# 2-Jump DNA Search Multiple Pattern Matching Algorithm

**Raju Bhukya[1], 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 in a DNA sequence or searching a pattern from a large data base is a major research area in computational biology. To extract pattern match 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 in the sequence. As performance plays a major role in extracting patterns from a given DNA sequence or from a database independent of the size of the sequence. When sequence databases grow, more efficient approaches to multiple matching are becoming more important. One of the major problems in genomic field is to perform pattern comparison on DNA and protein sequences. Executing pattern comparison on the DNA and protein data is a computationally intensive task. In the current approach we explore a new technique which avoids unnecessary comparisons in the DNA sequence called 2-jump DNA search multiple pattern matching algorithm for DNA sequences. 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***- *Characters, matching, patterns, sequence.*

## 1. Introduction

Bioinformatics is the application of computer technology for managing the 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 exact 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. The pattern matching problem has attracted a lot of interest throughout the history of computer science, particularly in the present day high performance computing and has used in various computer applications for several decades. These algorithms are applied in most of the operating systems, editors, search engines on the internet, retrieval of information (from text, image or sound) and searching nucleotide or amino acid sequence patterns in genome and protein sequence databases. Bioinformatics is a multi disciplinary science that uses methods and principle from mathematics and computer science and statistics for analyzing biological data where DNA pattern analysis plays a vital role, for various analyses like discrimination of cancer from the gene expression, mutations evolution, protein-protein interaction in cellular activities etc. Pattern matching plays a vital role in various applications in computational biology for data analysis like feature extraction, searching, disease analysis, structural analysis.

Pattern matching focuses on finding the occurrences of a particular pattern of in a text. The problem in pattern discovery is to determine how often a candidate pattern occurs, as well as possibly some information on its frequency distribution across the sequence/text. In general, a pattern will be a description of a set of strings, each string being a sequence of symbols. Hence, given a pattern, it is usual to ask for its frequency, as well as to examine its occurrences in a given sequence/text. Many algorithms have been developed each designed for a specific type of search. Although they all serve the same function but they vary in the way they process the search, and second in the methods they use to efficiently achieve the optimal processing time.

Every human has his/her unique genes. Genes are made up of DNA; therefore the DNA sequence of each human is unique. However, surprisingly, 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 must be some means of encoding the unique genetic code for the proteins used in making that tissue. The genetic code is information, which will be needed for biological growth and reproductive inheritance.

DNA is the basic blue print of life and it can be viewed as a long sequence over the four alphabets A, C, G and T. DNA contains genetic instructions of an organism. It is mainly composed of nucleotides of four types. Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). The amount of DNA extracted from the organism is increasing exponentially. So pattern matching techniques plays a vital role in various applications in computational biology for data analysis related to protein and gene in structural as well as the functional analysis. It focuses on finding the particular pattern in a given DNA sequence. 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 becomes more difficult for users to retrieve necessary information from the sequences. Hence more efficient and robust methods are needed for fast pattern matching techniques. It is one of the most important areas which have been studied in computer science. 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 multiple pattern matching and single pattern matching algorithms.

- Single pattern matching
- Multiple pattern matching techniques

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. In network security the pattern is a string indicating a network intrusion, attack, virus, and snort, spam or dirty network information, etc. 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 DNA sequence in a process called DNA sequencing. Comparison, pattern recognition, detecting similarity and phylogenetic trees constructing in genome sequences are the most popular tasks. The process of sequence alignment allows the insertion, deletion and replacements of symbols that representing the nucleotides or amino acids sequences. From the biological point of view pattern comparison is motivated by the fact that all living organisms are related by evolution. That implies that the genes of species that are closer to each other should show signs of similarities at the DNA level. Moreover, those similarities also extend to gene function. Normally, when a new DNA or protein sequence is determined, it would be compared to all known sequences in the annotated databases such as GenBank, SwissProt and EMBL.

Let $P = \{p_1, p_2, p_3,...,p_m\}$ 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*. It is an important application widely used in data filtering to find selected patterns, in security applications, and is also used for DNA searching. Many existing pattern matching algorithms are reviewed and classified in two categories.

