DESIGN AND ANALYSIS OF ALGORITHMS (22AIE212)

SEQUENCE ALIGNMENT

In partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

CSE(AI)

Centre for Computational Engineering and Networking

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112 (INDIA)

JUNE – 2024

A THESIS

Submitted by

GROUP 3

PRANAV G (CB.EN.U4AIE22016)

GANESH SUNDHAR S (CB.EN.U4AIE22017)

SIRI REDDY (CB.EN.U4AIE22019)

HARI KRISHNAN N (CB.EN.U4AIE22020)

TABLE OF CONTENTS

1 PROBLEM STATEMENT

Sequence alignment is a critical problem in various fields such as text processing, data comparison, and pattern recognition. It involves arranging two sequences to identify regions of similarity or dissimilarity. This can be used to find common patterns, measure similarity, or even detect anomalies. The objective is to align the sequences in such a way that the overall alignment score is optimized, which typically involves maximizing matches and minimizing mismatches and gaps.

Design and analyse an efficient algorithm to solve the sequence alignment problem.

For this purpose, we would be using two different classes of algorithms where the second method has a little bit of advantage over the other. Those classes of algorithms are:

1. Dynamic Programming
2. Divide and Conquer

We would be discussing the problem statement in detail separately for both Dynamic Programming algorithm and Divide and Conquer algorithm in their respective sections. The main difference between these two lies in the space complexity of those algorithms.

2 NEEDLEMAN WINSCH ALGORITHM

This is the algorithm that uses dynamic programming to solve the sequence alignment problem. The alignment created is a global alignment meaning that the alignment is calculated for the entire length of both the sequences. We have a modification of this algorithm to solve for the local sequence alignment which finds out pairs of local sequences that aligns well. That algorithm is called Smith Waterman algorithm and it also uses dynamic programming to find the locally aligned sequences. The Needleman Winsch algorithm was developed by Saul Needleman and Christian Wunsch in 1970.

2.1 THE PROBLEM

We type a lot in this digital word. We send messages to others in messengers like whatsapp, etc. Wherever we type, our keyboard will come up with suggestions for the words we want to type or correct the misspelled words. Our keyboard does this by searching for the word which is most similar to the word that we currently typed. But the main problem here is how do we define similarity.

We can do this by lining up the two words letter by letter. Take the example of correct and corract. We can either line them up with a mismatch at the 5^{th} location or add a gap in 5^{th} and 6^{th} locations as follows :

correct corract	corr-ect corra-ct	corre-ct corr-act

The hyphen(-) represents a gap in the alignment. We can represent similarity like this with costs for every mismatch and gap that occurs in the alignment. We can then approach the problem as a problem of minimization of the cost that occurs due to gaps and mismatches. This is the standard definition of the sequence alignment problem.

Now, to the more formal definition, suppose we are given two strings *X* and *Y* where *X* consists of the sequence of symbols $x_1x_2x_3....x_m$ and *Y* consists of the symbols $y_1y_2y_3....y_n$. Consider the two sets {1,2,…,m} and {1,2,…,n} representing the different positions in the strings *X* and *Y*. Consider a matching of these sets, where a matching is the set of ordered pairs with the property that each times occurs in at most one pair. We say that a matching M of these sequences is an alignment if there are no crossing pairs, if (i,j), (i',j') $\in$ M and $i < i'$ the $j < j'$.

Our definition of similarity will be based on finding the optimal alignment between X and Y, according to the following criteria. Suppose M is a given alignment between X and Y:

- First, there is a parameter $\delta > 0$ that defines a gap penalty. For each position of X or Y that is not matched in M—it is a gap—we incur a cost of δ.
- Second, for each pair of letters p, q in our alphabet, there is a mismatch cost of α_{pq} for lining up p with q. Thus, for each (i, j) $\in$ M, we pay the appropriate mismatch cost α_{xiyj} for lining up x_i with y_j. One generally assumes that $\alpha_{pp} = 0$ for each letter p—there is no mismatch cost to line up a letter with another copy of itself—although this will not be necessary in anything that follows.

- The cost of M is the sum of its gap and mismatch costs, and we seek an alignment of minimum cost.

The quantities δ and $\{\alpha_{pq}\}$ are external parameters that must be plugged into software for sequence alignment

2.2 THE ALGORITHM – WHY IT WORKS

2.2.1 LAST CHARACTERS MISMATCH

Let M be any alignment of *X* and *Y*. If (m, n) ∉ M, then either the mth position of *X* or the nth position of *Y* is not matched in M. ……...(1)

Suppose by way of contradiction that (m, n) ∉ M, and there are numbers i < m and j < n so that (m, j) ∈ M and (i, n) ∈ M. But this contradicts our definition of alignment: we have (i, n), (m, j) ∈ M with i < m, but n > i so the pairs (i, n) and (m, j) cross. So, those pairs cannot form an alignment and this proves the above statement.

