Greedy Motif Search

Muhammed Shajahan
S2 AIE 'B'
AM.EN.U4AIE21144

Navneeth Suresh
S2 AIE 'B'
AM.EN.U4AIE21147

Aswin US
S2 AIE 'B'
AM.EN.U4AIE21119

Akshay Krishnan T
S2 AIE 'B'
AM.EN.U4AIE21109

Niranjan Prasanth
S2 AIE 'B'
AM.EN.U4AIE21148

Abstract-DNA motif discovery is an important problem in bioinformatics and it is essential for identifying transcription factor binding sites that play key role in the gene expression process. Motifs are generally short patterns which repeat among a set of DNA sequences. However, it is computationally expensive and impractical to find them with exhaustive search. Therefore, probabilistic and heuristic approaches can be used for tackling this problem. This study focuses on greedy construction algorithms for finding DNA motifs. The first instances of k-mers are taken and their profile matrices are calculated to find the motifs. The results obtained show the effectiveness and usability of the proposed method for the DNA motif discovery problem.

Keywords-Motifs, transcription factor, k-mer, Profile matrix, DNA, Greedy

I. INTRODUCTION

Due to expanding biological research needs and quick technology advances in gathering such data, the use of genetic data for biological research has been expanding quickly. Sequencing machines read DNA sequences (and other types of sequences) as part of the traditional method of creating raw genetic data. The output of current sequencing machines is typically a very large in number (tens of millions) of short strings of characters.

Each of these strings, referred to as "reads," corresponds to a brief segment of a lengthy DNA sequence. Since the DNA is made up of pairings of four nucleic acids, the reads that represent its many components are strings made up of the four potential characters/letters A, C, G, and T, each of which stands for a different nucleic acid.

A species' individual members each have a unique DNA sequence (moreover, each human has two versions of most of his or her chromosomes and, in fact, small variations may exist among the many cells in a human body). The following scenario is frequently seen in the analysis of such differences: one is interested in comparing the DNA of one specific member of the species to the reference, and one possesses some "reference" .

'Motif discovery' (or 'motif finding') in biological sequences can be defined as the problem of finding short similar sequence elements (building the 'motif') shared by a set of nucleotide or protein sequences with a common biological function. The identification of regulatory elements in nucleotide sequences, like transcription factor binding sites (TFBSs), has been one of the most widely studied flavors of the problem, both for its biological significance and for its bioinformatic hardness [1, 2].

This first step of gene expression, 'transcription', is finely regulated by a number of different factors, among which 'transcription factors' (TFs) play a key role binding DNA near the transcription start site of genes (in the 'promoter' region), but often also within the region to be transcribed or in distal elements like 'enhancers' or 'silencers' [3, 4]. The actual DNA region interacting with and bound by a single TF (called TFBS) usually ranges in size from 8–10 to 16–20 bp. TFs bind the DNA in a sequence-specific fashion, that is, they recognize sequences that are similar but not identical, differing in a few nucleotides from one another.

The basic idea of the greedy motif search algorithm is. to find the set of motifs across a number of DNA. sequences that match each other most closely. Here the greedy algorithm has some qualities like and it does no search for all possibilities of a particular thing we are searching. Greedy will only consider some initial conditions and will always pursue in that direction while searching. Of course it may or may or may not lead us to the global solution, but definitely we will get an important local solution which can resemble the global solution. Especially when it comes to DNA, it doesn't matter whether it is a global solution or not, both are going to be similar.

Basically, greedy chooses the best in each case, and there is no backtracking, it is local solution and not an exhaustive search. Exhaustive search means it does not go and search for all possibilities, rather it will just consider one or two initial possibilities and come at a solution.

The major advantage is that this method uses very less time and results closely resembles the known motifs. Remaining algorithms, which search for all possibilities which we will get global solution takes hours or even days to get results since genomes are really long. So that is the reason why we are using motif discovery using Greedy since it takes very less time and results closely resembles the known motifs. We will be getting set of patterns which are

extremely similar and that's more than enough since we are only looking for motifs which we do not know how they look in prior.

