Neighbourhood of a String

Pranav Jayasankar Nair, AM.EN.U4AIE20056
Department Of Computer Science (Artificial Intelligence)
School Of Engineering,Amrita Vishwa VidyaPeetham
Clappana PO ,Kollam,Kerala,India
amenu4aie20056@am.students.amrita.edu

***Abstract*—In order to efficiently solve the frequent words with mismatches problem, we generate the d-neighbourhood of a string. The d-neighbourhood of a k-mer is the set of all k-mers with a maximum hamming distance "d" from the given k-mer. By creating this set of k-mers, we significantly reduce the run time of the frequent words with mismatches problem as we only have to consider k-mers in the neighbourhood while searching.**

***Index Terms*—d-neighbourhood,k-mer,hamming distance,frequent words with mismatches**

I. Introduction

In order to solve the frequent words with mismatches problem we have to consider 4^k k-mers. To prevent this, we first generate all k-mers that are close to a string. K-mers close to the given k-mer are called neighbours of the k-mer and this collection of close strings forms the neighbourhood of the string. Our goal is to define the d-neighbourhood of a string, where d is the maximum hamming distance the string can have from the original. Throughout we will consider both iterative and recursive approaches to generate the neighbourhood of a string and examine the efficiency of each.

II. Literature Review

The use of genetic information for biological research has been rapidly growing due the growing needs in biological research, coupled with rapid technological advancements in acquiring such data. The classical process of producing raw genetic data, known as "sequencing", involves reading DNA sequences (and other types of sequences) by "sequencing machines". The output of current high-throughput "next-generation" sequencing machines is typically a very large number (tens of millions) of short strings of characters. Each of these strings, called "a read", represents a small part a of long DNA sequence. The DNA is composed of pairs of 4 nucleic acids, therefore the reads that represent parts of the DNA are strings composed of 4 possible characters/letters, A, C, G and T , each representing a different nucleic acid. Different individual members of a species have different DNA sequences (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). In the analysis of such differences, the following situation is often encountered: one has some "reference" strings of characters representing the "typical" genome of a species (a "typical genome") and one is interested in comparing the DNA of one individual member of the species to the reference. This is done by sequencing a sample of the DNA of that particular individual member and comparing the result to the reference. Since the output of sequencing machines is given in the form of short strings, representing fragments of the DNA, the analytical process begins with deciding where each read "belongs". In other words, one estimates which interval in the reference strings corresponds to each of the reads. The process of estimating where each read "belongs" is known as "read alignment". The typical approach to alignment is to take each of the reads and find a location in the reference where the reference is very similar to the read. Many of the reads may be "perfectly aligned", meaning that they are identical to some substring of the in the reference. Because of the differences between individuals, it is clear that not all the reads.[2]

To generate the neighbourhood of a string, the Hamming Distance is necessary, as it acts as the measure of closeness of the neighbourhood. A k-mer Pattern appears as a substring of Text with at most d mismatches if there is some k-mer substring Pattern' of Text having d or fewer mismatches with Pattern, i.e., HAMMINGDISTANCE(Pattern, Pattern') ¡= d. But why do we need to find the neighbours of a k-mer? Usually, while calculating frequent words with mismatches use an array CLOSE of size 4k whose values we initialize to zero. In FREQUENTWORDSWITHMISMATCHES, CLOSE(i) = 1 whenever Pattern = NUMBERTOPATTERN(i, k) is close to some k-mer in Text. Once we have the neighbours, we can optimize this code my considering only close k-mers for the APPROXIMATEPATTERNCOUNT operation instead of considering all 4k possible k-mers.

