the character of the window in any order and computes its shift with the occurrence shift of the character *T* immediately after the right end of the window. The FC-RJ[11] algorithm searches the whole text string for the first character of the pattern and maintains an occurrence list by storing the index of the corresponding character. It uses an array equal to size of the text string for maintaining occurrence list. Time and space complexity of pre-processing is *O(n).*

Ukkonen[15] proposed automation method for finding approximate patterns in strings. He proposed the idea using a DFA for solving the inexact matching problem. Though automata approach doesn't offer time advantage over Boyer-Moore algorithm[3] for exact pattern matching. The complexity of this algorithm in worst and average case is *O(m+n).* In this every row denotes number of errors and column represents matching a p attern prefix. Deterministic automata approach exhibits *O(n)* worst case time complexity. The main difficulty with this approach is construction of the DFA from NFA which takes exponential time and space. Wu.S.Manber.U[16] proposed the algorithm for fast text searching allowing errors. The first bit-parallel method is known as "*shift-or*" which searches a pattern in a text by parallelizing operation of non deterministic finite automation. This automation has $m+1$ states and can be simulated in its non deterministic form in $O(mn)$ time. The filtering approach was started in 1990. This approach is based upon the fact it may be much easier to tell that a text position doesn't match. It is used to discard large areas of text that cannot contain a match. The advantage in this approach is the potential for algorithms that do not inspect all text characters. By using dynamic programming approach especially in DNA sequencing Needleman-Wunsch[9] algorithm and Smith-waterman algorithms[14] are more complex in finding exact pattern matching algorithm. By this method the worst case complexity is $O(mn)$. The major advantage of this method is flexibility in adapting to different edit distance functions. The Raita algorithm[10] utilizes the same approach as Horspool algorithm[6] to obtaining the shift value after an attempt. Instead of comparing each character in the pattern with the sliding window from right to left, the order of comparison in Raita algorithm[10] is carried out by first comparing the rightmost and leftmost characters of the pattern with the sliding window. If they both match, the remaining characters are compared from the right to the left. Intuitively, the initial resemblance can be established by comparing the last and the first characters of the pattern and the sliding window. Therefore, it is anticipated to further decrease the unnecessary comparisons.

The Aho-Corasick algorithm[1] developed at Bell Labs in 1975 by Alfred Aho and Corasick is an extension of the KMP algorithm[7]. The AC algorithm consists of constructing a finite state pattern matching machine from the keyword and then using the machine to process the text in a single pass. It can find an occurrence of several patterns in the order of $O(n)$ time, where n is the length of the text, with pre-processing of the patterns in linear time. Two dimensional pattern matching methods are commonly used in computer graphics. Takaoka and Zhu[19] proposed using a combination of the KMP algorithm[7] and RK methods in an algorithm developed for two dimensional cases. The second approach that runs faster when the row length of the pattern increases and is significantly faster than previous methods proposed. Three dimensional pattern matching is useful in solving protein structures, retinal scans, finger printing, music, OCR and continuous speech. Multi-dimensional matching algorithms are a natural progression of string matching algorithms toward multi-dimensional matching patterns including tree structure, graphs, pictures, and proteins structures.

3. Multiple Pattern Matching Algorithm using Pair-Count

The most common approach is to improve efficiency which involves the idea of indexing method where the number of comparisons is reduced when compared with different existing algorithms. So a new index based algorithm is proposed. In such approach the characters are indexed according to their indexes as they occur in the text/sequence. The efficiency and performance highly depends upon the character size. The objective of the work is to find the patterns from the sequence file of large size. Many different solutions have been proposed to bring the optimal results with exact matching sequence data but gets inaccurate and slow results. Latest computational technology uses fast algorithms which made relatively easy in bringing accurate results.

In the proposed work indexes has been used for the DNA sequence. We have to search a pattern in a string whose alphabet set = $\{A, C, G, T\}$. Let the string be S of n characters and the pattern P of m characters. After creating the index the algorithm will search for the pattern in the string using the index of least occurring character in the string and the pair comparison method for comparison is done once we align with the least count character with the pattern. The index based algorithm uses a table called stab[4][n] which stores all the indexes of each character in its corresponding vector with the occurrence index. Current algorithm used for the pair count technique is suitable for the DNA pattern matching as well as for the protein sequence and normal text comparison. The algorithm provides dynamic array generation mechanism for pattern matching of all 256 ASCII character value. As we scan the sequence from left to right the characters are places into the table by using the hash function. Generally the algorithm generated here will provide the view for 2D array generation at runtime so as to construct only those many character subscript as many are available in the text string. Such dynamic array generation reduces the time as well as space complexity both in comparison to other existing popular methods. It also helps in decreasing the number of searching comparison for the 2D array. Here step 3 to step 7 shows the 2D array generation procedure and appropriate example is discussed at later sections for random character among all 256 ASCII values.

3.1 Algorithm

```
Input-  Combination of pattern P[ ] (pattern of string to search)
        S [ ] (text of the string)
Step 1  Initialize
        i =0, k = 0 , l = 0; {// Incremental variables}
Step 2  Char C = S[i]
        int d = int ( S[i])      //ASCII value conversation
                //2D DYNAMIC ARRAY
Step 3  If c is distinct       // Subscript table construction
                m = (d - 64) % 5
                subscript[k][l] = c
                l = l+1
                subscript[k][l] = m
                k = k+1;
Step 4  If c not the last char in S[ ]
                i= i+1;
                GOTO step 2
```

```
Step 5  Take character c from pattern P[ ]  // 2D array construction
            c= P[i]
Step 6  Stab[n][m] = c   // stab[ ][ ] is 2D array
            m=m+1
            stab[n][m] = index of c
Step 7  repeat till all c in S[ ]
        m=m+1
        index= i
        stab[n][m]=index
          //2D DYNAMIC CONSRTUCTION ENDS
Step 8  Next i
         n=n+1
         GOTO step 5
Step 9  for each pattern in combination of pattern
                    Follow step 10 to step 15
Step 10  Calculate and store each character of pattern P[ ] in an array c_occur[ ] and their index in c_index[ ]
Step 11  Compute i where i=index of character having minimum occurrence
Step 12  Compute pos in pattern where Pos=index of character (having min occurrence in s[ ] in pattern
Step 13  Compute l
         l= i - pos of char in pattern[ ]
Step 14  Compare the pair of character from s and pattern
         no_of_comp++;
         if (compare is true)
          continue next pair compare
          no_of_comp++
          no_of_occr++
              if(false)
          no_of_comp++
Step 15  Next i
         GOTO Step 3
```

The basic idea used here is to store all the indexes of each character in its corresponding row in the 2D vector. Now using this pre-processed index we will not be opting for normal sequential comparison rather we have implemented ASCII indexing technique which reduces the comparisons as well as memory usage. The ASCII indexing technique will be used to reduce the pre-processing time. It means the searching of row in 2-D array for the occurrence index of character will take time when normal method is applied, but if we apply ASCII indexing technique it reduces the time complexity by reducing number of search to get correct row of 2D array. For DNA pattern (*A,C,G,T*) we get array subscript by using the hash function [(S[i]-64)%5], and the subscript Table.1 is as follows.