II. THE MOTIF DISCOVERY PROBLEM

Here we have made use of python language for Greedy motif Algorithm. To evaluate motifs or common sequence parts in DNA, profile matrices can be constructed. A consensus string that represents the motif found is formed by taking the nucleotide that has the highest count for each column in the profile matrix. A greedy approach could be selecting the best possible solution component at a time. Then we find the best probability k-mer from the list of all k-mers and profile matrix. The greedy motif search algorithm's core premise is to locate the collection of motifs that most closely match each other across a number of DNA sequences. We have also made use of Laplace's rules in order to counter the zero issue.

In order to evaluate motifs or common sequence parts in DNA, profile matrices can be constructed. As a first step, we need to determine starting positions for each motif candidate and so on. Starting indices may take value between [0,n-l], where n is the sequence length and l is the motif length. After subsequences are extracted from each sequence, they are aligned and combined in an alignment matrix of size t x l, where l indicates motif length and t indicates the first motif candidate starting is extracted from sequence index 0.

III. PROBLEM STATEMENT

Given: Integers *k* and *t*, followed by a collection of strings *Dna*.

Return: A collection of strings *BestMotifs* resulting from running *GreedyMotifSearch*(*Dna, k, t*). If at any step you find more than one *Profile*-most probable *k*-mer in a given string, use the one occurring first.

IV. SAMPLE DATASET & OUTPUT

Sample Dataset

```
3 5
GGCGTTCAGGCA
AAGAATCAGTCA
CAAGGAGTTCGC
CACGTCAATCAC
CAATAATATTCG
```

Sample Output

```
CAG
CAG
CAA
CAA
CAA
```

IV. METHODOLOGY

A) Forming the DNA matrix:-

If a set of DNA or Single string DNA, Then to make the process easier we will be converting that into a matrix by dividing DNA at equal intervals.

B) Selecting the 1st seen k-mers to create profile matrix:-

COUNT(*Motifs*)

A:	2	2	0	0	0	0	9	1	1	1	3	0
C:	1	6	0	0	0	0	0	4	1	2	4	6
G:	0	0	10	10	9	9	1	0	0	0	0	0
T:	7	2	0	0	1	1	0	5	8	7	3	4

PROFILE(*Motifs*)

A:	.2	.2	0	0	0	0	.9	.1	.1	.1	.3	0
C:	.1	.6	0	0	0	0	0	.4	.1	.2	.4	.6
G:	0	0	1	1	.9	.9	.1	0	0	0	0	0
T:	.7	.2	0	0	.1	.1	0	.5	.8	.7	.3	.4

C) Selecting the kmer (motif_1) with highest score from 1st row.

The profile obtained earlier is used to evaluate each kmer in the 1st row of the DNA matrix

Eg: let a sequence be ACGTACG and k=3

From these sequence ACG,CGT,GTA,TAC… are taken. ie, all possible 3-mer from sequence is taken and sent to evaluation using the Profile matrix, we will be getting a value.

From the values obtained, one with the highest score is taken and that will be our 1st Motif and is added to the motif matrix

D) Updating the Motif matrix with a new motif (i):-

1st motif will get added, ie the motif matrix now

have only one entry
For Eg; let TAC be the the one with highest score from the above stated example
Then matrix will be [TAC].
Now we will be creating the profile matrix for [TAC]
Ie,we have a new profile matrix now.
This profile matrix will be used to evaluation of the kmer with highest score in 2nd Row of the DNA matrix just like we did for the 1st row.
Thus we will get a motif and that is added to the motif matrix.
Now using these two, we will be creating new count matrix and profile matrix. Using this updated profile matrix, we will be evaluating the best k-mer from the 3rd row and added to motif matrix.
This process will keep on adding.
So the number of rows the DNA matrix have, that many entries will be there in the motif matrix. I.e. that many motifs we will get.

E) Repeating this step for every other row:-
F) Returning the motif matrix which will contain
Motif_1, Motif _2, etc.