- Exact string matching algorithm
- Inexact/approximate string matching algorithms

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 phase or study of the pattern.
- Processing phase or 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.

Bioinformatics has found its applications in many areas. It helps in providing practical tools to explore proteins and DNA in number of other ways. Bio-computing is useful in recognition techniques to detect similarity between sequences and hence to interrelate structures and functions. Another important application of bioinformatics is the direct prediction of protein 3-Dimensional structure from the linear amino acid sequence. It also simplifies the problem of understanding complex genomes by analyzing simple organisms and then applying the same principles to more complicated ones. This would result in identifying potential drug targets by checking homologies of essential microbial proteins. Bioinformatics is useful in designing drugs. Pattern matching in biology differs from its counterpart in computer science. DNA strings contain millions of symbols, and the pattern itself may not be exactly known, because it may involve inserted, deleted, or replacement of the symbols. Regular expressions are useful for specifying a multitude of patterns and are ubiquitous in bioinformatics. However, what biologists really need is to be able to infer these regular expressions from typical sequences and establish the likelihood of the patterns being detected in new sequences.

The sequence of 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. For example, 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 to go for efficient algorithms 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 gene database.

Since children inherit their genes from their parents, they can also inherit any genetic defects. Children and siblings of a patient generally have a 50% chance of also being affected with the same disease. Genetic testing can identify those family members who carry the familial unusual mutation and should undergo annual tumor screening from an early age. Genetic testing can also identify family members who do not carry the familial unusual mutation and do not need to undergo the increased tumor surveillance recommended for patients with unusual mutations. The unusual pattern in the strand reflects in the split strand and hence increases in the unusual mutations increase in the cells. All familial cancer syndromes are caused by a defect in a gene that is important for preventing development of certain tumors. Everybody carries two copies of this gene in each cell, and tumor development only occurs if both gene copies become defective in certain susceptible cells. Genetic testing can help to diagnose by detecting defects in the unusual mutated gene.

The rest of the paper is organized as follows. We briefly present the background and related work in section 2. Section 3 deals with proposed model *i.e.*, 2-JUMP DNA search multiple pattern matching algorithm. Experimental results and discussion are presented in Section 4 and we make some concluding remarks in Section 5.

## 2. Background and Related Work

This section reviews some work related to DNA sequences. An alphabet set $\sum = \{A, C, G, T\}$ is the set of characters for DNA sequence which are used in this algorithm.
The following notations are used in this paper:
DNA sequence characters $\sum = \{A, C, G, T\}$.
$\phi$ Denotes the empty string.

$|P|$ Denotes the length of the string *P*.
*S[n]* Denotes that a text which is a string of length *n*.
*P[m]* Denotes a pattern of length *m*.
*CPC*-Character per comparison ratio.

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 IBKMPM [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 preprocessing 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. The probability of having a match of the pattern in the given text is mathematically proved.

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 compares 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. Time and space complexity of preprocessing is *O(n).* FC_RJ uses an array equal to size of the text string for maintaining occurrence list.

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 pattern 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 proposed using a combination of the KMP[6] 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. 2-JUMP DNA Search Multiple Pattern Matching Algorithm

In this method we use combination of both the techniques

- Index Based Search
- ASCII sum

The index based search has been well established. Here we created index table of the input data and our search skips primarily on the index-row of the first character of the pattern. However in our proposed work, we go one step ahead and rather than using primitive method of comparing single character at a time, we rather compare sum of two characters of both input sequence data and pattern. This reduces our comparisons by one-third (we count one comparison for sum). After we match it completely we go for order checking in the subgroups sequentially until there is a mismatch or it completely matches.