2.2.2 FORMULATION OF RECURRENCE

The above statement can be rewritten as follows:

In an optimal alignment M, at least one of the following is true:

(i) (m, n) ∈ M; or

(ii) the mth position of X is not matched; or

(iii) the nth position of Y is not matched

Let OPT(i,j) denote the minimum cost of alignment between $x_1x_2x_3\ldots x_i$ and $y_1y_2y_3\ldots y_j$. Then, in case

(i) holds, we pay α_{xmyn} and then align $x_1x_2 \ldots x_{m-1}$ with $y_1y_2 \ldots y_{n-1}$ and we get

$$OPT(m,n) = \alpha_{xmyn} + OPT(m-1,n-1)$$

(ii) holds, we pay a gap cost of δ since the mth position of X is not matched and then align $x_1x_2 \ldots x_{m-1}$ with $y_1y_2 \ldots y_n$ and we get

$$OPT(m,n) = \delta + OPT(m-1,n)$$

(iii) holds, we similarly get the recursion as follows

$$OPT(m,n) = \delta + OPT(m,n-1)$$

2.2.3 THE RECURRENCE RELATION

The minimum alignment costs satisfy the following recurrence for $i \geq 1$ and $j \geq 1$:

$OPT(i, j) = \min[\alpha_{xiyj} + OPT(i-1, j-1), \delta + OPT(i-1, j), \delta + OPT(i, j-1)]$.

Moreover, (i, j) is in an optimal alignment M for this subproblem if and only if the minimum is achieved by the first of these values.

2.3 PSEUDOCODE

```
Alignment(X, Y)
  Array A[0...m, 0...n]
  Initialize A[i, 0] = iδ for each i
  Initialize A[0, j] = jδ for each j
  For j = 1, ..., n
     For i = 1, ..., m
        Use the recurrence (6.16) to compute A[i, j]
     Endfor
  Endfor
  Return A[m, n]
```

2.4 C++ SOURCE CODE

```
// The headers used
#include<iostream>
#include<cmath>
#include<string>
#include<algorithm>

using namespace std;

// Use a structure to store two strings and return the output as a new
datatype as we can't return two values

struct Sequences{
    string str1;
    string str2;
};


// Function to display the 2D array contatining the costs in order to check if
the algorithm works
```

```
void display_cost(double *A, int m, int n){
    for(int i=m;i>=0;i--){
        for(int j=0;j<=n;j++){
            cout << A[(i)*(n+1) + j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}


// The algorithm to calculate the optimal sequence alignment using dynamic
programming

Sequences optimal_sequence_alignment(Sequences str, double EmptyCost, double
MatchCost){
    string str1 = str.str1;   // Initialise variables to store the input
obtained as a Sequences data type
    string str2 = str.str2;
    int m = str1.length(), n = str2.length(), i, j;

    // If the length of the first sequence is 0, align it with the second
sequence by adding n gaps and return the output
    if(m==0){
        Sequences ans;
        string ans1 = "";
        for(i=0;i<n;i++){
            ans1 += " ";
        }
        ans = {ans1, str2};
        return ans;
    }
    // If the length of the second sequence is 0, align it with the second
sequence by adding m gaps and return the output
    if(n==0){
        Sequences ans;
        string ans2 = "";
        for(i=0;i<m;i++){
            ans2 += " ";
        }
        ans = {str1, ans2};
        return ans;
    }

    // Initialise the 2D array to store the cost values
    double A[m+1][n+1], CurrCost = 0;

    // Initialise the first row and first column of the 2D array created with
gap costs multipled by the number of gaps encountered so far
```

```
    for(int i=0; i<=m; i++){
        A[i][0] = i*EmptyCost;
    }
    for(int j=0; j<=n; j++){
        A[0][j] = j*EmptyCost;
    }

    // Iterate through the array row wise first and then column wise. Then use
the recurrence relation at each iteration to build the 2D dynamic programming
array
    for(i=1;i<=m;i++){
        for(j=1;j<=n;j++){
            CurrCost = 0;   // Set the CurrCost to 0. This variable holds the
match or mismatch cost
            if(str1[i-1] != str2[j-1]){
                CurrCost = MatchCost;   // If a mismatch is encountered,
change the CurrCost to the mismatch cost, else it will be 0
            }
            A[i][j] = min(CurrCost + A[i-1][j-1], min(EmptyCost + A[i-1][j],
EmptyCost+ A[i][j-1]));   // The recurrence relation
        }
    }

    // Traceback the 2D array formed to obtain the optimal sequnce alignment
formed

    i=m;   // Start from the end and reach (0,0)
    j=n;
    string ans1 = "", ans2 = "";   // Variables to store the output
    double v1, v2, v3;
    while(i!=0 && j!=0){
        // Get the matchcost at each step
        CurrCost = 0;
        if(str1[i-1] != str2[j-1]){
            CurrCost = MatchCost;
        }

        /*
        Calaculate the costs of the 3 possible positions from where we
obtained our current position (i,j). This can be from a
            (i) Match/Mismatch and (i-1,j-1)
            (ii) Gap and (i-1,j)
            (iii) Gap and (i,j-1)
        */
        v1 = EmptyCost + A[i][j-1];
        v2 = EmptyCost + A[i-1][j];
        v3 = CurrCost + A[i-1][j-1];
```

```
        // Update the string variables created with either the characters from
the strings or gap and character from one string.
        // It depends on which path the algorithm took to reach the current
position calculated from the above values
        if(v1<v2 && v1<v3){
            ans1 = " " + ans1;
            ans2 = str2[--j] + ans2;
        }
        else if(v2<v3){
            ans2 = " " + ans2;
            ans1 = str1[--i] + ans1;
        }
        else{
            ans1 = str1[--i] + ans1;
            ans2 = str2[--j] + ans2;
        }
    }

    // If any one of the values reach zero i.e. (i,0) or (0,j), we add gaps
until the other value becomes zero
    while(i != 0){
        ans2 = " " + ans2;
        ans1 = str1[--i] + ans1;
    }

    while(j != 0){
        ans1 = " " + ans1;
        ans2 = str2[--j] + ans2;
    }

    // Store the two strings into the sequences datatype
    Sequences output = {ans1, ans2};

    //display_cost(A[0], m, n);
    return output;   // Return the output
}


// Main function containing sample values to check for the correctness of our
algorithm
int main(){
    int n = 9;   // Number of examples used
    double EmptyCost = 1;  // Empty cost (gap) is 1
    double MatchCost = 2;  // Cost for all mismatchs are 2 and all matchs are
0. This is in general. We can change this depending on the field in which the
algotihm is used.
    // Array of samples
    Sequences inputs[] = {{"correct", "corract"}, {"fast","fasting"}, {"cat",
"dog"}, {"dog", "dig"}, {"internet", "interest"}, {"happiness", "happening"},
```

```
{"computer", "commuter"}, {"programming", "program"}, {"transform",
"transaction"}};
    Sequences output;   // Variable to store the output

    // Iterate through each smaple and print the output for each
    for(int i=0; i<n; i++){
        output = optimal_sequence_alignment(inputs[i], EmptyCost, MatchCost);
        cout << "Case " << i+1 << " :-" << endl <<
        "Sequence 1 : '" << output.str1 << "'" << endl <<
        "Sequence 2 : '" << output.str2 << "'" << endl << endl;
    }
}
```

