ADSA Assignment-3 Report

(CS6013)

Submitted by
Shivangi Parashar

AI20MTECH14012

Course Instructor

Dr. Rakesh Venkat

Contents

System Specification

OS: Ubuntu 20.04 LTS
Compiler version : g++(Ubuntu 9.3.0-10ubuntu2)9.3.0

Problem Description

Sequence alignment is application of Dynamic Programming to align DNA sequences. In the global DNA sequence alignment problem, we are given as input two DNA sequences S1, S2, which are strings using the four characters A, C, T, G. The strings are of length m and n, where possibly m ≠n. Goal is to compute the best possible (minimum) alignment cost between them.

Alignment cost

.

for 2 characters scoring function is given by S(x,y) given below:

$$s(x,y) = \begin{cases} 0 & if\ x = y \\ \delta & if\ x = \_\ or\ y = \_ \\ \alpha & if\ x \neq y \end{cases}$$

Gap Penaty	δ	1
Mismatch Penalty	α	2

Example

Consider the target and query sequences as AACAGTTACC and TAAGGTCA. Then the optimal alignment and alignment cost are,

AACAGTTACC

TA_AGGT_CA

Score would be given by:

Score=2+0+1+0+0+2+0+1+0+2=8

Input Specifications

- User should enter valid input.
- Standard Input used for taking
- 3rd line: X sequence
- 4th line: Y sequence

Programming Paradigm used

- Dynamic Programming is used as a programming paradigm.
- Original Problem is one of the sub problems.
- Each sub problem is easily solved from smaller sub problems

Programming Language used

- C++ as a programming Language

Utility Library

Pair container is used. The pair container is a simple container defined in <utility>header consisting of two data elements or objects.
The first element is referenced as first and the second element as second. Order is fixed.
To access the elements, we will use variable name followed by dot operator followed by the keyword first or second.
For accessing the nested pairs we have to access just like JSON object or like a structure.

Time complexity

.

- The matrix is calculated in O (MN) .

Space Complexity

- Space requirement is O (M+N).

Concept

Goal

Given 2 **strings $(x_1,x_2...................x_n)$ and $(y_1,y_2...................y_n)$** we need to find the minimum alignment cost.

An alignment P is a set of ordered pairs (x_i-y_j)such that each character appears in atmost 1 pair and no crossings

Now we need to define optimal substructure of our problem

Opt(i,j)=minimum cost of aligning prefix strings **$(x_1,x_2...................x_m)$ and $(y_1,y_2...................y_n)$** .

$$\textbf{opt}(i, j) = \min \begin{cases} \alpha_{x_{i+1}y_{j+1}} + g(i + 1, j + 1) \\ \delta + g(i, j + 1), \\ \delta + g(i + 1, j)]. \end{cases}$$

Dynamic Program paradigm is used to solve the above problem.

For index1 < n and index2 < m we have

$$g(i, j) = \min[\alpha_{x_{i+1}y_{j+1}} + g(i + 1, j + 1), \delta + g(i, j + 1), \delta + g(i + 1, j)]. \quad \text{eq.1}$$

where δ is gap penalty

and α is miss penalty

Cases Possible

We want best alignment between 2 sequences consider p ,q then we will have following cases

Case1:

P_i aligns to q_j

P_1.............P_{i-1} P_i

q_1.............$q_{j-1}q_j$

Align p[1…i] with q[1…j]

Hence we will continue with i+1 and j+1

Case2:

Q_j aligns to a gap

P_1.............P_i _

q_1.............$q_{j-1}q_j$

Align a gap in P to q[j]

Hence we will add '_' in string 1 and continue with i+1 and j+1

Case3:

P_i aligns to a gap

P_1.............P_{i-1} P_i

q_1.............q_j _

Align a gap in q to P[i]

Hence we will add '_' in string 2 and continue with i+1 and j+1

Algorithm used

The alignment score is calculated using the concept of dynamic programming paradigm where optimal cost of every sequence is calculated and stored.
In this we are using nested pair utility library of c++ to store pair.
Steps of algorithm are depicted well with the help of below diagram

Example

Consider target and query sequences TAAGGTCA and AACAGTTACC with δ, α as 2 and 1 .

Corresponding cost matrix would be

X/Y	T	A	A	G	G	T	C	A	-
A	7	8	10	12	13	15	16	18	20
A	6	6	8	10	11	13	14	16	18
C	6	5	6	8	9	11	12	14	16
A	7	5	4	6	7	9	11	12	14
G	9	7	5	4	5	7	9	10	12
T	8	8	6	4	4	5	7	8	10
T	9	8	7	5	3	3	5	6	8
A	11	9	7	6	4	2	3	4	6
C	13	11	9	7	5	3	1	1	4
C	14	12	10	8	6	4	2	1	3
_	16	14	12	10	8	6	4	2	0

- Here we will use memoization concept of dynamic programming to speed up the process by storing the results of expensive calls and returning the cached results when we need same input again.
- used pair utility of c++ ,we will use nested pair dp[i][j] to store the value of sub problems being evaluated.
- While calculation we need to check whether dp[i][j] contains value.Then,it is a already evaluated dp state we will just return value .
- We have maintained extra array to store all the precomputed results.

