Assignment 2
Due date: 23 Jun 2020

1. Consider a length-20 read R and a genome T. We want to find all 2mismatch hits of R. We try two approaches:
   - Mismatch seed: A length-10 1-mismatch seed of R[1..10] (i.e. all 1mismatch patterns of R[1..10], i.e. 1^i01^j where i+j=9)
   - Gapped seed: Six length-20 weight-10 shapes $1^{10}0^{10}$, $1^50^51^50^5$, $1^50^{10}1^5$, $0^51^{10}0^5$, $0^51^50^51^5$, and $0^{10}1^{10}$.

   For each approach, can we recover all 2-mismatch hits of R in T? Also, for each position in T, how many hash entries do we need to verify? Among the two approaches, which one is better?

Answer:

①For Mismatch seed, if we want to find 2mismatch, we need two use 1^i01^j twice each time, it just like $1^i01^j1^i01^j$ but we can not find 2mismatch like $1^i001^j1^i1^j$ $1^i1^j001^i1^j$ and $1^i1^j1^i001^j$; But for Gapped seed, we can cover all of the situations.

②Mismatch seed: The total situations of 1^i01^j(i+j=9) are 10, for twice using, we can get 2*10=20;

Gapped seed: The total situations are 6 and just need once using;

③ The Gapped seed approach are more independent, the probability of having at. least one hit in a homologous region is higher, so the second one is better.

2. Given two DNA sequences S_1 and S_2 of length n and m, respectively, can you give an efficient dynamic programming algorithm that returns the number of possible optimal global alignments between S_1 and S_2? What is the time complexity of your algorithm?

Answer:

This is a multiple back-tracking problem, there are two steps of my algorithm:

①First of all, we have to do global alignments and get the filled V table;

②Second, when we do back-tracking starting from the last entry, we need to build a list to dynamically save the new roads' position we find and refresh the former roads' position. After back-tracking we just need to check the length of list and we can get the number of possible optimal global alignments between S_1 and S_2. So, totally we also just need to do one time back-tracking and we can get the number of all optimal global alignments.

For time complexity, step one need to fill in whole table, this step's time complexity equals to O(mn), step two just need do one time back-tracking, so his step's time complexity equals to O(n). So the time complexity of whole algorithm equals to O(mn+n)=O(mn).