2.5 OUTPUT

```
PS F:\Ganesh\Amrita\Subjects\Sem 4\Design and analysis of algorithms\Project> .\a
Case 1 :-
Sequence 1 : 'correct'
Sequence 2 : 'corract'

Case 2 :-
Sequence 1 : 'fast   '
Sequence 2 : 'fasting'

Case 3 :-
Sequence 1 : 'cat'
Sequence 2 : 'dog'

Case 4 :-
Sequence 1 : 'dog'
Sequence 2 : 'dig'

Case 5 :-
Sequence 1 : 'interne t'
Sequence 2 : 'inter est'

Case 6 :-
Sequence 1 : 'happ  iness'
Sequence 2 : 'happenin  g'

Case 7 :-
Sequence 1 : 'computer'
Sequence 2 : 'commuter'

Case 8 :-
Sequence 1 : 'programming'
Sequence 2 : 'progra m   '

Case 9 :-
Sequence 1 : 'trans   form'
Sequence 2 : 'transactio n'
```

2.6 ANALYSING THE ALGORITHM

2.6.1 TIME COMPLEXITY

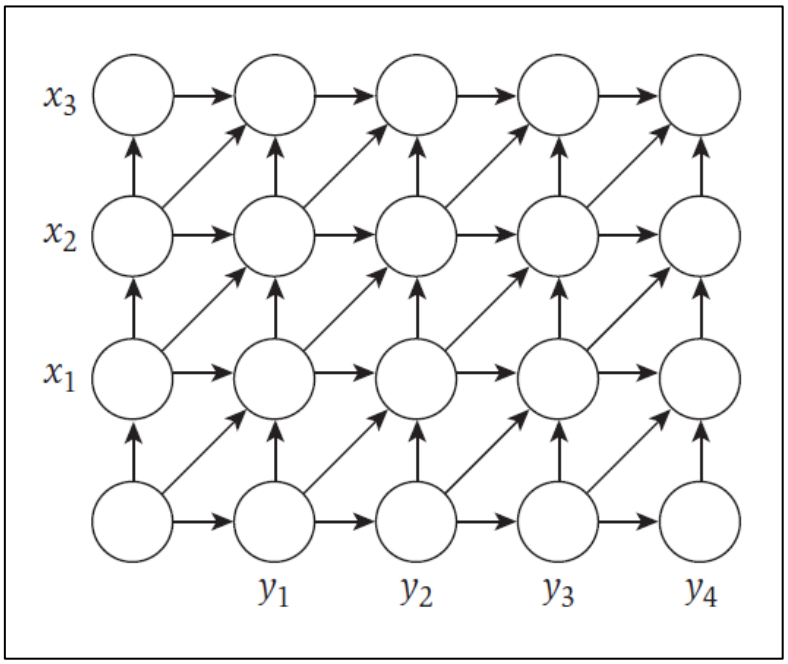
We have to figure out the basic instruction first in order to obtain the time complexity of the algorithm. As it would be obvious, the basic instruction of our algorithm would be the one that lies in the innermost for loop. The recurrence relation that we use would be the basic instruction in our sequence alignment using dynamic programming. As it is clear, the first for loop runs m times and the inner most loop runs n times for 1 iteration of the outer loop. So, the total number of times the innermost loop / recurrence relation runs is m x n which gives us a time complexity of O(mn).