### 3.1. Algorithm

```
 Input[n] : Input character array of length n.
 Patt[m] :  Pattern character array of length m.
 IndexTable[4][n] – index Table of input of length 4*n (ACGT)
   Let i,j,startIndex,flag,compare,counter  integer variables
    i=j=start Index=compare=counter=0.
    Flag=1
1.  Create the index table.
2.  Fetch startIndex as per first letter of pattern.
    startIndex = IndexTable [firstLet][i];
3.  while(n-startIndex > m)
            while(j<m)
if(m-j==1) // odd no. of characters in pattern.
            if(input[startIndex+j] != pat[j])
            compare++;
             flag=0;
              break;
            Inp2 =input[startIndex+j]+input[startIndex+j+1];
            Pat2 = pat[j]+pat[j+1];
            Compare++;
             If(inp2!=pat2)
             Flag=0;
                        Break;
             Else
             compare++;
 If(input[startIndex+j] != pat[j]|| input[startIndex+j+1] != pat[j+1])
                        flag=0;
                        break;
                        If(flag == 1)
                        Counter++;
                        Else
                        Flag=1;
                        J=0;
        StartIndex = IndexTable[firstLet][++i];
```

### 3.2. Index Based Search

This method has been invented and used to reduce the search time drastically. In this method we make an Index table of given input on the basis of characters involved which in our case are *A,C,G,T*. So, we have a (4xSize of input) table. Now we concentrate only on the index row of first character of our pattern and continue our comparison technique from the first index onwards. Based upon our comparisons results of success or failure we can directly jump to next potential occurrence of pattern by moving to the next index in the row chosen. We continue above operations till we finish all indexes of that row. In this way we need not move serially through the input, but rather we only concentrate only on the potential strings.

## 3.3 ASCII SUM (or 2-Jump)

Our unique comparison method adds further benefits to our Index Based Search. Here we use unique property of characters involved in our search patterns and input. As we are dealing with only genetic data, so our domain confines to following four characters *A, C, G, T*. Further reducing these characters to single digits by mod formula.

Table.1. Subscript values of DNA sequence characters

| S.No | DNA | ASCII Value | ASCII Value-64 | (ASCIIValue-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 |

Now we can use unique property of above integers. Any sum of above in combination of two gives a unique number in return.

A + A ~ 1 + 1 = 2
A + T ~ 1 + 0 = 1
A + G ~ 1 + 2 = 3
A + C ~ 1 + 3 = 4

And so on for other integers too. Now we can use this to reduce our both input size and patterns to half the length they actually are, *i.e.,* we combine two neighboring alphabets (or their reduced integers) to give single integers.

E.g. *Sequence=ATTGCCATA*
Equivalent integers: 100233101
*Pattern-GCCA*
Equivalent integers: 2 3 3 1

Here the first character of pattern is '*G*'. From our sequence we find that first index of character '*G*' is at 4. So we start forming groups from 4th index onwards. 2-Sum groups starting at '*G*' of sequence: (2+3), (3+1) = 5,4.
2-Sum groups of pattern: (2+3), (3+1) = 5,4.

So, now rather than comparing each character/integer separately we can compare two of them in one go. If in one go we find that our pattern string matches a substring of the input, and then we can go further and compare the two characters. This will be necessary as the two characters may exist in reverse order form as compared to that of pattern.

E.g. *input- AT*
*Pattern- TA*

But, such comparison will be required only if pattern matches. Thus over all we find following result: Say, comparisons found over pattern lengths in general are '*n*'. By our methods we reduce them to halves *i.e.,* '*n/2'*. Further adding the single comparisons if our pattern matched: *n/2 + p*. Where *p* is length of pattern, which is generally quite small. Thus taking *p->0*. We get total number of comparisons is *n/2*. The conversion of input can be done on the fly or while creation of index table.

## 3.4. Trivial Cases in Comparisons

*Case i:* If $S = \phi$ *i.e.,* $|S| = 0$ and $P = \phi$ *i.e.,* $|P| = 0$ then the number of occurrences of *P* in *S* is 0.
*Case ii:* If $S = \phi$ *i.e.* $|S| = 0$ and for any $|P| \geq 0$ then the number of occurrences of *P* in *S* is 0.
*Case iii:* If $S \neq \phi$ *i.e.,* $|S| \neq 0$ and for any $|P| = 0$ then the number of occurrences of *P* in *S* is 0.
*Case iv:* If $S \neq \phi$ *i.e.,* $|S| \neq 0$, $P \neq \phi$ *i.e.,* $|P| \neq 0$ and $|S| \leq |P|$ then the number of occurrences of *P* in *S* is 0.

