Frequent Words with Mismatches and Reverse Complements Problem

KALLA LIKHIT SAI ESWAR
AM.EN.U4AIE20137
CSE-AI dept
Amrita School of Engineering
Amritapuri, Kerala, India
amenu4aie20137@am.students.amrita.edu

***Abstract*—In that oriC region, the DNA A boxes are invisible and by using the minimum skew problem we can identify the oriC region. DnaA cannot only bind to perfect DNA A boxes but also bind to their slight variations like Mismatches and reverse complement. Our objective is to find an algorithm that will find those mismatches. From the frequent words in a pattern algorithm, and the concept of Reverse complement can be used for this. First, the reverse complement of the given DNA sequence is done then the neighbors are found, and the obtained sequence is passed to the frequent word's algorithm. Which then return the k-mers with the given particular hamming as the output.**

Keywords—Reverse Complement, Mis Matches, Hamming Distance, K-mers.

I. INTRODUCTION

Bioinformatics is a field of computational science that has to do with the analysis of sequences of biological molecules. It usually refers to genes, DNA, RNA, or protein, and is particularly useful in comparing genes and other sequences in proteins or any other sequences within an organism, looking at evolutionary relationships between organisms, and using the patterns that exist across DNA and protein sequences to figure out what their function is.

Replication of DNA is the process in which DNA makes copies of itself. The oriC region is the site where the DNA replication begins. The DNAA protein plays a role in promoting the unwinding of DNA in the oriC. The concentration of DnaA protein determines the initiation phase of DNA replication. As replication begins, active DnaA binds to 9-mer repeats upstream of oriC called DNA box.

The Escherichia coli (E. Coli) genome consist of circular DNA and 46,00,000 base pairs. Solving the Minimum Skew Problem provided an approximate location of oriC at position 3923620 in E. coli.

From the case of Vibrio Cholerae, it is observed that a DNA A box may appear with slight variations like mismatches and reverse compliments which can be solved by using Frequent Words with Mismatches and Reverse Complements Problem which help us to find DNA A Boxes in E. Coli The experimentally confirmed DnaA box in E. coli (TTATCCACA) is a most frequent 9-mer with 1 mismatch, along with its reverse complement TGTGGATAA which matched with our output.

II. LITERATURE REVIEW

A. **Bioinformatics** is an extremely encouraging field that has countless applications in the aspects of agribusiness, microbiology, biomedicine etc.

B. **Hogeweg and Hesper, 1970; Hogeweg,1978, 2011** - made up the term bioinformatics as the investigation of data measures in biotic frameworks. This definition set bioinformatics in equal connection to biophysics which manages the investigation of actual cycles in natural frameworks or biochemistry.

C. **Frederick Sanger** - Computers became significant in atomic science when protein arrangements became available after decided on the grouping of insulin in the mid-1950s. Comparing numerous successions physically ended up being unreasonable.

D. **Margaret Oakley Dayhoff** - a pioneer in bioinformatics, who has been hailed by David Lipman, director of the National Center for Biotechnology Information, as the "mother and father of bioinformatics, (Moody, 2004), arranged one of the first protein sequence databases, at first distributed as books and spearheaded strategies for succession alignment and sub-atomic development.

III. METHODOLOGY

For finding DnaA boxes that bind to their slight variations like Mismatches and reverse complements, we redefine the Frequent Words Problem's algorithm to account for both mismatches and reverse complements.

1. Number of Mismatches or the Hamming distance 'd'

The number of positions that two DnaA sequences of the same length differ or the number of mismatches between the two DnaA sequences is the Hamming distance.

By modifying the Frequent Words algorithm in order to find DnaA boxes by identifying frequent k-mers, possibly with mismatches. We say that position i in k-mers p1 · · · pk and q1 · · · qk is a mismatch if pi != qi. The number of mismatches between strings p and q is called the Hamming distance between these strings and is denoted HAMMINGDISTANCE(p, q).

Algorithm:

```
Count ← 0
for i = 0 to |a| do
    if a[i] != b[i] then
      Count ← Count + 1
return Count
```

2. Find Frequent Words with Mismatches and Reverse Complements

The most frequent k-mer with up to d mismatches in Text is simply a string Pattern maximizing COUNTd(Text, Pattern) among all k-mers.