Python codes: (multiple back-tracking.py)

```
import numpy as np

seq1 = ['t', 'g', 'a', 'c', 'a', 'a', 't', 'c', 'c', 'c']
seq2 = ['t', 'g', 'a', 'g', 'c', 'a', 't', 'g', 'g', 't']
alphabet = ['a', 'c', 'g', 't']
similarity_matrix = [[2, -1, -1, -1], [-1, 2, -1, -1], [-1, -1, 2, -1], [-1, -1, -1, 2]]
pergap_score = -1


def Needleman_Wunsch_algorithm_with_linear_gap_penalty(seq1, seq2, alphabet,
similarity_matrix, pergap_score):
    v_table = np.zeros((len(seq1) + 1, len(seq2) + 1))  # initialize the V table with zero
    for i in range(1, len(v_table)):
        v_table[i, 0] = pergap_score + v_table[i - 1, 0]
    for i in range(1, len(v_table[0])):
        v_table[0, i] = pergap_score + v_table[0, i - 1]
    for i in range(1, len(v_table)):
        for j in range(1, len(v_table[0])):  # For standard Smith-Waterman dynamic programing, we
need to fill in
            # the whole table
            for m in range(len(alphabet)):  # First of all, we need to know the character of the entry
now, so we find
                # it and give it to m and n
                if seq1[i - 1] == alphabet[m]:
                    break
            for n in range(len(alphabet)):
                if seq2[j - 1] == alphabet[n]:
                    break
            match = v_table[i - 1][j - 1] + similarity_matrix[n][m]  # Calculate the match case
            deletion = v_table[i - 1][j] + pergap_score  # Calculate the insertion case
            insertion = v_table[i][j - 1] + pergap_score  # Calculate the deletion case
            v_table[i][j] = max(match, deletion, insertion)  # Fill in the V[i][j] basing on four cases
    return v_table


# Run Needleman-Wunsch algorithm and get the V table
v_table = Needleman_Wunsch_algorithm_with_linear_gap_penalty(seq1, seq2, alphabet,
similarity_matrix, pergap_score)

# i,j is the position of last entry
i = len(v_table) - 1
j = len(v_table[0]) - 1
total_ways = [[i, j]]  # This list is used to dynamically save the new road's position and refresh the
former road's
# position when running back-tracking
count = 0  # Count the number of all optimal alignments
while count != len(total_ways):  # If the count is not equals to the list's length, it means there are
still some
    # alignments not finish the back-tracking
    for item in range(len(total_ways)):  # For each of the position in list, we need to do refreshing
or appending until
        # the back-tracking finish
        ways = []  # To save the entry's source, there have three cases: just one source, two sources
and three sources
        i = total_ways[item][0]  # Starting from the last entry
        j = total_ways[item][1]
        if i == j == 0:  # If there is a [0,0] appeared in list, it means there is one alignment finish the
```

```
        # back-tracking
        count += 1
        continue
    for m in range(len(alphabet)):  # First of all, we need to know the character of the entry now,
so we find it
        # and give it to m and n
        if seq1[i - 1] == alphabet[m]:
            break
    for n in range(len(alphabet)):
        if seq2[j - 1] == alphabet[n]:
            break
    if v_table[i][j] == v_table[i - 1][j - 1] + similarity_matrix[m][n]:  # S[i] aligns with T[j]
        ways.append([i - 1, j - 1])
    if v_table[i][j] == v_table[i - 1][j] + pergap_score:  # Deletion
        ways.append([i - 1, j])
    if v_table[i][j] == v_table[i][j - 1] + pergap_score:  # Instertion
        ways.append([i, j - 1])
    if len(ways) == 1:  # If the length of ways(list) is 1, it means case1: This entry just one source
        total_ways[item] = ways[0]  # We just need to refesh the position to this source's position
    elif len(ways) > 1:  # Else there is case2 or case3, it means this entry have two or three
sources
        total_ways[item] = ways[0]  # Refesh the position to the first source's position
        for s in ways[1:]:  # And then append new sources' position into the list(total_ways)
            total_ways.append(s)
print('The number of all optimal global alignments is:', count)
```

result：

The number of all optimal global alignments is: 2

3. Given a query Q[1..m] and a database sequence S[1..n]. A hit (x, y) exists if Q[x..x+w-1] looks similar to S[y..y+w-1], where w is the word size. A hit (x,y)∈D is said to satisfy the two-hit requirement if there exist (x', y')∈D such that (1) x'-x = y'-y > w and (2) x' – x < A for some constant A. Suppose you are given a list D of hits (x_1, y_1), (x_2, y_2), …, (x_s, y_s). Assume $x_1 \le x_2 \le \ldots \le x_s$. Can you give an efficient algorithm to identify all hits satisfying the two-hit requirement? What is the running time?

Answer:

The easiest way to solve this problem is using double "for" nested loop, but the time complexity will be O(mn), that's too slow. So, we need to find a faster algorithm.

Because the x is already ordered, and the number of satisfied (x',y') is few. So we can use a very classical algorithm called dichotomizing search algorithm to quickly find the range of (x',y') that satisfied: x'-x = y'-y > w and x'- x< A.

My algorithm have three steps:

①For each n-1 (x,y) ∈D, we need to use two dichotomizing search algorithm, the first dichotomizing search algorithm we need to find the (x',y') that from this (x',y') on, the later are satisfy: x'-x > w ;

②The dichotomizing search algorithm is basing on the first step, we need to find the (x',y') that from this (x',y') on, the later aresatisfy: x'-x < A by dichotomizing search algorithm;

③ Finally, we've already find the range makes (x',y') satisfy: w < x'-x < A, so now we just need to find the (x',y') satisfy the last condition: x'-x = y'-y using a simple scan ;

After using dichotomizing search algorithm, the time complexity will be: $O((m-1)*2*\log_2(n)) = O(m\log_2(n))$, it's faster than O(mn)