3.4. To understand the algorithm assume a string *S=AGAATGCAGCTACAAGGTTCCATTCTGTCTCGCACTA* of 37 characters and pattern *P= ATGCAG*. Therefore the string can be viewed as follows in an indexing table.

Table.2. Index values of *A,C,G* and *T* sequence characters

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| ***T 0*** | 5 | 11 | 18 | 19 | 23 | 24 | 26 | 28 | 30 | 36 |
| ***A 1*** | 1 | 3 | 4 | 8 | 12 | 14 | 15 | 22 | 34 | 37 |
| ***G 2*** | 2 | 6 | 9 | 16 | 17 | 27 | 32 | | | |
| ***C 3*** | 7 | 10 | 13 | 20 | 21 | 25 | 29 | 31 | 33 | 35 |

As *'A'* being our first character of pattern the target indexes are 1, 3, 4, 8, 12, 14, 15, 22, 34 and 37.
Here *S2* and *P2* refer to combination of two characters of input string and pattern respectively. *S* and *P* refer to whole input and pattern s1. First we begin at index 1 because '*A*' is starting from index 1. We then form 2-groups of input and pattern both.

*i.e., S2 = A+G*
*P2 = A+T*

Clearly *S2!= P2* therefore *S!= P*. So we skip and go to next index.

2. At index 3 we get another probable match. We form 2-groups of input and pattern both.

*i.e., S2 = A+A*
*P2 = A+T*

Again we find *S2!=P2*, so we can match directly from next index.

4. Next we move to index 4. Here,

*S2 = A+T*
*P2 = A+T*

So we get *S2=P2*, we move further to next subgroup,

*S2 = G+C*
*P2 = G+C*

As S2=P2 we proceed further,

*S2=A+G*
*P2=A+G,*

As all subgroups have matched we go for checking order in our subgroups. In case of first subgroup, we find character in same order as pattern, so we go for next subgroup. Here also characters are in same order as per pattern. Same follows up to the last subgroup .So we do three more comparisons and over all in 6 comparisons we are getting our pattern matched.
Thus *S*=*P*. We now proceed to next index.

5. Next we move to index 8. Here,

*S2 = A+G*
*P2 = A+T*

Clearly *S2!=P2*. Thus we conclude *S!=P* and move to further index.

6. However at 12, we find

*S2 = A+C*
*P2 = A+T.*

Here too we find *S2!=P2* giving us *S!=P*. We check for next index now.

7. At index 14,

*S2 = A+A*
*P2 = A+T*

So *S2!=P2*. Without further checking we skip to next index.

8. Next at index 15,

*S2 = A+G*
*P2 = A+T*

Again we have *S2!=P2*. We need not check further and continue our search from next index.
9. Next at index 22,

*S2 = A+T*
*P2 = A+T*

We find successful match in this subgroup so we check for next subgroup too,

*S2=T+C*
*P2=G+C*

But here we find mismatch *i.e., S2!=P2*. Without checking further we can skip to next index.

10. However at next index *i.e.,* 34 we find that remaining length of input string *S* is 4 characters, while our pattern string *P*'s length is 6 characters. Therefore it is not possible to match pattern with sequence. So we skip remaining comparisons.

**Proof:** Let N=Input String say, ATTTGACCTTGAAA...

By converting the string to equivalent numerical sequence using formula,

*N[i] = (N[i] – 64) % 5, i* = Length of Input.

Now we apply same to Pattern P,

*P[i] = (P[i] – 64) % 5*, *i* = Length of Pattern.

First we prepare *P,*

*P[j] = P[i] + P[i+1]*

*j++, i+2*

Where *P*' is another array of length half that of *P*.

Now we process N,

2Sum = *N[i]+N[i+1],* where *i*<length of *P*

Compare (P'[j],2Sum)

Where Compare function compares the two quantities and breaks the whole operation if it find mismatch.

Thus we see effectively maximum number of comparisons require.

Max (length of*(P'),* (length of*(N))/2);*

in case of even comparisons and

Max (length of*(P'),* (length of*(N))/2) + 1*;

in case of odd comparisons. Also the comparisons are finally going to end as length of N is finite.