Table.1.Array Subscript values for the DNA sequence

S. No	DNA	ASCII value	Val(ASCII)-64	{Val(ASCII)-64}%5	Array Subscript
1.	A	65	1	1	1
2.	C	67	3	3	3
3.	G	71	7	2	2
4.	T	84	20	0	0

By the above table we can fetch the occurrence of *A,C,G,T* in 2D array by just going into the row which is the subscript of corresponding character. Suppose we have a character A which is having ASCII value of 65 and by using the hash function we will get its array subscript as 1. By using the array subscript of each character we can go directly to the row of 2D array which is the subscript of the corresponding character. The hash function [*(S[i]-64)%5*] always returns a subscript value in the range 0,1,2,3 which is needed for subscripting 2D array of size *[4][n]*. The subscript values 0,1,2,3 represent the characters *T, A, G* and *C* respectively. So for each character in the string of the function *((S[i]-64)%5)* directly references to its corresponding row in the 2D table.

3.2 Pre-processing and Searching

Suppose when it comes to a normal text where a pattern is to be searched in that text, then as far as pre-processing is concerned it will first fetch each distinct character from the text one by one by incrementing the index of the text and calculate for each character with the corresponding subscript (stored in *subscript*[ ]). The advantage of calculating subscript is number of comparison is gradually reduced with array subscript. In pattern matching once the subscript is known we can go directly to that row in the occurrence table which reduces the comparison by *n/4*. Pre-processing of index occurrence table is done by taking each character one by one from the text and then its corresponding index is stored in 2D array with the assigned subscript for the character as row of the 2D array. It goes as, if once we got the character then its ASCII subscript is taken, and the corresponding row of 2D array is filled with the index of the character. The pre-processing method is used for constructing 2D array as it goes till all the character of the text completes *i.e.*, till we reaches the end of the text. After this we are ready with occurrence index of each character in the text and also with the subscript of each distinct character going to come in the text or pattern. With the presence of these two tables it almost reduces the initial comparison at pre-processing phase to one-fourth of the normal algorithm. Once we get the text (**S[ ]**) and pattern (**P[ ]**) as input our primary job is to search for the minimum count (*i.e.* occurrence of character) character among the pattern's character by going through the count column of subscript array. Once we get the minimum count character then we have to access its occurrence index from 2D array with the help of its subscript value which is already available with us as a pre-processed table.

As we are now having first occurrence index of the minimum count character then we go for alignment method of the pattern with the text. For comparison of character the text pointer is decremented to one less than the length of the pattern (*i.e. L= n-1*), where L= comparison pointer. After alignment and pointer shift, now pair wise comparison is done from text (S [ ]) and pattern (P[ ]) one by one each pair is taken, if first pair matches the corresponding pattern's first pair then we shift pointer twice for comparison of next pair and it goes till the pattern end is encountered. If matching fails then we switch's to next occurrence of the character from the 2D array. After the occurrence of pattern in the text or after a matching failure we go again to 2D array for next occurrence index and minimum count character. Then the same alignment procedure is followed till the end of the text. For a new pattern again the minimum count character is searched and corresponding subscript valued row is fetched for the occurrence index of that character. In this scheme pattern matching uses the pre-processed table of subscript and 2D array of reference of each different pattern, to calculate their occurrence. The pre-processing of table is needed at the starting of the text, which is once done can be used for any type and number of pattern for that particular text.

3.3 Example for the DNA Text Sequence

Take a string *S=GCTCGATTTCGATGGCTCGAATCCTA* of 26 DNA characters and *P = TCGA* of 4 characters. The index table stores all the indexes of each character *A, C, G* and *T* in its corresponding row as it occurs in the sequence. The 0th row stores the indexes of occurrences of the character *T*, 1st row for *A*, 2nd row for *G* and 3rd row for *C*. It also stores the total number of occurrences of each character in separate array. The comparisons will start from the character in pattern *P* which is occurring least number of times in the string. For the sequence *S* the element *T, A, G* and *C* are occurring 8, 5, 6 and 7 times respectively. Here *A* is occurring least number of times, so *A* is used as the initial alignment and once if there is a match of pattern *A* with the sequence character *A* then rest of the pair will be compared in a sequential order for the pattern matching process. Once we have aligned the minimum count character with the text then pairing concept is applied for comparison in which we compare pair wise. Comparing pair wise often reduces our comparison in respect of normal comparison sequential method.

Table.2.DNA Sequence 2D index table

Subscript	Indexes								Count
T 0	2	6	7	8	12	16	21	24	8
A 1	5	11	19	20	25				5
G 2	0	4	10	13	14	18			6
C 3	1	3	9	15	17	22	23		7

In this technique the character *A* in pattern *P* is aligned with *A* in the DNA sequence *S*. Once the alignment is completed then it will match pair wise one by one pairs of the DNA sequence from the starting character of the given pattern by using the comparison pointer (*i.e. L= n-1*). In the index table the occurrence of the *A* character is 5 *i.e.,* least count is 5, so *A* will be used for matching process. Here the table has an extra field called count which increments every time as the character occurs in the sequence. The count helps to find the least count character which is available from the index table, so only those many comparisons can be done. The algorithm maps to the first occurrence of *A* according to the table and then starts comparing the first pair of the pattern with the possible match in the string relative to that *A*.

*S=GCTCG**A**TTTCGATGGCTCGAATCCTA*
*P =TCG**A***

Here the alignment is done in accordance with least count character *i.e., A*, now we perform pairing comparison from pattern *P* with text *S*.

*S=GC**TC**GATTTCGATGGCTCGAATCCTA*
*P=**TC**GA*

First pair *TC* matches with the first pair of text so it goes to next step *i.e.*, it will now compare for next pair of it.