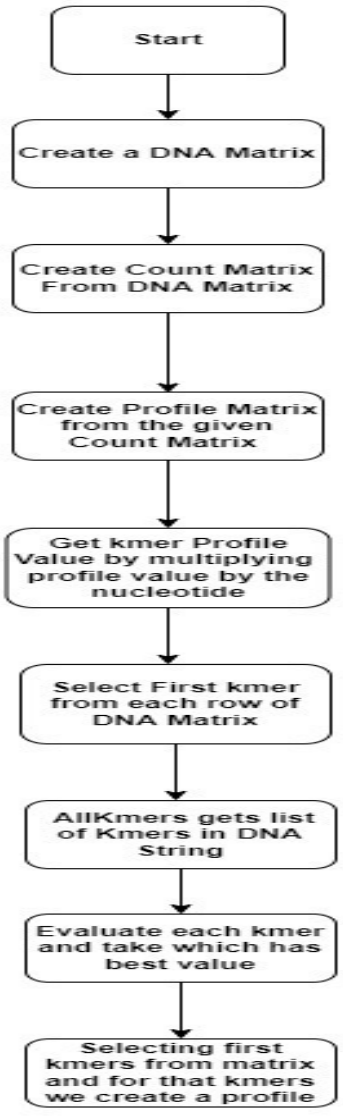
G) Algorithm :-
Greedy Motif Search Algorithm Our proposed greedy motif search algorithm, GreedyMotifSearch, tries each of the k-mers in DNA1 as the first motif. For a given choice of k-mer Motif1 in DNA1, it then builds a profile matrix Profile for this lone k-mer, and sets Motif2 equal to the Profile-most probable k-mer in DNA2. It then iterates by updating Profile as the profile matrix formed from Motif1 and Motif2, and sets Motif3 equal to the Profile-most probable k-mer in DNA3. In general, after finding i − 1 k-mers Motifs in the first i − 1 strings of DNA, GreedyMotifSearch constructs Profile(Motifs) and selects the Profile-most probable kmer from DNAi based on this profile matrix. After obtaining a k-mer from each string to obtain a collection of Motifs, GreedyMotifSearch tests to see whether Motifs outscores the current best scoring collection of motifs and then moves Motif1 one symbol over in DNA1, beginning the entire process of generating Motifs again.

H) Pseudo Code:-

```
def GreedyAlgorithm(k,dna,rowSize):
    Motifs → Empty List
    Matrix → Dna Matrix(dna,rowSize)
    First Kmers = Select First Kmer(k,Matrix)
    First Kmer Profile → Profile Matrix(CountMatrix(First Kmers))
    First Row All Kmers → AllKmers(Matrix[0],k)
    Motif1 → BestProbable Kmer( First Row All Kmers, First Kmer Profile)
    Motifs.append(Motif1)
    Count → 1

    for each row in DnaMatrix:
        Profile → Profile Matrix(CountMatrix(Motifs))
        Mot = Best Probability Kmer(All Kmers(Matrix[Count],k),Profile)
        Motifs.append(Mot)
        Count → count+1
    return Motifs
```

V. PROGRAM FLOW



VI. DISCUSSION

At first, a code was made that was successfully able to generate the neighborhood of the given string. However, a drawback was found in the code. The time complexity of the code was large when implemented on large datasets.
So, another code was made to reduce the time complexity for the same given logic of the first code that was made. In this code, the code size was reduced as much as possible. The program once again successfully generated the required Output

VII. IMPLEMENTATION

Function to create a DNA matrix and it will loop through that cuts the DNA at a desired rowsize and will append to the rows.

```
def DnaMatrix(dna,rowSize):
    DNA_Matrix = []
    for n in range (0,int(len(dna)/rowSize)):
        m = n+1
        row = dna[n*rowSize:m*rowSize]
        DNA_Matrix.append(row)
    return DNA_Matrix
```

Then we create a function CountMatrix, in which the matrix can only be created only if a DNA Matrix exists. We initialize counters and increment values accordingly and append to list, then we take transpose and return the count matrix

```
def CountMatrix(DNA_Matrix):
    DNA_Matrix = [[DNA_Matrix[j][i] for j in range(len(DNA_Matrix))] for i in range (len(DNA_Matrix[0]))]
    countA = 1
    countT = 1
    countC = 1
    countG = 1
    list = []
    for row in DNA_Matrix:
        for base in row:
            if base == "A":
                countA+=1
            if base == "T":
                countT+=1
            if base == "G":
                countG+=1
            if base == "C":
                countC+=1
        list.append(str(countA)+str(countT)+str(countG)+str(countC))
        countA = 1
        countT = 1
        countC = 1
        countG = 1
    Count_Matrix = [[list[j][i] for j in range(len(list))] for i in range (len(list[0]))]
    return Count_Matrix
```