Algorithm:
FindingFrequentWordsWithMismatchesAndReverseComplementsBySorting(Text, k, d)

```
    FrequentPatterns ← an empty set
    Neighborhoods ← an empty list
    for i = 0 to |Text| − k do
        add NEIGHBORS(Text(i,k), d) to
    Neighborhoods
        add NEIGHBORS(Text(i, k), d) to
    Neighborhoods
    form an array NeighborhoodArray holding all
    strings in Neighborhoods
    for i = 0 to |Neighborhoods| − 1 do
        Pattern ← NeighborhoodArray(i)
        Index(i) ←
        PATTERNTONUMBER(pattern)
        Count(i) ← 1
    SortedIndex ← SORT(Index )
    for i = 0 to |Neighborhoods| − 2 do
        if SortedIndex(i) = SortedIndex(i+1)
    then
            Count(i+1) ← Count(i)+1
    maxCount ← maximum value in array Count

    for i = 0 to |Neighborhoods| − 1 do
        if Count(i) = maxCount then
            Pattern ←
            NUMBERTOPATTERN(SortedIndex(i), k)
            add Pattern to
            FrequentPatterns
    return FrequentPatterns
```

3. NEIGHBORS (Pattern, d)

Given a k-mer Pattern, we, therefore, define its d-neighbourhood NEIGHBORS(Pattern, d) as the set of all k-mers that are close to Pattern.

Algorithm:

NEIGHBORS (Pattern, d)

```
    if d = 0 then
        return {Pattern}
    if |Pattern| = 1 then
        return {A,C,G,T}
    Neighborhood ← an empty set
    SuffixNeighbors ←
    NEIGHBORS(Suffix(Pattern), d)
    for each string Text from SuffixNeighbors do
        if
     HAMMINGDISTANCE(Suffix(Pattern),
     Text) < d then
            for each nuceotide x do add x
        + Text to Neighborhood
        else
            add FirstSymbol(Pattern) +
        Text to Neighborhood
    return Neighborhood
```

4. Reverse Complements

Given a nucleotide p, we denote its complementary nucleotide as p. The reverse complement of a string Pattern = p1 ⋯ pn is the string Pattern = pn ⋯ p1 formed by taking the complement of each nucleotide in Pattern, then reversing the resulting string. We will need the solution to the following problem throughout this chapter.

Algorithm:

Reverse Complement:

rText ← reverse
Text ← an empty string
for i = 0 to |rText| − 1 **do**
Text(i) ← complement of rText(i)
return *Text*

5. PATTERNTONUMBER(Pattern)

Our approach to computing PATTERNTONUMBER(Pattern) is based on a simple observation. If we remove the final symbol from all lexicographically ordered k-mers, the resulting list is still ordered lexicographically (think about removing the final letter from every word in a dictionary). In the case of DNA strings, every (k-1) mers in the resulting list is repeated four times.

Algorithm:

PATTERNTONUMBER(Pattern):

if Pattern contains no symbols **then**
return 0
symbol ← LastSymbol(Pattern)
Prefix ← PREFIX(Pattern)
return PATTERNTONUMBER(Prefix) + SYMBOLTONUMBER(symbol)

6. NUMBERTOPATTERN(index, k)

In order to compute the inverse function NUMBERTOPATTERN(index, k), we return to above, which implies that when we divide index = PATTERNTONUMBER(Pattern) by 4, the remainder will be equal to SYMBOLTONUMBER(symbol), and the quotient will be equal to PATTERNTONUMBER(PREFIX(Pattern)). Thus, we can use this fact to peel away symbols at the end of Pattern one at a time.

Algorithm:

NUMBERTOPATTERN(index, k):

if k = 1 **then**
return NUMBERTOSYMBOL(index)
prefixIndex ← QUOTIENT(index, 4)
r ← REMAINDER(index, 4)
PrefixPattern ← NUMBERTOPATTERN(prefixIndex, k-1)
return concatenation of PrefixPattern with *symbol*

IV. RESULT

When we give a dataset (Fig 1.) is uploaded containing the DNA sequence along with length of the k-mers ‘k’ and the hamming distance ‘d’, we obtain the output (Fig 2.) after executing the code. Also solved the Rosalind Problem (Fig 3.)