*S=GC**TCGA**TTTCGATGGCTCGAATCCTA*
*P =**TCGA***

For second pair also we get the pair matching, which tells that one occurrence of the pattern has been found. Now we go to the second index in the table of occurrence of *A*.

*S=GCTCGATTTCG**A**TGGCTCGAATCCTA*
*P=TCG**A***

Here the next *A* is present in the 11 index. So the initial alignment is performed.

*S=GCTCGATT**TC**GATGGCTCGAATCCTA*
*P=**TC**GA*

Then the first pair *TC* matches with the first pair of text, now compare for next pair of it.

*S=GCTCGATT**TCGA**TGGCTCGAATCCTA*
*P =**TCGA***

For second pair also we get the pair matching, where second occurrence of the pattern has been found. Now we go to the third index in the table of occurrence of *A*.

*S=GCTCGATTTCGATGGCTCG**A**ATCCTA*
*P =TCG**A***

The third index as available in 2D index table of *A* which is 19. After aligning the *A* we perform pair comparison.

*S=GCTCGATTTCGATGGC**TC**GAATCCTA*
*P =**TC**GA*

First pair *TC* matches with the first pair of text so it goes to next step *i.e.*, it will now compare for next pair of it.

*S =GCTCGATTTCGATGGC**TCGA**ATCCTA*
*P =**TCGA***

For second pair also matches with the sequence and next pattern is found from the sequence. Now we go to the fourth index in the table of occurrence of *A*.

*S =GCTCGATTTCGATGGCTCGA**A**TCCTA*
*P =TCG**A***

Here the first pair *TC* do not matches with the text's pair so we leave rest of the pair comparison for this index and move to last index of *A*. Here also the pattern doesn't match so we stop.

*S =GCTCGATTTCGATGGCTCGAATCCT**A***
*P =TCG**A***

By above example we can conclude that taking least count and pairing comparison often reduces the number of comparison in corresponding to other algorithmic technique where normal checking is done. More over use of ASCII method with 2D array also reduces the space and time complexity factor which is very high in many earlier proposed algorithms.

3.4 Normal text sequence

Take a string *S=ACDAGOODDAACDAGACGOODD* of 21 characters and combination of pattern for which we will be checking the general English text pattern. Here the distinct characters for the two pattern combination is *P={ ACDA, GOOD}* of 5 distinct character in a new combination. The index table stores all the indexes of each character *A, C, D, G* and *O* in its corresponding row as it occurs in the sequence. The 0th row stores the indexes of occurrences of the character *O*, 1st row for *A*, 2nd row for *G*, 3rd row for *C*, 4th row for *D* with the help of the hashing technique. It stores the total number of occurrences of each character in separate array. The comparisons will start from the character in pattern *P* which is occurring least number of times in the string. For the sequence *S* the element *A, C, D, G,* and *O* are occurring 5, 3, 6, 3 and 4 times respectively. Here *C* is occurring least number of times, so *C* is used as the initial alignment and once if there is a match of pattern *C* with the sequence character *C* then rest of the pair will be compared in a sequential order for the pattern matching process. Once we have aligned the minimum count character with the text then pairing concept is applied for comparison in which we compare pair wise. In this technique the character *C* in pattern *P* is aligned with *C* in the sequence *S*. Once the alignment is completed then it will match pair wise using one by one pairs of the English text sequence from the starting character of the given pattern. In the index table the occurrence of the *C* character is only 3 *i.e.,* least count is 3 so *C* will be used for matching process. Here the table has an extra field called count which increments every time as the character occurs in the sequence. The count helps to find the least count character which is available from the index table, so only those many comparisons can be done. The algorithm maps to the first occurrence of *C* according to the table and then starts comparing the first pair of the pattern with the possible match in the string relative to that *C*.

*S=A**C**DAGOODDAACDAGACGOODD*
*P=A**C**DA*

Here the alignment is done in accordance with least count character *i.e.*, *C* in our case with the text, now we perform pairing comparison from pattern *P* with text *S*.

S= ***AC**DAGOODDAACDAGACGOODD*
P= ***AC**DA*

First pair *AC* matches with the first pair of text so it goes to next step *i.e.,* it will now compare for next pair of it.

S= ***ACDA**GOODDAACDAGACGOODD*
P= ***ACDA***

For second pair also we get the match, where the first occurrence of the pattern has been found. Now we go to the second index in the table of occurrence of *C*.

*S=ACDAGOODDAA**C**DAGACGOODD*
*P=A**C**DA*

The second occurrence of the *C* is at 11 position .Now we perform pairing comparison from pattern *P* with text *S*.

*S=ACDAGOODDA**AC**DAGACGOODD*
*P=**AC**DA*

First pair *AC* matches with the first pair of text so it goes to next step *i.e.*, it will now compare for next pair of it.

*S=ACDAGOODDA**ACDA**GACGOODD*
*P=**ACDA***

For second pair also we get the pair matching, so the second pattern has been found from the text. Now we go to the third index in the table of occurrence of *C.*

*S=ACDAGOODDAACDAGA**C**GOODD*
*P=A**C**DA*

The third occurrence of the *C* is at 16th position. So we perform pairing comparison from *P* with text *S.*

*S=ACDAGOODDAACDAG**AC**GOODD*
*P=**AC**DA*

First pair *AC* matches with the first pair of text so it goes to next step *i.e.,* it will now compare for next pair of it.

*S=ACDAGOODDAACDAG**AC**GOODD*
*P=**AC**DA*

For the next pair since it do not matches so we leave this occurrence and since once further occurrence exist so we go for net combination of pattern. For 2nd combination we have P = {*GOOD*}. In the index table the occurrence of the *G* character is only 3 *i.e.,* least count is 3 so *G* will be used for initial alignment process. If we use the pair-count process two patterns will be matched related to GOOD pattern.

3.5 Mathematical Proof

Let *X* be a text of size *m* and *Y* be given pattern of size *n*, *m>= n*, let us assume stab[n][m] be 2D array having occurrence index of each character. Then for pre-processing tables needed for pattern matching mathematical calculation done are as follows.

Character ASCII value

e.g. T= 84

Now here hashing technique is employed with a hash function for construction of hash key.

Val = |[ASCII val(char) - 64]| %5

Subscript = val;

Index occurrence table

Stab [val][0] = 1st position index

And it goes on for each character's next occurrence and for next character the corresponding hash key is taken for stab[ ][ ] row.