## 4. Experimental Results and Discussions

In this section we present several experiments comparing our algorithm to the existing algorithms and evaluating with the number and size of patterns on the performance. Each experiment was performed on different pattern sizes and the comparison results are noted. The text file which we used for our experiments was a collection of 1024 nucleotide sequence characters. From the below figure we can draw the following conclusions. As the size of the pattern increases the number of comparisons increases but in the proposed technique as the size increases the number of comparisons decreases in some of the cases. The patterns are randomly chosen from the given file size of 1024 characters.

4.1. The below DNA sequence dataset has been taken for the testing of 2-jump algorithm .The DNA biological sequence S∈∑*of size n=1024 and pattern P∈∑*. Let S be the following DNA sequence.

"AGAACGCAGAGACAAGGTTCTCATTGTGTCTCGC
AATAGTGTTACCAACTCGGGTGCCTATTGGCCTCC
AAAAAAGGCTGTTCAACGCTCCAAGCTCGTGACCT
CGTCACTACGACGGCGAGTAAGAACGCCGAGAAG
GTAAGGGAACTAATGACGCGTGGTGAATCCTATG
GGTTAGGATCGTGTCTACCCCAAATTCTTAATAAA
AAACCTAGGACCCCCTTCGACCTAGACTATCGTAT
TATGGACAAGCTTTAACTGTCGTACTGTGGAGGCT
TCAAAACGGAGGGACCAAAAAATTTGCTTCTAGC
GTCAATGAAAAGAAGTCGGGTGTATGCCCCAATTC
CTTGCTGCCCGGACGGCCAGTTCATAATGGGACAC
AACGAATCGCGGCCGGATATCACATCTGCTCCTGT
GATGGAATTGCTGAATGCGCAGGTGTGCTTATGTA
CAATCCACGCGGTACTACATCTTGTCTCTTATGTA
GGGTTCAGTTCTTCGCGCAATCATAGCGGTACGAA
TACTGCGGCTCCATTCGTTTTGCCGTGTTGATCGG
GAATGCACCTCGGGGACTGTTCGATACGACCTGGG
ATTTGGCTATACTCCATTCCTCGCGAGTTTTCGATT
GCTCATTAGGCTTTGCGGTAAGTAAGTTCTGGCCA
CCCACTTCGAGAAGTGAATGGCTGGCTCCTGAGCG
CGTCCTCCGTACAATGAAGACCGGTCTCGCGCTAA
ATTTCCCCCAGCTTGTACAATAGTCCAGTTTATTAT
CAAAGATGCGACAAATAAATTGATCAGCATAATC
GAAGATTGCGGAGCATAAGTTTGGAAAACTGGGA
GGTTGCCAGAAAACTCCGCGCCTACTTTCGTCAGG
ATGATTAAGAGTATCGAGGCCCCGCCGTCAATACC
GATGTTCTTCGAGCGAATAAGTACTGCTATTTTGC
AGACCCTTTGCCAGGCCTTGTCTAAAGGTATGTTA
CTTAATATTGACAATACATGCGTATGGCCTTTTCC
GGTTAACTCCCTG".

The index table (*index Tab[4][1024]*) for sequence *S* is very large in number of DNA sequence characters . For different patterns sizes which has been chosen randomly from the above DNA sequence the number of occurrences and the number of comparisons is shown in the Table. 3. To check whether the given pattern is present in the sequence or not we need an efficient algorithm with less comparison time and complexity. By the current technique different patterns are analyzed and the graph is plotted by using these results and analyzed accordingly. From the below experimental results, improvement can be seen that 2-JUMP algorithm gives good performance compared to the some of the popular methods shown in the Table.4. Here we have taken five fields in the Table .3. The pattern text, number of characters in the pattern, number of occurrences of a pattern, the proposed method and the number of comparisons and comparisons per character. The number of comparisons per character (CPC ratio) 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 .3.Experimental results analysis of 2-jump algorithm