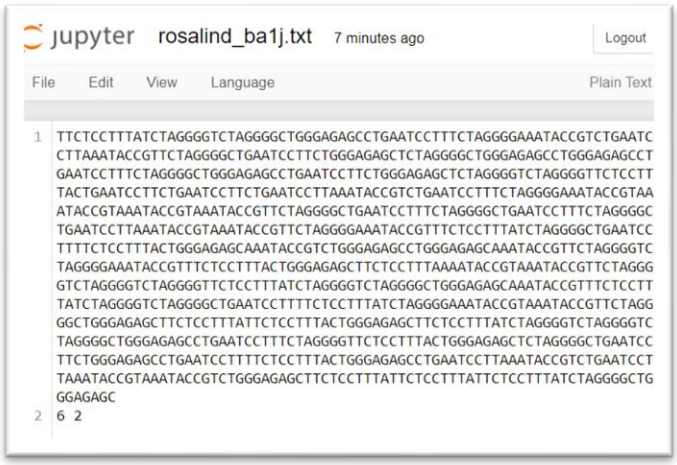

Fig 1. Dataset ‘rosalind_ba1j.txt’

Fig 2. Output

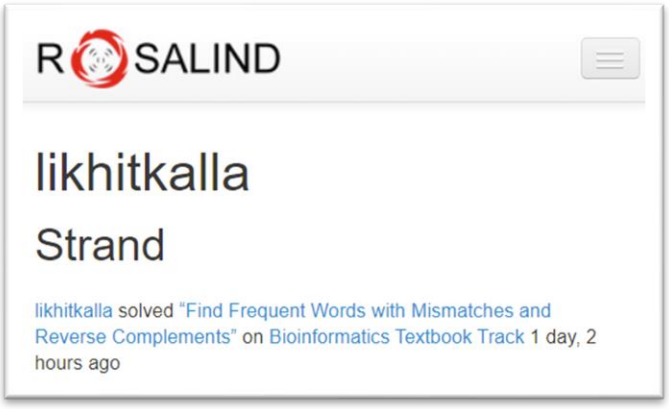

Fig 3. Rosalind Profile Strand

V. CONCLUSION

By improvising the frequent words problem, i.e. the Frequent Words with Mismatches and Reverse Complement Problem, we were able to locate DnaA boxes in the oriC region of the DNA sequence. As a result of experimentally derived DnaA boxes, we obtain the most frequent k-mer of length ‘k’ with ‘d’ mismatch, along with its reverse complement.

VI. References

[1] Compeau, Phillip, and Pavel Pevzner. Bioinformatics algorithms: an active learning approach. Active Learning Publishers, 2015.

[2] 8 Finding Frequent Words with Mismatches.pptx (sharepoint.com)

[3] ROSALIND | Find Frequent Words with Mismatches and Reverse Complements

[4] FrequentWordsWithMismatchesAndReverseComplementsProblem.pdf (bioinformaticsalgorithms.com)

[5] BA-chap1and2.pdf (byu.edu)

[6] Algorithms: Frequent Words with Mismatches and Reverse Complements Problem (bioinformaticsalgorithms.blogspot.com)

VII. Appendix

```
from collections import defaultdict

def HammingDistance(seq1,seq2):
    d=0
    for i in range(len(seq1)):
        if seq1[i]!=seq2[i]:
            d+=1;
    return d

def ReversePattern(pattern):
    temp=pattern.replace("A","X").replace("T","A").replace("X","T")
    return (temp.replace("G","X").replace("C","G").replace("X","C")[::-1])

def neighbour(pattern, mismatch, words):
    if mismatch == 0:
        words.add(pattern)
    else:
        bases = ['A', 'T', 'C', 'G']
        for i in range(len(pattern)):
            for j in range(len(bases)):
                new_pattern = pattern[:i] + bases[j] + pattern[i+1:]
                if mismatch <= 1:
                    words.add(new_pattern)
                else:
                    neighbour(new_pattern, mismatch-1, words)

def FindMostFrequentPattern(text, k, d):
    allfrequentwords = defaultdict(int)
    for i in range(len(text) - k + 1):
        frequentwords = set()
        neighbour(text[i:i + k], d, frequentwords)

        for words in frequentwords:
            allfrequentwords[words] += 1

    for t in allfrequentwords.keys():
        reverse_k = ReversePattern(t)
        for i in range(len(text) - k + 1):
            if HammingDistance(text[i:i + k], reverse_k) <= d:
                allfrequentwords[t] += 1

    result = set()
    for t in allfrequentwords.keys():
        if allfrequentwords[t] == max(allfrequentwords.values()):
            result.add(t)
            result.add(ReversePattern(t))
    for i in result:
        print(i, end=" ")

in_file = open('rosalind_ba1j.txt', 'r')
text = in_file.readline().strip()
k, d = map(int, in_file.readline().strip().split())
FindMostFrequentPattern(text, k, d)
```