2.6.2 SPACE COMPLEXITY

The major space that our algorithm uses is to store the 2D dynamic programming array that we create. So, the space complexity of our algorithm will be O((m+1)*(n+1)) which is equivalent to a space complexity of O(mn).

2.7 GRAPH BASED APPROACH

Consider a m x n 2D grid graph G with the rows labelled by symbols in the string X, the columns labelled by symbols in Y. We number rows from 0 to m and columns from 0 to n. The cost of each vertical edge and horizontal edge would be δ the gap cost. The cost of a diagonal edge from(i-1,j-1) to (i,j) would be α_{xiyj}. The objective here would be to find the minimum cost path from the node (0,0) to (m,n). The value at each node (i,j) would contain the value OPT(i,j). This is the direct effect of the recurrence relation and the graph is connected in the following manner :



3 HIRSCHBERG'S ALGORITHM

This algorithm uses dynamic programming to solve the sequence alignment problem. The alignment created is a global alignment meaning that the alignment is calculated for the entire length of both the sequences. Hirschberg's algorithm is a space-efficient divide-and-conquer approach to solving the sequence alignment problem. It is particularly effective for large sequences where the quadratic space complexity of traditional dynamic programming algorithms like Needleman-Wunsch is impractical.

3.1 THE PROBLEM

We can do nothing about the time complexity of the algorithm as we must always explore all the mn possibilities in order to obtain the optimal alignment for the given sequences. But the question is that should we be happy with the O(mn) space and is there anything we can do about the space complexity of the algorithm. Consider the application of sequence alignment in bioinformatics where nucleotide sequences are compared. The length of each sequence might be very high. Let's take the example where the lengths of each of the stringa is 1 million. Depending on the processor used, the time taken would be less of a worry than the space used. The space used would approximately be 10 GigaBytes.

Here, we use the divide and conquer approach to reduce the space complexity to a linear one while blowing up the running time by at most an additional constant factor. The main idea here is to divide the problem into many smaller versions of the original problem so that the space can be reused from one recursive call to the other leading to a linear space algorithm. This is the idea of the Hirschberg's algorithm.

3.2 THE ALGORITHM – WHY IT WORKS

3.2.1 SPACE EFFICIENT ALIGNMENT

3.2.1.1 ALGORITHM

If we only care about the value(cost) of the optimal alignment, then there is a straight forward approach to calculate it in linear space. The idea here is that in order to fill any cell in the array, the recurrence only requires values only from the previous row and previous column. So, we can collapse the array from m x n size to m x 2 size. Let A be the m x n array and B be the m x 2 array. Initially B[:,0] will hold the gap costs multiplied by their index i.e. the same values from A[:,0]. Then we calculate B[:,1] using the recurrence and then update the values of B[:,0] with the values of B[:,1]. Now, we again use the recurrence to calculate new B[:1]. At this step, B[:,1] contains the values from A[:,2]. Similarly, we can repeat this step until we get the values of A[:,n] and return the value A[m,n] as our cost for the optimal alignment.

The above algorithm uses the same O(mn) time as the basic instruction (recurrence calculation) runs for m*n times. But the space complexity as seen is O(2m) which is equivalent to O(m). But the problem here is that we can't obtain the alignment itself. We can only calculate the cost of the optimal alignment. In order to calculate the alignment itself via traceback, we need the entire 2D array. We could think of a way where we obtain the alignment during the same time the algorithm progresses. But the problem here is that one alignment may seen optimal at that point, but when the algorithm progresses, another alignment would become the optimal one

and this would cease to be the optimal solution. So, we can't find the optimal alignment as the algorithms progresses. So, we employ the divide and conquer approach to find the optimal alignment in linear space.

3.2.1.2 PSEUDOCODE FOR OPTIMAL ALIGNMENT

```
Space-Efficient-Alignment(X, Y)
  Array B[0...m, 0...1]
  Initialize B[i, 0] = iδ for each i (just as in column 0 of A)
  For j = 1, ..., n
     B[0, 1] = jδ (since this corresponds to entry A[0, j])
     For i = 1, ..., m
          B[i, 1] = min[α_{x_i y_j} + B[i − 1, 0],
                        δ + B[i − 1, 1],  δ + B[i, 0]]
     Endfor
     Move column 1 of B to column 0 to make room for next iteration:
          Update B[i, 0] = B[i, 1] for each i
  Endfor
```

3.2.2 PATH CROSSING A NODE

Let f(i,j) denote the cost of the path from (0,0) to (i,j) and let g(i,j) denote the cost of the path from (i,j) to (m,n). The statement here is that

The length of the shortest corner-to-corner path in G (graph) that passes through (i, j) is f (i, j) + g(i, j).