Then we create a function Profile Matrix which is a sum total of rows of the count matrix and divide by the total. Here we convert to integer as we will get count matrix as string.

```
def ProfileMatrix(Count_Matrix):
    list=[]
    total=0
    for i in range(len(Count_Matrix)):
        list.append(Count_Matrix[i][0])
    for i in range(0,len(list)):
        list[i]=int(list[i])
    for e1 in range(0,len(list)):
        total = total+list[e1]

    Profile_Matrix = [[int(x)/total for x in lst] for lst in Count_Matrix]
    return Profile_Matrix
```

Then we create a function KmerProfileValue, which is basically, if there is a kmer of a specific k value and a profile matrix and it takes value from profile matrix and it multiplies and store the value and returns the result for that specific kmer.

```
def KmerProfileValue(kmer,Profile):
    multiply = 1

    count = 0

    for i in kmer:

        global nuc
        if i=='A':
            nuc = Profile[0][count]
        if i=='C':
            nuc = Profile[1][count]
        if i=='G':
            nuc = Profile[2][count]
        if i=='T':
            nuc = Profile[3][count]
        count = count+1
        multiply = multiply*nuc
    return multiply
```

Here is where the greedy part starts, we have to select the first kmer from each row of DNA Matrix.

```
def SelectFirstKmer(k,DNA_Matrix):
    list = []
    for i in DNA_Matrix:
        list.append(i[0:k])
    return list
```

Here if we give a dna string and length k which are all possible kmers , here to get the last kmer we use the above code length of dna string - (k+1).

```
def AllKmers(dna_string,k):
    i=0
    kmer_List=[]
    for x in range(0,len(dna_string)-k+1):
        row = dna_string[i:i+k]
        kmer_List.append(row)
        i=i+1
    return kmer_List
```

If all kmers and a profile matrix, we evaluate each kmer and take the best value.

```
def BestProbabilityKmer(All_kmers,Profile_matrix):
    goodVal=0

    dict={}

    for i in All_kmers:

        val = KmerProfileValue(i,Profile_matrix)
        dict[val] = i

        if val>goodVal:
            goodVal=val
    return dict[goodVal]
```

This is the main function where we create a main DNA matrix. We select the first kmers from that matrix and we create a profile for that kmer.

```
def GreedyAlgorithm(k,dna,rowSize):
    Motifs=[]

    Matrix = DnaMatrix(dna,rowSize)
    firstKmers = SelectFirstKmer(k,Matrix)
    firstKmerProfile=ProfileMatrix(CountMatrix(firstKmers))
    firstRowAllKmers=AllKmers(Matrix[0],k)
    Motif1=BestProbabilityKmer( firstRowAllKmers,firstKmerProfile)
    Motifs.append(Motif1)

    counter=1

    for i in range(0,len(Matrix)-1):
        profile = ProfileMatrix(CountMatrix(Motifs))
        Mot = BestProbabilityKmer(AllKmers(Matrix[counter],k),profile)
        Motifs.append(Mot)
        counter = counter+1

    return Motifs
```

VIII. DATASET

https://drive.google.com/file/d/1sIlYZIiw8NHeAedI8V8JsOelxUiqdne5/view?usp=sharing

IX. RESULTS

Our Greedy motif algorithm, when implemented, ran error-free. It is understood that Greedy search is effective in reducing the time taken to compute the algorithm when compared to Brute-Force Algorithm. It will find a similar set of patterns which can be eligible candidates of Motifs in
DNA.

X. CONCLUSION

Even though Programmable implementation of the Greedy Motif Algorithm was successful, it is seen that they have a polynomial time complexity of O (ln2 + nlt) where l is the length of the motif, n is the length of the DNA samples, and t is the number of DNA samples. Many time results are dependent on the initial random motif and it is unlikely to get a good result in a single run. Greedy algorithm is a good method to find motifs approximately.

XI. BIBLIOGRAPHY

https://bioinformaticsalgorithms.com/data/debugdatasets/motifs/GreedyMotifSearch.pdf

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9219366&tag=1

https://www.mrgraeme.com/greedy-motif-search/

https://www.researchgate.net/publication/46381439_An_Improved_Heuristic_Algorithm_for_Finding_Motif_Signals_in_DNA_Sequences

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6471678/