In our case ***e.g.*** let us take *A, C, G, T i.e.*, n=4. If count [ ] being array keeping number of occurrence details so,

Stab[min(count[n])][0] = 1st index of the minimum count character in text *X*.

POS = position of that character in *Y i.e.,* pattern.

I = character index in 2D array.

Now, for alignment, text *X*'s pointer for comparison has to shift as per the length of the pattern and as per the given 1st character of the pattern. Mathematically it can be done as,

L = I – POS

Where *L*= 1st pair of *X* to be compared after alignment with *Y.*

The comparison took in pair as *X[L] + X[L+1]* with *Y[0] + Y[1]*.

Therefore,

L= character index in 2D array – POS.

X = *GCTCGATTTCGATGGCTCG**A**ATCCTSA*

Y =*TCG**A***

Here alignment is done but now for comparison with pairs of *X*, *L* is to be generated, *i.e.*,
In this case *I* = 20 (index of *A*'s occurrence)
POS= 3(position of character in *Y*)
Therefore by our formula *L*= I – POS we have,
L=20-3 = 17
i.e., pair comparison starts from 17th position of text *X*
So *X[17] + X[17+1]* compared with *Y[0] + Y[1].*

3.6 ASCII value generation for dynamic 2D array at runtime

For finding array subscript we use the hashing function mechanism, once we get the subscript the particular character occurrence can be fetched by going into the given subscript row of the 2D runtime array. For avoiding the hash collision here we use linear probing collision avoidance technique which results in all distinct array subscript for the each distinct character of the given text *S*. Linear probing is a scheme in hashing technology resolving collision in between hashing of values of hash methods(function) sequentially searching the hash table for a free location. This is accomplished using two values, one as a starting value and one as an interval between successive values in progressive modular arithmetic. The second value, which is the same for all keys and known as the *linear_size*, is repeatedly added to the starting value until a free space is found, or the entire table is traversed.

Req cell = (index_start + linear_size) % array length

This algorithm, which is used in different hashing techniques, provides good memory caching (if linear size is equal to one), through good locality of reference. The performance of linear probing is also more sensitive to input distribution when compared to double hashing**.** The concept used for creation of ASCII value 2D array is the hashing method to generate the array subscript of each occurring character in the given text pattern. Once the character subscript will be generated then we can easily access the particular character with its index value. To generate proper unique subscript of the character we follow certain collision avoidance technique in the hashing method. Here we have applied the technique called linear probing method. Its mathematical approach is as follows.
The normal subscript is assigned to each character with the corresponding generated hashing value till the space in 2D array is vacant for that particular index. If the cell is not vacant then the following logic is applied.

Req cell = (index_start + linear_size) % array length

Where, index _start = the initial index value of the 2D array
linear_size= the size jump what we take in seek of vacant position (in case of linear probing step size=1)
array length = size of the 2D array.
Given an ordinary hash function *H(x)*, a linear probing function (*H(x, i)*) would be

$$H(x,i) = (H(x)+i) \pmod n$$

Here *H(x)* is the hash function, *n* the size of the hash table, and the *linear size* is *i* in this case. Using linear probing, dictionary operation can be implemented in constant time. In other words, insert, remove and find operations can be implemented in *O(1).*

3.7 Example for the combination of the characters, numbers and special symbols

As we have constructed the array for the different text for the pattern matching, here we generate the 2D ASCII array table for the text. For all distinct character of the text hash function ({|Val(ASCII)-64|}%9) has been applied and using linear probing technique proper subscript has been allotted. The various selected distinct character are # $ % A G ! Q 9 + <- 8 H and using the appropriate hashing technique it allots the subscript value within the range of 0 to 11. The array subscript table follows with the linear probing method.

Table.3. Index subscript of different Symbols

S. No	Char	ASCII value	|Val(ASCII) -64|	{|Val(ASCI I)-64|}%9	Array Subscript
1.	#	35	29	2	2
2.	$	36	28	1	1
3.	%	37	27	0	0
4.	A	65	1	1	3
5.	G	71	7	7	7
6.	!	33	31	4	4
7.	Q	81	17	8	8
8.	9	57	7	7	9
9.	+	43	21	3	5
10.	<-	96	32	5	6
11.	8	56	8	8	10
12.	H	72	8	8	11

Take a string *S=$%89A#AA%%89A$+!Q#AA%88<-G$+!Q%89AA#AA%#988HH* of 44 characters and *P = %89A* of 4 characters. The index table stores all the indexes of each character %, 8, 9 and A in its corresponding row as it occurs in the sequence. The 0th row stores the indexes of occurrences of the character %, 10th row for 8, 9th row for 9 and 3rd row for A. It stores the total no of occurrences of each character in separate array. The comparisons will start from the character in pattern *P* which is occurring least number of times in the string. For the sequence *S* the element %, 8, 9 and *A* are occurring 6, 7, 4, 10 times respectively. Here 9 is occurring least number of times so 9 is used as the initial alignment and once if there is a match of pattern 9 with the sequence character 9 then rest of the pair will be compared in a sequential order for the pattern matching process.

For the given sequence/text we used all the symbols *S=$%89A#AA%%89A$+!Q#AA%88<G$+!Q%89AA#AA%#988HH* the occurrence table is shown table.4. From the above sequence S we have taken the combination of special symbols, characters, numbers and special characters. As we scan the sequence a table is created called dynamic 2D table and these sequence S will be sent by using the hash function. The advantage of doing this is only the symbols, characters, numbers or special symbols available in the sequence size table will be created. Here we are reducing the memory size without taking unnecessary all the 256 characters. Only the characters which are in the sequence only the 2D table will be created as shown below.

Table.4.Index values for all the combination of characters

S.No	Char index	Character text/sequence indexes										Count
1	% 0	1	8	9	20	29	37					6
2	$ 1	0	13	25								3
3	# 2	5	17	34	38							4
4	A 3	4	6	7	12	17	18	32	33	35	36	10
5	! 4	15	27									2
6	+ 5	14	26									2
7	<- 6	23										1
8	G7	24										1
9	Q 8	16	28									2
10	9 9	3	11	31	39							4
11	8 10	21	22	40	41	42						5

For the given text *S=$%89A#AA%%89A$+!Q#AA%88<-G$+!Q%89AA#AA%#988HH* and pattern the *P=%89A*. The pair comparison is done with aligning the pattern with 9 which is having minimum count.