Let l_{ij} denote the length of the shortest corner-to-corner path in G that passes through (i, j). Clearly, any such path must get from (0, 0) to (i, j) and then from (i, j) to (m, n). Thus its length is at least f (i,j) +g(i,j), and so we have $l_{ij} \geq$f(i,j) +g(i,j). On the other hand, consider the corner-to-corner path that consists of a minimum-length path from (0, 0) to (i, j), followed by a minimum-length path from (i, j) to (m, n). This path has length f(i,j)+g(i,j), and so we have l_{ij} ≤f(i,j) +g(i,j). It follows that l_{ij} =f(i,j)+g(i,j).

3.2.3 MINIMUM COST PATH

We have to find the minimum cost path from (0,0) to (m,n). The divide and conquer approach to solve this lies in the following statement.

Let k be any number in {0,...,n}, and let q be an index that minimizes the quantity f (q,k) + g(q,k). Then there is a corner-to-corner path of minimum length that passes through the node (q, k).

Let l^* denote the cost of the shortest length path in the graph G. Lets fix a value for k in {0,1,…,n}. The shortest corner to corner path must pass through one of the nodes in the k^{th} column. Otherwise, it is not a global alignment. Let the node where it passes be (p,k). Let q be the index minimizing f(q,k) + g(q,k). So, we have

$$\ell^* = f(p,k) + g(p,k) \geq \min_q f(q,k) + g(q,k).$$

The above equation comes from the fact that min f(q,k) + g(q,k) must be lesser than all other nodes in the k^{th} column. So, we have

$$\ell^* \geq f(q,k) + g(q,k).$$

Since *l** is the minimum length of any corner-to-corner path, we have

$$\ell^* \leq f(q,k) + g(q,k).$$

From the above two equation, we can easily see that

$$\ell^* = f(q,k) + g(q,k).$$

3.2.4 DIVIDE AND CONQUER FORMULATION

We divide the graph G into two halves at the $n/2^{th}$ column and calculate the values of f(i,n/2) and g(i,n/2) for all possible i values. We can determine the minimum value of f(i,n/2) + g(i,n/2) and conclude that there is a shortest corner-to-corner path passing through the node (i,n/2). Given this, we can search for the shortest paths between (0,0) and (i,n/2) as the first subproblem problem and (i,n/2) and (m,n) as the second subproblem recursively. The crucial point here is that we apply these recursive calls sequentially and reuse the space from one call to the next.

3.3 PSEUDOCODE

```
Divide-and-Conquer-Alignment(X,Y)
  Let m be the number of symbols in X
  Let n be the number of symbols in Y
  If m ≤ 2 or n ≤ 2 then
     Compute optimal alignment using Alignment(X,Y)
  Call Space-Efficient-Alignment(X,Y[1:n/2])
  Call Backward-Space-Efficient-Alignment(X,Y[n/2+1:n])
  Let q be the index minimizing f(q,n/2)+g(q,n/2)
  Add (q,n/2) to global list P
  Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
  Divide-and-Conquer-Alignment(X[q+1:n],Y[n/2+1:n])
  Return P
```