| ***S.No*** | ***Pattern*** | ***Patten Length*** | ***No. of Occur*** | ***2-jump*** | ***CPC*** |
|---|---|---|---|---|---|
| 1 | *A* | 1 | 259 | 259 | 0.2 |
| 2 | *AG* | 2 | 53 | 312 | 0.3 |
| 3 | *CAT* | 3 | 11 | 335 | 0.3 |
| 4 | *AACG* | 4 | 5 | 434 | 0.4 |
| 5 | *AAGAA* | 5 | 2 | 441 | 0.4 |
| 6 | *AAAAAA* | 6 | 3 | 456 | 0.4 |
| 7 | *AGAACGC* | 7 | 2 | 379 | 0.3 |
| 8 | *AAAAAAGG* | 8 | 1 | 460 | 0.4 |
| 9 | *GCTCATTAG* | 9 | 1 | 390 | 0.3 |
| 10 | *CCTTTTCCGG* | 10 | 1 | 377 | 0.3 |
| 11 | *TTTTGCCGTGT* | 11 | 1 | 431 | 0.4 |
| 12 | *TTCTTAATAAAA* | 12 | 1 | 435 | 0.4 |
| 13 | *GGGACCAAAAAAT* | 13 | 1 | 392 | 0.3 |
| 14 | *TTTTGCCGTGTTGA* | 14 | 1 | 432 | 0.4 |
| 15 | *CCTCCAAAAAAGGCT* | 15 | 1 | 382 | 0.3 |
| 16 | *GGCTGTTCAACGCTCC* | 16 | 1 | 392 | 0.3 |
| 17 | *TTTTCGATTGCTCATTA* | 17 | 1 | 432 | 0.4 |
| 18 | GGGATTTGGCTATACTCC | 18 | 1 | 395 | 0.3 |
| 19 | *GGCCTTGTCTAAAGGTATG* | 19 | 1 | 393 | 0.3 |
| 20 | *CCTGAGCGCGTCCTCCGTCA* | 20 | 1 | 382 | 0.3 |

From the below Table.4. results analysis it has been observed the following in terms of relative performance of our algorithm with some of existing algorithms. To measure the performance of the proposed algorithm with the existing popular algorithm we have used two parameters like CPC (Character per comparison ratio) and number of comparisons which are shown in Table.4. The proposed algorithm gives good performance with the algorithms like MSMPMA, Brute-force, Tri-Match, IKPMPM and Naïve string matching algorithms. From the Table.4. We have taken different pattern sizes from 1 to 16 and analyzed accordingly. In all the different cases the proposed technique gives better performance with existing algorithms.

Table .4.Comparisons of different algorithms with 2-jump

| **Pattern** | **2-JUMP** | | **IBKPMPM** | | **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*** |
| *A* | 259 | 0.2 | 259 | 0.2 | 1024 | 1.0 | 1024 | 1.0 | 1025 | 1.0 | 1024 | 1.0 |
| *AG* | 312 | 0.3 | 518 | 0.5 | 1230 | 1.2 | 1282 | 1.2 | 1284 | 1.2 | 1281 | 1.2 |
| *CAT* | 335 | 0.3 | 542 | 0.5 | 1298 | 1.2 | 1318 | 1.2 | 1321 | 1.2 | 1310 | 1.2 |
| *AACG* | 434 | 0.4 | 614 | 0.6 | 1359 | 1.3 | 1376 | 1.3 | 1380 | 1.3 | 1376 | 1.3 |
| *AAGAA* | 441 | 0.4 | 607 | 0.5 | 1375 | 1.3 | 1388 | 1.3 | 1393 | 1.3 | 1387 | 1.3 |
| *AAAAAAGG* | 460 | 0.4 | 623 | 0.6 | 1394 | 1.3 | 1409 | 1.3 | 1417 | 1.3 | 1407 | 1.3 |
| *TTCTTAATAAAA* | 435 | 0.4 | 634 | 0.6 | 1390 | 1.3 | 1390 | 1.3 | 1402 | 1.3 | 1399 | 1.3 |
| *GGCTGTTCAACGCTCC* | 392 | 0.3 | 580 | 0.5 | 1349 | 1.3 | 1349 | 1.3 | 1365 | 1.3 | 1349 | 1.3 |

Fig.1. Shows comparison of different algorithms with 2-JUMP.The proposed algorithm outperforms when compared with some of the popular algorithms. The current technique gives good performance in reducing the number of comparisons compared with other algorithms. The dotted line shows the 2-jump proposed model where as MSMPMA, Brute-Force, Trie-matching IKPMPM and Naïve string searching are shown by solid lines. From the below graph towards the X-axis we have the pattern size whereas towards Y-axis shows the number of comparisons. If we see the experimental analysis all the other algorithms will gives more than 1000 comparisons where as the proposed technique gives less than 500 comparisons due to the indexed method.