S= $**%89A**#AA%%89A$+!Q#AA%88<-G$+!Q%89AA#AA%#988HH
P= ***%89A***

Once the alignment has been done the pattern is matched pair wise. In this case one match has been found of pattern *P* in text *S*. Then we go for 11th index of 9 for comparing.

S=$%89A#AA%**%89A**$+!Q#AA%88<-G$+!Q%89AA#AA%#988HH
*P=**%89A***

Second match found of pattern *P* in text *S*. Then we go for the third occurrence of 9 at 31th index.

*S=$%89A#AA%%89A$+!Q#AA%88<-G$+!Q**%89A**A#AA%#988HH*
*P=**%89A***

Third match has been found for pattern *P* in *S*. Further no index of 9 in text *S* matches with the pattern.

3.8 Example for the Protein Sequence

Unlike what we have only 4 characters in the DNA, we can here have around 20 characters in the string text for which various different protein sequences. The below Table.5 shows different characters related to the protein sequence characters and the various patterns whose matching we will be done by pair count method.

Table.5.Protein sequence index subscript

Protein Sequences	ASCII value	|Val(ASCII) -64|	{|Val(ASCII)-64|}%9	Array Subscript
L	76	12	3	3
I	73	9	0	0
P	80	16	7	7
F	70	6	6	6
C	67	3	3	4
R	82	18	0	1
Q	81	17	8	8
H	72	8	8	9
K	75	11	2	2
W	87	23	5	5

For the given 10 distinct character of protein sequences we construct ASCII 2D array table to follow our pair count method for generating the array subscript for each character. Here also for 2D array generation we use a different hash function and a linear probing technique are used to avoid collision in assigning the subscript cell to the character in the 2D array. Randomly we have taken some protein characters and ASCII value is taken related to those characters and by doing hash operation we will get the array subscript ranging from 0-9. Now the corresponding sequences index of each character is taken in dynamic 2D array as follows.

Table.6.Protein sequence occurrence index

Char Index	Character text/sequence indexes										Count
I 0	0	1	7	10	19	31					6
R 1	16	28	43								3
K 2	3	5	6	17	18	29	30	37	38	39	10
L 3	8	20	24	32	36						5
C 4	4	21	33	44							4
W5	9	22	34	40							4
F 6	2	15	27								3
P 7	11	12	23	35	45						5
Q 8	13	25									2
H 9	14	26									2

Let us now take the protein sequence *i.e.,* the sequence be *S = IIFKCKKI**L**WIPPQHFRKKILCWPLQHFRKKILCWPLKKKWNNRCP* and pattern *P=KKIL*.
S is the sequence of the protein sequence and various patterns to be matched will be in *P*.

*S = IIFKC**KKIL**WIPPQHFRKKILCWPLQHFRKKILCWPLKKKWNNRCP*
*P=**KKIL***

Here the alignment is done with the character which is having least count. In this case *L* is having 5 as minimum count so we align pattern *P* with *S*. We perform pairing comparison from pattern *P* with sequence *S*.

*S = IIFKC**KK**ILWIPPQHFRKKILCWPLQHFRKHILCWPLKKKWNNRCP*
*P=**KK**IL*

First pair KK matches with the first pair of sequence, so we now compare for next pair.

*S = IIFKC**KKIL**WIPPQHFRKKILCWPLQHFRKKILCWPLKKKWNNRCP*
*P=**KKIL***

For second pair also matches with the sequence and the pattern is found from the sequence. Now we go to the second index in the table of occurrence of *L*. Here also for second index it matches so the given pattern is occurred in the text *S*.

*S= IIFKCKKILWIPPQHFR**KKIL**CWPLQHFRKKILCWPLKKKWNNRCP*

*P=**KKIL***

Then it will search for the further occurrence of the pattern in the text S. Further L occurs at 32nd index so the pattern is checked. So totally three patterns have been found from the sequence *S*.

*S = IIFKCKKILWIPPQHFRKKILCWPLQHFR**KKIL**CWPLKKKWNNRCP*

*P=**KKIL***

4. Experimental Results

In this section we present several experiments result analysis with some of the existing and the popular techniques. The below DNA sequence dataset has been taken for the testing of pair-count algorithm .The DNA biological sequence S∈∑*of size n=1024 and pattern P∈∑*. Let S be the following DNA sequence.

*AGAACGCAGAGACAAGGTTCTCATTGTGTCTCGCAATAGT
GTTACCAACTCGGGTGCCTATTGGCCTCCAAAAAAGGCT
GTTCAACGCTCCAAGCTCGTGACCTCGTCACTACGACGG
CGAGTAAGAACGCCGAGAAGGTAAGGGAACTAATGACGC
GTGGTGAATCCTATGGGTTAGGATCGTGTCTACCCCAAAT
TCTTAATAAAAAACCTAGGACCCCCTTCGACCTAGACTAT
CGTATTATGGACAAGCTTTAACTGTCGTACTGTGGAGGCT
TCAAAACGGAGGGACCAAAAAATTTGCTTCTAGCGTCAAT
GAAAAGAAGTCGGGTGTATGCCCCAATTCCTTGCTGCCC
GGACGGCCAGGCTTATGTACAATCCACGCGGTACTACAT
CTTGTCTCTTATGTAGGGTTCAGTTCTTCGCGCAATCATA
GCGGTACTTCATAATGGGACACAACGAATCGCGGCCGGA
TATCACATCTGCTCCTGTGATGGAATTGCTGAATGCGCAG
GTGTGAATACTGCGGCTCCATTCGTTTTGCCGTGTTGATC
GGGAATGCACCTCGGGGACTGTTCGATACGACCTGGGAT
TTGGCTATACTCCATTCCTCGCGAGTTTTCGATTGCTCATT
AGGCTTTGCGGTAAGTAAGTTCTGGCCACCCACTTCGAG
AAGTGAATGGCTGGCTCCTGAGCGCGTCCTCCGTACAAT
GAAGACCGGTCTCGCGCTAAATTTCCCCCAGCTTGTACAA
TAGTCCAGTTTATTATCAAAGATGCGACAAATAAATTGATC
AGCATAATCGAAGATTGCGGAGCATAAGTTTGGAAAACTG
GGAGGTTGCCAGAAAACTCCGCGCCTACTTTCGTCAGGA
TGATTAAGAGTATCGAGGCCCCGCCGTCAATACCGATGTT
CTTCGAGCGAATAAGTACTGCTATTTTGCAGACCCTTTGC
CAGGCCTTGTCTAAAGGTATGTTACTTAATATTGACAATAC
ATGCGTATGGCCTTTTCCGGTTAACTCCCTG.*

By the current technique different patterns are analyzed and the graph is plotted by using these results. Different pattern sizes has been taken from the DNA sequence ranging from 1 to 20 randomly and tested. The number of occurrences and the number of comparisons is shown in the Table.7. The number of comparisons per character (CPC) which is equal to (Number of comparisons/file size) can be used as a measurement factor, this factor affects the complexity time, and when it is decreased the complicity also decreases.