Python codes(two-hit requirement.py):

```
D = [[1, 2], [3, 6], [4, 2], [4, 5], [5, 6], [5, 7], [6, 6], [6, 4], [7, 10], [8, 11], [8, 6]]
w = 2
A = 6
two_hit = {}  # Define a dictionary to save (x,y):(x',y') that satisfy: x'-x = y'-y > w and x'- x< A

for item in range(len(D) - 1):  # A loop from the first item in set D to the penultimate item in set D, item = (x,y)
    # First of all, between the range of item~length(D) we need to find the (x',y') that from this (x',y') on, the
    # later are satisfy: x'-x > w by dichotomizing search algorithm
    low = item
    high = len(D) - 1
    site = 0  # To save the index of the (x,y) after first dichotomizing search
    while low <= high:
        mid = (low + high) // 2
        if D[mid][0] - D[item][0] > w:
            high = mid - 1
            site = mid
        else:
            low = mid + 1

    if site == 0:  # If there is no such (x',y') satisfy: x'-x > w, because the x is ordered so it means there is no
        # more (x',y') satisfy: x'-x > w, then we finish the whole algorithm and report
        break

    # Secondly, between the range of site~length(D) we need to find the (x',y') that from this (x',y') on, the later are
    # satisfy: x'-x < A by dichotomizing search algorithm
    low = site
    high = len(D) - 1
    site2 = 0  # To save the index of the (x',y') after second dichotomizing search
    while low <= high:
        mid = (low + high) // 2
        if D[mid][0] - D[item][0] < A:
            low = mid + 1
            site2 = mid
        else:
            high = mid - 1

    twohit = []  # Define a list to save all of the (x',y') that satisfied all of the conditions
    for i in range(site, site2 + 1):  # Finally, we've already find the range makes (x',y') satisfy: w < x'-x < A, so
        # now we just need to find the (x',y') satisfy the last condition: x'-x = y'-y
        if D[i][0] - D[item][0] == D[i][1] - D[item][1]:
            twohit.append(D[i])
    if twohit:  # Append the result from this turn into dictionary two_hit
```

```
    two_hit[str(D[item])] = twohit

for m, n in two_hit.items():  # Show all of the result pairs
  print(m, ':', n)
```

Result：

[1, 2] : [[4, 5], [5, 6]]
[3, 6] : [[7, 10], [8, 11]]
[4, 2] : [[8, 6]]

4. Consider three sequences S_1=CGTAGTA, S_2=ACGACGTA and S_3 = ACGTCGTA. Can you compute the multiple sequence alignment of S_1, S_2 and S_3 using progressive POA? Please detail the steps. (Assume global alignment. Score function: match=1, mismatch= -1, gap=-2. Using linear gap penalty.)

Answer:

First we need to do pairwise alignment:

S1=-cgtagta　　S1=-cgtagta　　S2= acgacgta
S2=acgacgta　　S3= acgtcgta　　S3= acgtcgta

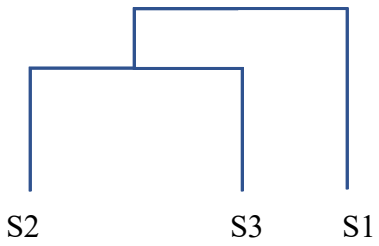
The score of each alignment:

[S1,S2]= 5*1-2*(-1)-1*(-2)=1;
[S1,S3]= 6*1-1*(-1)-1*(-2)=3;
[S2,S3]= 7*1-1*(-1)-0*(-2)=6;

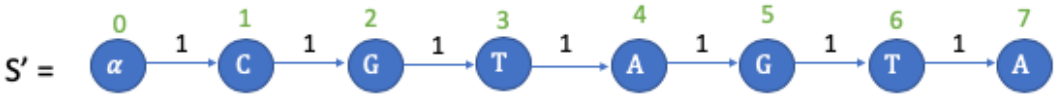
similarity matrix:

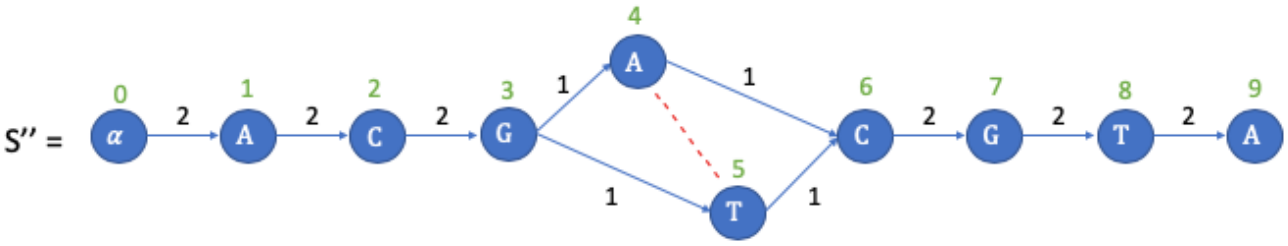
	S1	S2	S3
S1		1	3
S2			6
S3			

The corresponding neighbor-joining tree is:



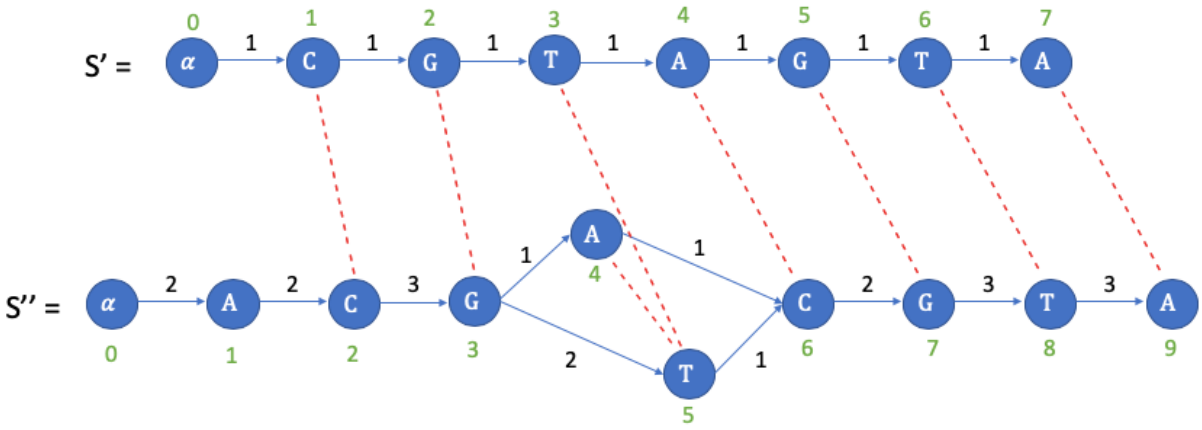
So we just need to align S2 with S3 and S1's graph:



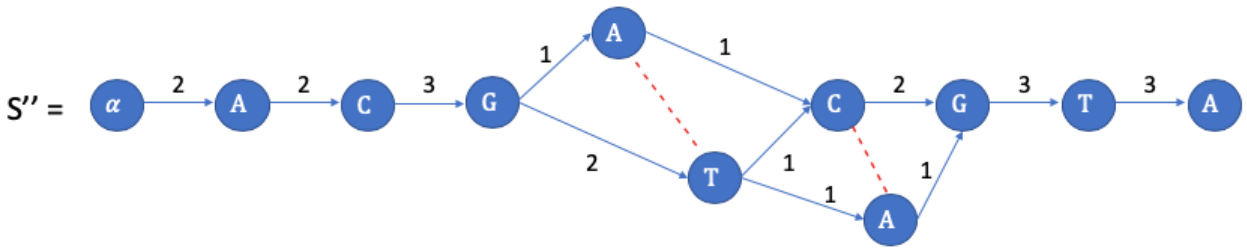


	-	A	C	G	A	T	C	G	T	A
-	0	-2	-4	-6	-8	-8	-10	-12	-14	-16
C	-2	-1	-1	-3	-5	-5	-7	-9	-11	-13
G	-4	-3	-2	0	-2	-2	-4	-6	-8	-10
T	-6	-5	-4	-2	-1	1	-1	-3	-5	-7
A	-8	-5	-6	-4	-1	-1	0	-2	-4	-4
G	-10	-7	-6	-5	-3	-3	-2	1	-1	-3
T	-12	-9	-8	-7	-5	-4	-4	-1	2	0
A	-14	-11	-10	-9	-6	-6	-5	-3	0	3

Then we incorporate S into G:



From the alignment, we merge S and G:



The progressive POA alignment is as follows:

```
S1 = -cgtagta
S2 = acgacgta
S3 = acgtcgta
```