Our idea for generating NEIGHBORS(Pattern, d) is as follows. If we remove the first symbol of Pattern (denoted FIRSTSYMBOL(Pattern)), then we will obtain a (k-1)mer that we consider as the suffix. Now, consider a (k-1)-mer Pattern' belonging to NEIGHBORS(SUFFIX(Pattern), d). By the definition of the d-neighbourhood NEIGHBORS(SUFFIX(Pattern), d), we know that HAMMINGDISTANCE(Pattern0, SUFFIX(Pattern)) is either equal to d or less than d. In the first case, we can add FIRSTSYMBOL(Pattern) to the beginning of Pattern' in order to obtain a k-mer belonging to NEIGHBORS(Pattern, d). In the second case, we can add any symbol to the beginning of Pattern' and obtain a k-mer belonging to NEIGHBORS(Pattern, d). Generating the neighbourhood

Identify applicable funding agency here. If none, delete this.

also proves useful in the brute force approach of motif finding. Brute force (also known as exhaustive search) is a general problem-solving technique that explores all possible candidate solutions and checks whether each candidate solves the problem. Such algorithms require little effort to design and are guaranteed to produce a correct solution, but they may take an enormous amount of time, and the number of candidates may be too large to check. A brute force approach for solving the Implanted Motif Problem is based on the observation that any (k, d)-motif must be at most d mismatches apart from some k-mer appearing in one of the strings of Dna. Therefore, we can generate all such k-mers and then check which of them are (k, d)-motifs.[2]

III. Methods

We will be using 2 different methods to solve this problem

A. Recursive method

```
NEIGHBORS(Pattern, d)
    if d = 0
        return {Pattern}
    if |Pattern| = 1
        return {A, C, G, T}
    Neighborhood ← an empty set
    SuffixNeighbors ← NEIGHBORS(SUFFIX(Pattern), d)
    for each string Text from SuffixNeighbors
        if HAMMINGDISTANCE(SUFFIX(Pattern), Text) < d
            for each nucleotide x
                add x • Text to Neighborhood
        else
            add FIRSTSYMBOL(Pattern) • Text to Neighborhood
    return Neighborhood
```

In the recursive method we use recursion on the newly made patterns while lowering the hamming distance to generate more patterns.

B. Iterative method

```
IMMEDIATENEIGHBORS(Pattern)
    Neighborhood ← the set consisting of single string Pattern
    for i = 1 to |Pattern|
        symbol ← i-th nucleotide of Pattern
        for each nucleotide x different from symbol
            Neighbor ← Pattern with the i-th nucleotide substituted by x
            add Neighbor to Neighborhood
    return Neighborhood

ITERATIVENEIGHBORS(Pattern, d)
    Neighborhood ← set consisting of single string Pattern
    for j = 1 to d
        for each string Pattern′ in Neighborhood
            add IMMEDIATENEIGHBORS(Pattern′) to Neighborhood
            remove duplicates from Neighborhood
    return Neighborhood
```

For the iterative method, we use 2 functions: Immediate neighbours and iterative neighbours. Immediate neighbours returns a set of all neighbours with a hamming distance of 1 from the pattern string. This is done be replacing each base in the string with one of the other 3 bases to create new patterns. Iterative neighbours gives the neighbours for any given hamming distance. To achieve this, we use the immediate neighbour algorithm repeatedly on every string in the neighbourhood. This is repeated d number of times to get all the string that have a hamming distance of d from the pattern string.

IV. Results

Using either method, we can obtain the neighbourhood of the string as show below. The code used to calculate the following values is given in the appendix. Calculating the neighbourhood of the string "ACG" with Hamming Distance d=2.

Iterative method:

```
Neighbours found (iterative method)
{'GCT', 'TCT', 'CCC', 'GTG', 'ATT', 'GAG', 'TCA', 'TCG', 'AAG', 'AGC', 'CAG', 'CTG', 'ATG', 'TGG'
, 'GCG', 'ACC', 'TCC', 'AAA', 'CCA', 'CCT', 'ACA', 'AGT', 'GCA', 'ACG', 'CGG', 'ACT', 'AGG', 'GCC
', 'AGA', 'GGG', 'TAG', 'ATC', 'ATA', 'TTG', 'CCG', 'AAC', 'AAT'}
```