Table.7.Comparison of pair count with existing algorithms

SNo	Pattern(P's)	No of Char	No of Occ	IFBMPM	CPC	IBKPMPM	CPC	Pair Count	CPC
1	*A*	1	259	518	0.5	259	0.2	259	0.2
2	*AG*	2	53	624	0.6	518	0.5	247	0.2
3	*CAT*	3	11	567	0.5	542	0.5	296	0.2
4	*AACG*	4	5	614	0.5	614	0.5	258	0.2
5	*AAGAA*	5	2	616	0.6	607	0.5	272	0.2
6	*AAAAAA*	6	3	627	0.6	620	0.6	356	0.3
7	*AGAACGC*	7	2	600	0.5	613	0.5	268	0.2
8	*AAAAAAGG*	8	1	634	0.6	623	0.6	281	0.2
9	*GCTCATTAG*	9	1	582	0.5	590	0.5	269	0.2
10	*CCTTTTCCGG*	10	1	562	0.5	578	0.5	266	0.2
11	*TTTTGCCGTGT*	11	1	650	0.6	650	0.6	264	0.2
12	*TTCTTAATAAAA*	12	1	651	0.6	634	0.6	277	0.2
13	*GGGACCAAAAAAT*	13	1	579	0.5	582	0.5	269	0.2
14	*TTTTGCCGTGTTGA*	14	1	638	0.6	654	0.6	265	0.2
15	*CCTCCAAAAAAGGCT*	15	1	578	0.5	558	0.5	270	0.2
16	*GGCTGTTCAACGCTCC*	16	1	598	0.5	580	0.5	273	0.2
17	*TTTTCGATTGCTCATTA*	17	1	643	0.6	633	0.6	273	0.2
18	*GGGATTTGGCTATACTCC*	18	1	598	0.5	580	0.5	277	0.2
19	*GGCCTTGTCTAAAGGTATG*	19	1	579	0.5	585	0.5	272	0.2
20	*CCTGAGCGCGTCCTCCGTAC*	20	1	570	0.5	582	0.5	268	0.2

The below Fig.1 shows the graph analysis of the proposed method with the existing techniques. We have analyzed by taking different algorithms like IBKMPM and IFBMPM with the pair-count technique. The values have been taken from the above 1-20 pattern shown in above table 7. The dotted line shows the proposed method where as the above two lines shows the K-Partition and IFBMPM techniques.

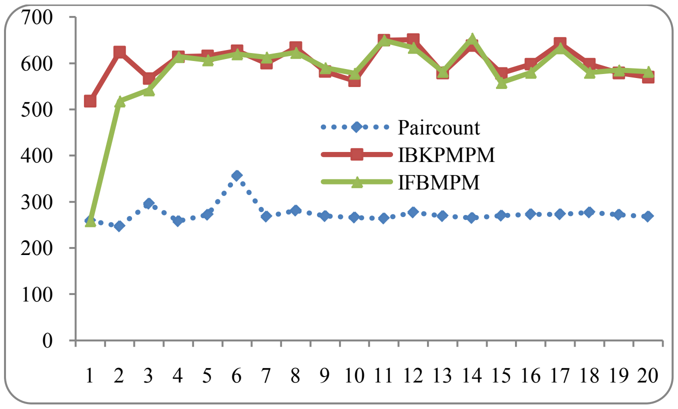

Fig 1. Comparison of different algorithms using DNA sequence

From Table.8.observations has been made for the following in terms of relative performance of our algorithm with the existing algorithms. The proposed algorithm gives good performance in two parameters like CPC ratio and number of comparisons with the algorithms like MSMPMA, Brute-force, Tri-Match, Naïve string matching, IFBMPM and IBKPMPM algorithms. From the below table the total number of occurrences of each pattern, the number of comparison and CPC ratio of each algorithm is given.

Table.8. Results of proposed algorithm with existing techniques

S.No	PATTERN	Occurrence	Boyer Moore	Quick Search	Deviki Paul	Pair-Count
1	A	259	1024	1024	259	259
2	A,AG	259,52	1758	1676	711	506
3	A,AG,CAT	259, 52,11	2399	2283	1228	802
4	A, AG, CAT, AACG	259, 52,11, 4	2909	2787	1754	1060
5	A, AG, CAT, AACG, AAGAG	259, 52,11, 4, 2	3282	3153	2281	1332
6	A, AG, CAT,AACG, AAGAG,AAAAAAC G	259, 52,11, 4, 2,0	3554	3529	2831	1602
7	A, AG, CAT,AACG, AAGAG,AAAAAAC G TTCTTAATAAAA	259, 52, 11, 4, 2, 0, 1	3731	3729	3398	1879
8	A, AG, CAT,AACG, AAGAG, AAAAAACG, TTCTTAATAAAA, GGCTGTTCAACGCTCC	259, 52,11, 4, 2,0, 1, 0	4041	4086	3928	2152

The below Fig.2 shows the comparison between different algorithms used for comparison in the above table for different pattern size ranging from 1-16 in size with the algorithms like IBKPMPM, IFBMPM, MSMPMA, Brute-Force, Tri-match and Naïve string search algorithm. It is clear that proposed (pair-count) algorithm outperforms when compared with all other algorithms. From the above Table.8 the total number of occurrences of each pattern, the number of comparison and CPC ratio of each algorithm is given. For each of algorithm we have two fields *i,e.,* number of comparisons and CPC ratio. In some of the cases as the size of the pattern increases the number of comparisons and comparison per character decreases in case of proposed method.