Structure of our nested pair dp looks something like
Pictorial view of our algorithm

dp[i][j]={{'?','?'},-1}when we have unevaluated dp state Initially

Representation of pair dp

dp.ff	**dp.ss**
{a[index1][index2]}	**{value}**

- Now,we will call the function find_ans to compute the results
- Here if index1==n & index2=m then we reached end of string and our pair would have

dp.ff	**dp.ss**
{'\0','\0'}	**0**

- If we have,already evaluated dp state then we should simply return dp[index1][index2]
- If we reach end of string 1 then we insert gap in string1 and continue with index2+1,index1
- If we reach end of string 2 then we insert gap in string 2 and continue with index+1,index2 .
- We have maintained one more nested pair to check for lexicographically the smallest query.
- **Initially we set our nested ans pair as ans={{'?','?'},INF}** having infinity as the answer as we haven't done anything initially then in the successive calls we will update its value.

ans.ff	**ans.ss**
{{'?','?'}	**INF**

Pseudo Code

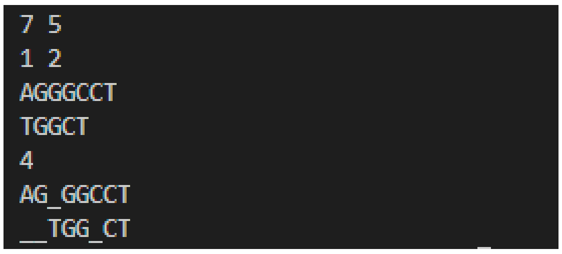
```
For i=0 to n:
 dp[i]=new pair<pair<char,char>,l1>[m+1]
 for(j=o to m)
 dp[i][j]={{'-','-'},-1}};      //unevaluated state

 find_ans(l1x,l1y,index1,index2,n,m,string a,string b
pair<pair<char,char>,l1>**dp)
 if(index1==n and  index2==m)
         return{{'\0', '\0'},0} ;     //end of both strings
if dp!='?')
          return dp[index1][index2]
if index2=m
dp[index1][index2]={{a[index1],'_'},x+find_ans(x,y,index+1,index2,n,
m,a,b,dp).ss};
return dp[index1][index2]
if index1 =n
dp[index1][index2]={{'_',b[index2]},
x+find_ans(x,y,index1,index2+1,n,m,a,b,dp).ss};
return dp[index1][index2]

//now 3 cases arise
//case3
//We are doing this to find lexigographically smallest query.
Add '-' in  string 2 and continue with index+1 and index2
          if(ans.ss>value3)
                   ans->{{a[index1],'_'},value3}
//case1
Continue with index1+1 and index2+1
           if(ans.ss>value1)
                      ans->{{a[index1],b[index2]},value1}}

//case2
Add'_' in string1 and  continue with index+1 and index2+1
           if(ans.ss>value2)
                     ans->{{'_',b[index2]},value2}}
dp[index1][index2]=ans
return dp[index1][index2]
```

Time Complexity

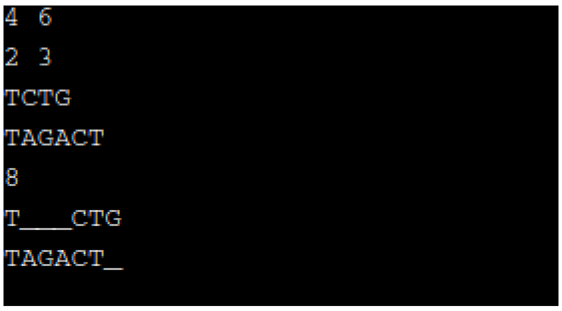
Since there are m*n number of elements .Hence,time complexity =0(mn)
Amortised cost=O(1)
Expected cost=O(mn)

Some tests and results

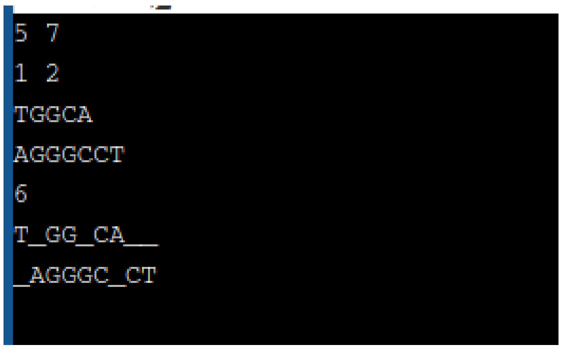
Test results snippets



```
7 5
1 2
AGGGCCT
TGGCT
4
AG_GGCCT
__TGG_CT
```



```
4  6
2  3
TCTG
TAGACT
8
T___CTG
TAGACT_
```



```
5  7
1  2
TGGCA
AGGGCCT
6
T_GG_CA__
_AGGGC_CT
```