Recursive method:

```
Neighbours found (recursive method)
{'TCC', 'TAG', 'CCT', 'CCC', 'GAG', 'ACG', 'GCC', 'TGG', 'AAC', 'CTG', 'AAT', 'CCA', 'AGA', 'GTG'
, 'GGG', 'AGG', 'TCT', 'CAG', 'CGG', 'AAA', 'ATA', 'GCA', 'ACA', 'ACC', 'TCG', 'ATC', 'TTG', 'ATT
', 'GCT', 'ACT', 'ATG', 'AGC', 'GCG', 'CCG', 'AGT', 'TCA', 'AAG'}
```

To calculate the better formula, the python time module was used and the following was observed.

```
================== RESTART: D:\Amrita\Bio\project\recursive neighbors.py ==================
Neighbours found (recursive method)
{'AAC', 'TCG', 'CCA', 'TAG', 'ATC', 'TTG', 'GAG', 'CCC', 'GGG', 'AGA', 'GCA', 'ATA', 'AAG', 'ATT'
, 'TGG', 'AGG', 'CCG', 'ACC', 'ACT', 'CGG', 'AAA', 'TCA', 'GCG', 'AGT', 'AAT', 'CTG', 'GCT', 'ACG
', 'GTG', 'TCT', 'GCC', 'ACA', 'ATG', 'CCT', 'AGC', 'CAG', 'TCC'}
--- 0.09496402740478516 seconds ---
Neighbours found (iterative method)
{'TCT', 'CCC', 'TAG', 'GCC', 'TCC', 'AAG', 'AGA', 'ACA', 'ACT', 'GCG', 'TGG', 'ATA', 'CCG', 'CTG'
, 'CGG', 'CCT', 'AAA', 'ACC', 'ATG', 'AGC', 'GCA', 'AGT', 'AAT', 'TCG', 'CCA', 'AGG', 'GTG', 'TCA
', 'AAC', 'CAG', 'GGG', 'ATT', 'GAG', 'ACG', 'GCT', 'ATC', 'TTG'}
--- 0.09517526626586914 seconds ---
```

V. Conclusion

By creating and testing the two algorithms used (iterative and recursive), we have successfully been able to obtain the required neighbourhood of a string. On testing with the time module, we observe that the recursive version of the algorithm takes less time (not by much but still significant enough) compared to the iterative version. We now have an efficient way of the neighbourhood of a string. This can now be applied to increase the efficiency of larger algorithms like the frequent words with mismatches and the greedy motif search problems.

VI. References

[1] Finding Hidden Messages in DNA,Philip Compeau and Pavel Pevzener, Active Learning Publishers 2015

[2] https://cpsc.yale.edu/sites/default/files/files/tr1453.pdf

VII. Appendix

Iterative Code

```
import time
def immediateNeighbors(pattern):
    neighborhood=set()

    for i in range(len(pattern)):
        symbol=pattern[i]
        bases = ['A', 'T', 'C', 'G']
        for nuc in bases:
            if nuc!=symbol:
                neighbor=pattern[:i]+nuc+pattern[i+1:]
                neighborhood.add(neighbor)
    return neighborhood



def iterativeneighbors(pattern,d):
    neighborhood=set()
    neighborhood.add(pattern)

    for i in range(d):
        temp=set()
        for string in neighborhood:
            for j in immediateNeighbors(string):
                if j not in neighborhood:
                    temp.add(j)
        for t in temp:
            neighborhood.add(t)
    return neighborhood


start_time=time.time()

print("Neighbours found (iterative method)")
print(iterativeneighbors("ACG",2))


print("--- %s seconds ---" % (time.time() - start_time))
```

Recursive Code

```
import time
words=set()
def neighbor(pattern, d):
    global words
    if d == 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 d <= 1:
                    words.add(new_pattern)
                else:
                    neighbor(new_pattern, d-1)


start_time=time.time()


neighbor("ACG",2)
print("Neighbours found (recursive method)")

print(words)

print("--- %s seconds ---" % (time.time() - start_time))
```