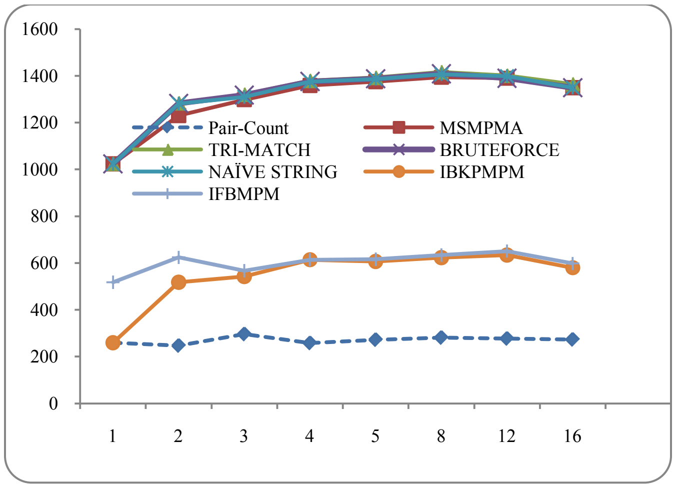

Fig.2. Comparison different algorithms using DNA sequence

The below Table.9. Shows the comparison with the different algorithms related to the DNA sequence compared with the algorithms like Boyer-Moore, Quick Search, DP with the proposed technique. From the experimental result analysis it is observed that the highest number of comparisons in the existing techniques is above 4000 where as in the proposed one it is less than 2200 comparisons for the largest pattern size of 16.

Table.9. Comparison of BM, QS, DP with PC for DNA Sequence

Pattern	PAIR COUNT		IBKMPM		IFBMPM		MSMPMA		BRUTE-FORCE		TRI-MATCH		NAÏVE STRING	
	No .of Com	*CPC*	*No. of Com*	*CPC*	*No. of Com*	*CPC*	*No. of Com*	*CPC*	*No. of Com*	*CPC*	*No. of Com*	*CPC*	*No. of Com*	*CPC*
A	259	0.2	259	0.2	518	0.5	1024	1.0	1024	1.0	1025	1.0	1024	1.0
AG	247	0.2	518	0.5	624	0.6	1230	1.2	1282	1.2	1284	1.2	1281	1.2
CAT	296	0.2	542	0.5	567	0.5	1298	1.2	1318	1.2	1321	1.2	1310	1.2
AACG	258	0.2	614	0.6	614	0.5	1359	1.3	1376	1.3	1380	1.3	1376	1.3
AAGAA	272	0.2	607	0.5	616	0.6	1375	1.3	1388	1.3	1393	1.3	1387	1.3
AAAAAAGG	281	0.2	623	0.6	634	0.6	1394	1.3	1409	1.3	1417	1.3	1407	1.3
TTCTTAATAAAA	277	0.2	634	0.6	651	0.6	1390	1.3	1390	1.3	1402	1.3	1399	1.3
GGCTGTTCAACGCTCC	273	0.2	580	0.5	598	0.5	1349	1.3	1349	1.3	1365	1.3	1349	1.3

The Fig.3 shows the graph comparison of different algorithms with the proposed pair count technique related to the DNA sequence**.** The algorithms which we have compared are Boyer-Moore, Quick search and DP with the proposed technique pair-count algorithm. The above shown are the different patterns which we have used for the comparisons.

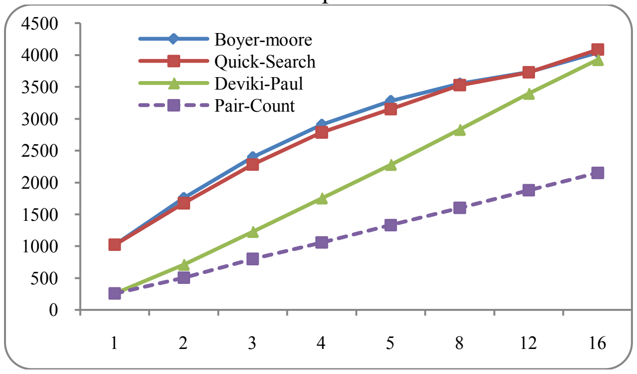

Fig 3. Comparison of different algorithms using DNA sequence

4.1 Experimental Analysis of Normal Text Sequence

A text of size 1024 characters were taken as given below and tested with the multiple patterns as shown in the Table.10. The below normal text sequence dataset has been taken for the testing of pair-count algorithm .The text sequence S∈∑*of size n=1024 and pattern P∈∑*. Let S be the following normal text sequence as shown below.

PATTERNMATCHINGISONEOFTHEBASICANDMOSTIMPO RTANTISSUESINTHERESEARCHAREASOFCOMPUTERSCI

ENCETHEMEANINGOFTHEPATTERNMATCHINGISTHATF INDINGTHEOCCURENCESOFAGIVENPATTERNINTHEGIV ENTEXTPATTERNMATCHINGISONEOFTHEMAJORISSUES INHEAREAOFNETWORKSECURITYANDALSOINMANYOTH ERAREASTHEINCREASEINNETWORKSPEEDANDTRAFFIC MAYCAUSETHEEXISTINGALGORITHMSTOBECOMEAPER FORMANCEBOTTLENECKTHEREFOREITISVERYNECESSA RYTODEVELOPMOREEFFICIENTPATTERNMATCHINGAL GORITHMINORDERTOOVERCOMETROUBLESONPERFOR MANCETHEREARESEVERALALGORITHMSINUSEINWHIC HDPALGORITHMISYIELDINGGOODRESULTSINMANYCAS ESHOWEVERTHISALGORITHMWASPROPOSEDONLYFOR THESINGLEPATTERNMATCHINGBUTNOWADAYSITISPAT TERNMATCHINGISONEOFTHEBASICANDMOSTIMPORTA NTISSUESINTHERESEARCHAREASOFCOMPUTERSCIENC ETHEMEANINGOFTHEPATTERNMATCHINGISTHATFINDI NGTHEOCCURRENCESOFAGIVENPATTERNINTHEGIVEN TEXTPATTERNMATCHINGISONEOFTHEMAJORISSUESIN THEAREAOFNETWORKSECURITYANDALSOINMANYOTER AREASTHEINCREASEINNETWORKSPEEDANDTRAFFICMA YCAUSETHEEXISTINGALGORITHMSTOBECOMEAPERFO RMANCEBOTTLENECKTHEREFOREITISVERYNECESSARY TODEVELOPMOREEFFICIENTPATTE