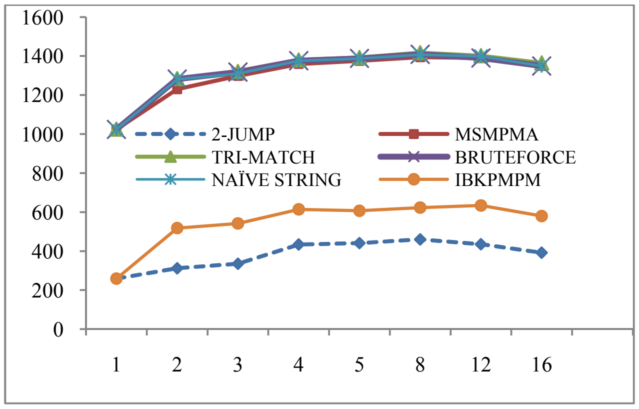

Fig.1. Comparison of different algorithms with 2-JUMP.

The following are observed from the experimental results.

- Reduction in number of comparisons.
- The ratio of comparisons per character has gradually reduced and is less than 1.
- Suitable for unlimited size of the input file.
- Once the indexes are created for input sequence we need not create them again.
- For each pattern we start our algorithm from the matching character of the pattern which decreases the unnecessary comparisons of other characters.
- It gives good performance for DNA related sequence applications.

### Applications in Bioinformatics

Different biological problems of bioinformatics involve the study of genes, proteins, nucleic acid structure prediction, and molecular design.

- Alignment and comparison of DNA, RNA, and protein sequences.
- Gene mapping on chromosomes.
- Gene finding and promoter identification from DNA sequences.
- Interpretation of gene expression and micro-array data.
- Gene regulatory network identification.
- Construction of phylogenetic trees for studying evolutionary relationship.
- DNA and RNA structure prediction.
- Protein structure prediction and classification.
- Molecular design.
- Organize data and allow researchers to access existing information and submit new entries.
- Develop tools and resources which are used for analysis and management of biological data.
- Use sequence data to analyze and interpret the results in a biologically meaningful manner.
- To help researchers in the pharmaceutical industry in drug design process.
- Finding similarities among strings such as proteins of different organisms.
- Finding similarities among parts of spatial structures.
- Constructing of phylogenetic trees called the evolution of organisms.
- Classifying new data according to previously clustered sets of annotated data.

## 5. Conclusion

In this paper we have proposed a new algorithm for DNA pattern matching called 2-jump index based search for DNA pattern matching. The proposed technique enhances the comparison time and the CPC ratio when compared with some of the popular techniques. The proposed algorithm is implemented, analyzed, tested and compared. The experimental result shows that there is a large amount of performance improvement due to this the overall performance increases.

## References

[1] Aho, A. V., and M. J. Corasick, ‘‘Efficient string matching: an aid to bibliographic Search, ’’ Communications of the ACM **18** (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", Proc. of International Conference on Advances in Computer Science 2010 pp 83-87.
[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.

Raju Bhukya has received his B.Tech in Computer Science and Engineering from Nagarjuna University in the year 2003 and M. Tech degree in Computer Science and Engineering from Andhra University in the year 2005. He is currently working as an Assistant Professor in the Department of Computer Science and Engineering in National Institute of Technology, Warangal, Andhra Pradesh, India. He is currently working in the areas of Bio-Informatics.

Somayajulu DVLN has received his M. Sc and M. Tech degrees from Indian Institute of Technology, Kharagpur in 1984 and in 1987 respectively, and his Ph. D degree in Computer Science & Engineering from Indian Institute of technology, Delhi in 2002. He is currently working as Professor and Head of Computer Science & Engineering at National Institute of Technology, Warangal. His current research interests are bio-informatics, data warehousing, database security and Data Mining.