3.4 C++ SOURCE CODE

```cpp
// The headers used
#include<iostream>
#include<cmath>
#include<string>
#include<algorithm>
#include<vector>

using namespace std;

// Use a structure to store two strings and return the output as a new
datatype as we can't return two values

struct Sequences{
    string str1;
    string str2;
};

// Initialise a structure containing two variables to store the location of
the nodes in the shortest path in the graph
struct Point{
    int x;
    int y;
};

vector<Point> global;   // The vector of size m+n that stores the points in
the shortest path


// Function to compare the points in the shortest path in order to arrange
them in the ascending order after inserting them

bool comparePoints(Point p1, Point p2){
    if (p1.x == p2.x) {
        return p1.y < p2.y;
    }
    return p1.x < p2.x;
}


// The function to calculate the cost of the optimal alignment in a space
efficient way

double space_efficient_alignment(Sequences str, double EmptyCost, double
MatchCost){
    string str1 = str.str1;   // Variables to store the input strings
    string str2 = str.str2;
    int m = str1.length(), n = str2.length(), i, j;
```

```
    // If any of the values is 0, the optimal cost would be the gap cost
multiplied by the other length
    if(m==0){
        return EmptyCost*n;
    }
    if(n==0){
        return EmptyCost*m;
    }

    double A[2][m+1], CurrCost;   // The m x 2 array to store the cost values
and find the optimal cost

    for(i=0;i<=m;i++){
        A[0][i] = EmptyCost*i;
    }

    for(i=1;i<=n;i++){
        A[1][0] = EmptyCost*i;
        for(j=1;j<=m;j++){
            CurrCost = 0;
            if(str1[j-1] != str2[i-1]){
                CurrCost = MatchCost;   // Set the mismatch cost if it is not
zero
            }
            A[1][j] = min(CurrCost + A[0][j-1], min(EmptyCost + A[0][j],
EmptyCost+ A[1][j-1]));   // The recurrence step
        }

        for(j=0; j<=m; j++){
            A[0][j] = A[1][j];   // The update step to update the values from
the first row to the zeroth row
        }
    }

    return A[1][m];   // Return the optimal cost
}


// The dynammic programming step to calculate the minimum alignment if the
length of any of the strings is less than 2

void dynamic_programming(Sequences str, double EmptyCost, double MatchCost,
int l1, int l2){
    string str1 = str.str1;    //The variables to store the input strings
    string str2 = str.str2;
    int m = str1.length(), n = str2.length(), i, j;

    // Push all the nodes into the vector when the length of any one the
strings is zero
```

```
    if(m==0){
        for(i=1;i<n;i++){
            global.push_back({l1,l2+i});
        }
        return;
    }
    if(n==0){
        for(i=1;i<m;i++){
            global.push_back({l1+i,l2});
        }
        return;
    }

    double A[m+1][n+1], CurrCost = 0;   // Array to store the costs

    for(int i=0; i<=m; i++){
        A[i][0] = i*EmptyCost;
    }
    for(int j=0; j<=n; j++){
        A[0][j] = j*EmptyCost;
    }

    for(i=1;i<=m;i++){
        for(j=1;j<=n;j++){
            CurrCost = 0;
            if(str1[i-1] != str2[j-1]){
                CurrCost = MatchCost;
            }
            A[i][j] = min(CurrCost + A[i-1][j-1], min(EmptyCost + A[i-1][j],
EmptyCost+ A[i][j-1]));   // The recurrence relation
        }
    }

    // Traceback the 2D array formed to obtain the optimal sequnce alignment
formed
    i=m;
    j=n;
    double v1, v2, v3;
    while(i!=0 && j!=0){
        CurrCost = 0;
        if(str1[i-1] != str2[j-1]){
            CurrCost = MatchCost;
        }

        v1 = EmptyCost + A[i][j-1];
        v2 = EmptyCost + A[i-1][j];
        v3 = CurrCost + A[i-1][j-1];
```

```
        if(v1<v2 && v1<v3){
            j--;
        }
        else if(v2<v3){
            i--;
        }
        else{
            i--;
            j--;
        }
        if(i!=0 || j!= 0){
            global.push_back({l1+i,l2+j});
        }
    }

    while(i != 0){
        i--;
        if(i!=0){
            global.push_back({l1+i,l2});
        }
    }

    while(j != 0){
        j--;
        if(j!=0){
            global.push_back({l1,l2+j});
        }
    }
}


// The recursive function to calculate the nodes in the shortest path using
divide and conquer

void sequence_alignment_rec(Sequences str, double EmptyCost, double MatchCost,
int l1, int l2){
    string str1 = str.str1;   // Strings to store the inputs
    string str2 = str.str2;
    int m = str1.length(), n = str2.length(), i, j, mid;
    if(m<=2 || n<=2){
        dynamic_programming(str, EmptyCost, MatchCost, l1, l2);   // Calculate
the optimal alignment using dynammic programming if the length of any of the
strings is less than 2
        return;
    }

    mid = (int)(floor((double)(n)/2)) - 1;   // The mid column index
    double cost = space_efficient_alignment({"", str2.substr(0,mid+1)},
EmptyCost, MatchCost) + space_efficient_alignment({str1, str2.substr(mid+1, n-
```

```
mid-1)}, EmptyCost, MatchCost);   // Initialise the cost when the minimum path
cost is at the node (0,n/2)
    double temp;
    int pos = -1;   // The position variable initialised to -1
    for(i=0;i<m;i++){
        // For loop to calculate the node at which the shortest path passes in
order to get the minimum cost alignment
        temp = space_efficient_alignment({str1.substr(0,i+1),
str2.substr(0,mid+1)}, EmptyCost, MatchCost) +
space_efficient_alignment({str1.substr(i+1, m-i-1), str2.substr(mid+1, n-mid-
1)}, EmptyCost, MatchCost);
        if(temp < cost){
            cost = temp;
            pos = i;
        }
    }
    // Calculate the cost for the last node in the n/2 column
    temp = space_efficient_alignment({str1, str2.substr(0,mid+1)}, EmptyCost,
MatchCost) + space_efficient_alignment({"", str2.substr(mid+1, n-mid-1)},
EmptyCost, MatchCost);
    if(temp < cost){
        cost = temp;
        pos = m;
    }

    // Push that point into the vector containing the points in the shortest
path
    global.push_back({l1+ pos + 1, l2 + mid + 1});


    // Recursive calls to calulate the results for the subproblems generated
    if(pos==-1){
        // If the node is the first node
        sequence_alignment_rec({"", str2.substr(0,mid+1)}, EmptyCost,
MatchCost, l1, l2);
        sequence_alignment_rec({str1, str2.substr(mid+1, n-mid-1)}, EmptyCost,
MatchCost, l1, l2+mid+1);
    }
    else if(pos==m){
        // If the node is in the middle
        sequence_alignment_rec({str1, str2.substr(0,mid+1)}, EmptyCost,
MatchCost, l1, l2);
        sequence_alignment_rec({"", str2.substr(mid+1, n-mid-1)}, EmptyCost,
MatchCost, l1+m, l2+mid+1);
    }
    else{
        // If the node is at the last
```

```
        sequence_alignment_rec({str1.substr(0,pos+1), str2.substr(0,mid+1)},
EmptyCost, MatchCost, l1, l2);
        sequence_alignment_rec({str1.substr(pos+1,m-pos-1), str2.substr(mid+1,
n-mid-1)}, EmptyCost, MatchCost, l1+pos+1, l2+mid+1);
    }
}


// Function to take the inputs, calculate the optimal alignment using the
rcursion and sort them

Sequences optimal_sequence_alignment(Sequences str, double EmptyCost, double
MatchCost){
    string str1 = str.str1;
    string str2 = str.str2;
    int m = str1.length(), n = str2.length(), i, j, ind1, ind2, a1=0, a2=0;
    global.push_back({0,0});
    global.push_back({m,n});

    // Use the recursive call to get the optimal alignment
    sequence_alignment_rec(str, EmptyCost, MatchCost, 0, 0);

    // Processing the nodes to get the alignment using the sorting function
    sort(global.begin(), global.end(), comparePoints);   // Builtin function
to sort
    string ans1 = "", ans2 = "";

    // Create the alignment itself from the sorted list of nodes
    for(i=1; i<global.size(); i++){
        ind1 = global[i].x - global[i-1].x;
        ind2 = global[i].y - global[i-1].y;

        if(ind1 == 1 && ind2 == 1){
            // If the difference is 1, we use the characters from each string,
it would either be a match or a mismatch
            ans1 += str1[a1++];
            ans2 += str2[a2++];
        }
        else if(ind1 == 1){
            // If there is a gap in the second string
            ans1 += str1[a1++];
            ans2 += " ";
        }
        else{
            // If there is a gap in the first string
            ans1 += " ";
            ans2 += str2[a2++];
        }
    }
```

```
    return {ans1, ans2};   // Return the output
}


// Main function containing sample values to check for the correctness of our
algorithm

int main(){
    int n = 9;   // Number of examples used
    double EmptyCost = 1;  // Empty cost (gap) is 1
    double MatchCost = 2;  // Cost for all mismatchs are 2 and all matchs are
0. This is in general. We can change this depending on the field in which the
algotihm is used.
    // Array of samples
    Sequences inputs[] = {{"correct", "corract"}, {"fast","fasting"}, {"cat",
"dog"}, {"dog", "dig"}, {"internet", "interest"}, {"happiness", "happening"},
{"computer", "commuter"}, {"programming", "program"}, {"transform",
"transaction"}};
    Sequences output;   // Variable to store the output

    // Iterate through each smaple and print the output for each
    for(int i=0; i<n; i++){
        output = optimal_sequence_alignment(inputs[i], EmptyCost, MatchCost);
        global.clear();
        cout << "Case " << i+1 << " :-" << endl <<
        "Sequence 1 : '" << output.str1 << "'" << endl <<
        "Sequence 2 : '" << output.str2 << "'" << endl << endl;
    }
}
```