Table.10. Comparison with BM, QS, DP with PC for Text Sequence

S.No	PATTERN	Occurrence	Boyer Moore	Quick Search	Deviki Paul	Pair-Count
1	H	48	1024	1024	48	48
2	H,OF	48, 12	1571	1440	133	96
3	H, OF, AND	48, 12, 6	1963	1787	233	120
4	H, OF, AND,MOST	48, 12, 6,2	2295	2054	375	161
5	H, OF, AND,MOST, GIVEN	48, 12, 6, 2, 4	2587	2292	495	180
6	H, OF, AND,MOST, GIVEN, MATCHING	48, 12, 6, 2, 4, 8	2812	2506	595	231
7	H, OF, AND,MOST, GIVEN,MATCHING, PATTERNMATCH	48, 12, 6, 2, 4, 8, 8	3028	2732	738	321
8	H, OF, AND,MOST, GIVEN,MATCHING, PATTERNMATCH, ONEOFTHEBASICAND	48, 12, 6, 2, 4, 8, 8, 2	3173	2907	850	372

Fig.4. shows the graphical comparison of different algorithms with pair-count related to the normal text sequence. Towards X-axis we have taken different pattern sizes ranging from 1 to 16 in size and towards Y-axis we have taken number of comparisons from the above Table.10. The proposed pair-count algorithm outperforms when compared with all other algorithms. The current technique gives good performance in reducing the number of comparisons compared with other algorithms. The dotted line shows the pair-count model where as Boyer-Moore, Quick search and DP is shown by solid lines.

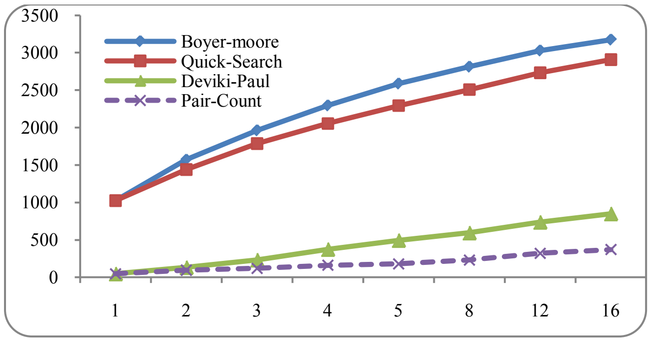

Fig 4. Comparison of different algorithms with using TEXT sequence

4.2 Advantages with the proposed technique

The following are observed from the experimental results. Reduction in number of comparisons with the some of the existing popular techniques. The ratio of comparisons per character has gradually reduced and is less than 1in the proposed technique where as it is greater than 1 in existing ones. The proposed algorithm is suitable for any size of the input file and once the indexes are created for input sequence we need not create them again and again. For each pattern we start our algorithm from the matching character of the pattern which decreases the unnecessary comparisons of other characters.

Conclusion

We have presented a n ew model, which is simple and yet effective algorithm for biological multiple pattern matching algorithm. In this study a new technique for improving the performance of Multiple pattern matching algorithm using pair-count is proposed. Comparison of proposed algorithm is made with existing algorithms on the basis of the number of comparisons and the attempts made by different pattern sizes of different algorithms to complete the task. Our algorithm outperform in case of number of comparisons related to the DNA sequences. The proposed model is shown to be very efficient as well as fast and gives the accurate results with the existing techniques. The analysis illustrate that the pair-count algorithm is better than the number of existing algorithms. Based on the experimental work carried out with DNA sequence data, pair-count approach gives the better performance.

References

[1] Aho, A. V., and M. J. Corasick, ‘‘Efficient string matching: an aid to bibliographic Search, ’’ Communications of the ACM (June 1975), pp. 333 340.

[2] .Berry, T. and S. Ravindran, 1999. A fast string matching algorithm and experimental results. In: Proceedings of the Prague Stringology Club Workshop ’99, Liverpool John Moores University, pp: 16-28.

[3] .Boyer R. S., and J. S. Moore, ‘‘A fast string searching algorithm‘Communications of the ACM 20, 762- 772, 1977.

[4] .D.M. Sunday, A very fast substring search algorithm, Comm. ACM 33 (8) (1990) 132–142.

[5] .Devaki-Paul, “Novel Devaki-Paul Algorithm for Multiple Pattern Matching” International Journal of Computer Applications (0975 – 8887) Vol 13– No.3, January 2011.

[6] .Horspool, R.N., 1980. Practical fast searching in strings. Software practice experience, 10:501-506

[7] .Knuth D., Morris. J Pratt. V Fast pattern matching in strings, SIAM Journal on Computing, Vol 6(1), 323-350, 1977.

[8] .Kurtz. S, Approximate string searching under weighted edit distance. In proceedings of the 3rd South American workshop on string processing. Carleton Univ Press, pp. 156-170, 1996

[9] .Needleman, S.B Wunsch, C.D(1970). “A general method applicable to the search for similarities in the amino acid sequence of two proteins.” J.Mol.Biol.48,443-453.

[10] .Raita, T. Tuning the Boyer-Moore-Horspool string-searching algorithm. Software - Practice Experience 1992, 22(10), 879-884.

[11] .Rami H. Mansi, and Jehad Q. Odeh, "On Improving the Naive String Matching Algorithm," Asian Journal of Information Technology, Vol.8, No. I, ISS N 1682-3915,2009, pp. 14-23.

[12] .Raju Bhukya, DVLN Somayajulu,‘‘An Index Based Forward backward Multiple Pattern Matching Algorithm, ‘World Academy of Science and Technology..June 2010, pp347-355

[13] .Raju Bhukya, DVLN Somayajulu,”An Index Based K-Partition Multiple Pattern Matching Algorithm”, ACEEE International Journal in Network Security Vol. 02, No. 02, Apr 2011.

[14] .Smith,T.F and waterman, M (1981). Identification of common molecular subsequences T.mol.Biol.147,195-197.

[15] .Ukkonen,E., Finding approximate patterns in strings J.Algor. 6, 1985, 132-137.

[16] .Wu S., and U. Manber, ‘‘Agrep — A Fast Approximate Pattern-Matching Tool,’’ Usenix Winter 1992 Technical Conference, San Francisco (January 1992), pp. 153 162.

[17] .Wu.S.,Manber U., and Myers,E .1996, A sub-quadratic algorithm for approximate limited expression matching. Algorithmica 15,1,50-67, Computer Science Dept, University of Arizona,1992.

[18] .Ziad A.A Alqadi, Musbah Aqel & Ibrahiem M.M.EI Emary, Multiple Skip Multiple Pattern Matching algorithms. IAENG International Vol 34(2) 2007.

[19] .Zhu, Rui Feng; T. Takaoka.“On improving the average case of the boyer-moore string matching algorithm” Journal of Information Processing **10** (3) 1987, 173–177.