3.5 OUTPUT

```
PS F:\Ganesh\Amrita\Subjects\Sem 4\Design and analysis of algorithms\Project> .\a
Case 1 :-
Sequence 1 : 'corr ect'
Sequence 2 : 'corra ct'

Case 2 :-
Sequence 1 : 'fast   '
Sequence 2 : 'fasting'

Case 3 :-
Sequence 1 : ' cat'
Sequence 2 : 'd og'

Case 4 :-
Sequence 1 : 'dog'
Sequence 2 : 'dig'

Case 5 :-
Sequence 1 : 'interne t'
Sequence 2 : 'inter est'

Case 6 :-
Sequence 1 : 'happ  iness'
Sequence 2 : 'happenin  g'

Case 7 :-
Sequence 1 : 'co mputer'
Sequence 2 : 'comm uter'

Case 8 :-
Sequence 1 : 'programming'
Sequence 2 : 'progra m   '

Case 9 :-
Sequence 1 : 'trans    form'
Sequence 2 : 'transacti o n'
```

3.6 ANALYSING THE ALGORITHM

3.6.1 TIME COMPLEXITY

This is a recursive algorithm and hence we have to calculate the time complexity of the algorithm using a recurrence relation. Let M(m,n) denote the recurrence relation. The space efficient alignment algorithm uses O(mn) time to build up the arrays in order to get the cost of the optimal alignment. Then, we have those two recursive calls. So, the recurrence is formed as follows:

M(m,n) <= cmn + M(q,n/2) + M(m-q,n/2)
Base cases :
(i) T(m,2) <= cm
(ii) T(2,n) <= cn

Case 1 : m = n:-

The first case is that m is equal to n and the split poin is exactly in the middle, i.e. at the position n/2. Then we can express the recurrence as follows

$M(n) <= cn^2 + 2M(n/2)$
$M(n) \in O(n^2)$

Case 2 : m ≠ n:-

The second case is the more general one. Here, we assume that the time complexity is O(mn) for some constant k and the assumption is that

Let M(m,n) <= kmn

Now, for the base cases m<=2 and n<=2, we see that these hold as long as k >= c/2. Now, assuming that M(m', n') <= km'n' holds for a smaller product, we have

M(m,n) <= cmn + M(q,n/2) + M(m-q,n/2)
<= cmn + $k_1$qn/2 + k_2(m-q)n/2
<= cmn + $k_1$qn/2 + $k_2$mn/2 - $k_2$qn/2

Now, assume a constant k = max{k_1,k_2}. So, the equation can be rewritten and solved in the following manner

$$M(m,n) <= cmn + kqn/2 + kmn/2 - kqn/2$$
$$<= cmn + kmn/2$$
$$<= (c+k/2)\, mn$$
$$<= k_3 mn$$
$$\text{So, } M(m,n) \in O(mn)$$

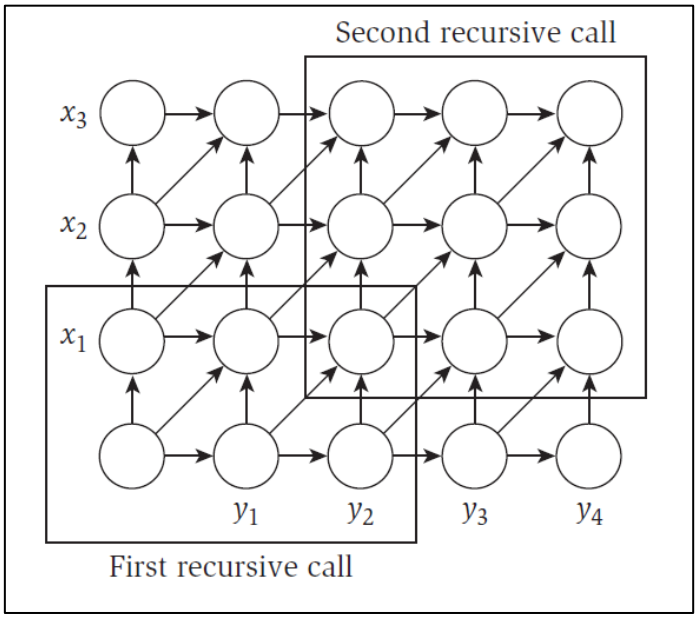
From this, we can see that the time complexity of the sequence alignment using divide and conquer is the same as that using the dynamic programming which is O(mn).

3.6.2 SPACE COMPLEXITY

The space that our algorithm uses is the array used to store the nodes in the minimum cost path from (0,0) to (m,n). So, the size of that array would be our space complexity. The size of the array is m+n. So, the space complexity of the algorithm would be O(m+n). We can see that the space complexity reduces from O(mn) to O(m+n). The algorithm runs in a linear space and this is the primary motive of the divide and conquer algorithm for sequence alignment.

3.7 GRAPH REPRESENTATION

The graph representation of the node selected and the subproblems generated in an example case of strings with length 3 and 4 with the optimal split at (2,3) is as follows:



4 CONCLUSION

The study and implementation of sequence alignment algorithms, such as Needleman-Wunsch and Hirschberg's algorithms, highlight the crucial balance between time and space complexity in solving computational problems. Needleman-Wunsch's dynamic programming approach offers a robust solution with manageable time complexity but at the cost of quadratic space usage. This becomes impractical for very large sequences, as demonstrated in bioinformatics applications.

Hirschberg's algorithm addresses this challenge by utilizing a divide-and-conquer strategy to significantly reduce space complexity while maintaining acceptable time complexity. This makes Hirschberg's algorithm particularly valuable for applications involving large datasets where memory resources are a limiting factor.

Through detailed problem formulation, algorithm design, pseudocode development, and complexity analysis, this report underscores the importance of selecting appropriate algorithms based on the specific requirements and constraints of the problem at hand. The trade-offs between time and space complexity must be carefully considered to achieve optimal performance in sequence alignment tasks, ensuring that the algorithms can handle real-world data efficiently.

In conclusion, while Needleman-Wunsch remains a foundational algorithm for sequence alignment due to its simplicity and effectiveness, Hirschberg's algorithm provides a compelling alternative for scenarios where space efficiency is paramount. Future work could explore further optimizations and hybrid approaches to enhance the performance of sequence alignment algorithms, ensuring their applicability across a broader range of fields beyond bioinformatics, such as text processing, data comparison, and pattern recognition.