Kyle Hippe 300518740

COMP 361 Design and Analysis of Algorithms

Zohar Levi

19 August 2019

Assignment 2

1. Optimal Sub-Structure

The global sequence alignment problem is similar to the longest common sub-sequence problem and the proof for optimal sub structure is similar.

Let X and Y be sequences of DNA base pairs. For any indexes, i , j, from 0 to length of X and Y respectively, we can determine if the sequences at that index are in optimal alignment based off of three cases

1. X[i] and Y[j] exist and are equal

   This represents a 'match' and an alignment cost of +1

2. X[i] and Y[j] exist but are not equal

   This represents a 'mismatch' and an alignment cost of -1

3. X[i] does not exist OR Y[j] does not exist (exclusive or)

   This represents a 'gap' and an alignment cost of -2

If we begin at the first index of each sequence and continue iterating through the sequences saving the previous scores, we will eventually end up with a solution composed of optimal solutions to sub-sets of each sequence compared to each other.

Proof:

Let A be an optimal alignment for two sequences X, Y.

Let $a \subseteq A$ where a is an optimal sub alignment for the sequences X and Y and $a_{0..k} + a_{k..n} = A$ for some value k

Proof by Contradiction:

Let $a_{0..k}'$ be some subset of A such that it costs less than a. This would result in a final solution $a'_{0..k} + a_{k..n} = A'$ where A' is now a more optimal solution to the alignment of sequences X and Y than A was. This is a contradiction since A (not A′) is the most optimal alignment of the sequences and no other solution can have a more optimal alignment cost.

2. Greedy Algorithm

A greedy algorithm for the global sequence alignment would provide the least cost scoring of each comparison with no 'forward thought' or backtracking to truly optimally align a sequence.

Given the scoring sequence above, a greedy algorithm would inspect, for each index i in both sequences and if the elements at the specific index match, insert each element into the final sequence and update the score with the match cost. If the elements do not match, each base would be appended to the sequence because although they don't match, a mismatch score is still more optimal than a space score. And finally, if one of the sequences does not exist (size differences between the two sequences) a gap will be inserted in the respective final sequence and the element in the sequence that does exist will be appended to the appropriate final sequence and the score will be updated to reflect a space penalty. The pseudo-code is provided below:

```
input: two sequences x, y with lengths n, m respectively
output: aligned sequences x` and y` both of length z (the larger of n, m)
along with a score

let score = 0

for i = 0 to max(n,m) //for loop that caters to longest sequence
    if x[i] & y[i] exist and are equal
        append both to x` and y`
        update score with match cost (+1)

    if x[i] and y[i] exist but are not equal
        append both to x` and y`
        update score with mismatch cost (-1)

    if x[i] exists and y[i] does not
        append x[i] to x` and gap to y`
        update score with gap cost (-2)

    if x[i] does not exist and y[i] does exist
        append gap to x` and y[i] to y`
        update score with gap cost (-2)
```

This algorithm, however, is not always optimal and is prone to failure. An example sequence would be:

```
AGTACGG

GAGTACGGA
```

The greedy algorithm produces an output of:

```
AGTACGG--

GAGTACGGA

-------** (-11)
```

While the optimal alignment is actually:

```
-AGTACGG-
GAGTACGGA
*+++++++*  (+3)
```

This proves that a greedy algorithm is not the right choice in this instance, but a dynamic solution is.

3. Dynamic Algorithm

To solve this dynamically, we need a method of evaluating all, or as many of as possible, sub-solutions/partial alignments to the total alignment of two sequences.

As input, we will receive two sequences X, Y with lengths n and m respectively.

First, we will construct and initialize a two-dimensional array with n rows and m columns. The first row and column will be initialized as an accumulation of gaps since comparing a non-zero length sequence to a zero-length sequence will always result in n gap penalties, where n represents the length of the non-zero length sequence.

From here, there will be two for loops that visit each cell of the table and determine from which area the maximum score can be achieved. The three options are from the column to the left, the row above, or from the diagonal to the left and above the current cell. The maximum can be determined by adding the gap penalty to the scores from the left cell or above cell and by adding a match or mismatch penalty to the diagonal cell. The match or mismatch is determined by if the current indexes i, j correspond to matched, or unmatched base pairs in the original sequences X and Y. Once the maximum has been determined, place the score in the cell and progress through the iterations. This respects the optimal substructure as any point in the matrix will represent an optimal solution to the sub-sequences of X [0…i] and Y[0…j].

The pseudo code is given below:

```
let match = 1
let mismatch = -1
let gap = -2
alignment(X, Y,):
 A[length of X][length of Y]
 initialize array so that column 0 is an accumulation of gaps
 initialize array so that row 0 is an accumulation of gaps

 for i ← 1 to length of X:
    for j ← 1 for length of Y:
        A[i][j] = max of{
            A[i-1][j-1] + match or mismatch
           //represents the diagonal, score depends on if the elements match
            A[i-1][j] + gap
            //represents a gap in X, optimal score from the left cell
            A[i][j-1] + gap
            //represents a gap in Y, optimal score from above cell
        }
```

Next, we need to backtrack to make our final alignment of the sequences. In order to backtrack, we will start at the last row and column of the matrix, since this is the maximal alignment score, and using the values of the cells to the left, above, and diagonal, determine where the current value could have come using the scoring schema provided above. Using this we will append the correct character to each output sequence and update the current index to the cell the current value originated from. This will continue until the current indexes reach (0,0), the origin of the scoring matrix. If a cell could have originated from the two different cells, arbitrarily choose one as both branches represent an optimal alignment. This will generate an aligned set of sequences.

4. Analysis

The asymptotic complexity of this algorithm is fairly simple to determine. Since the cost for calculating the maximum value at an index is constant and the cost of assigning a value to an index is constant, the majority of the time complexity of this algorithm comes from the two for loops that, for each index of sequence X, run over every index of sequence Y. This means that the time complexity for establishing the scoring matrix is O(n * m) where n is the length of sequence X and m is the length of sequence Y. The worst case would be if the lengths of both sequences were equal leading to a time complexity of $O(n^2)$ for establishing the scoring matrix.

Next to backtrack through the matrix and establishing the optimal alignment for the sequences would take approximately O(n + m) since at most we can travel n elements in the horizontal direction and m elements in the vertical direction. Since this is a linear complexity and is much less than $O(n^2)$ it will not be factored into the total time complexity.

The combination of $O(n^2 + n + m)$ leads to a total time complexity of $O(n^2)$ in the worst case, and O(n*m) in the case that the two sequences are not the same length.

5. Implementation

Input: For the input, I have a sequence X and a sequence Y. Since I implemented the algorithm in Python, these are lists (linked lists)

```
        Input = {
                X = []
                Y = []
        }
```

Output: The output of the program is two lists that represent the X′ and Y′ sequences (properly aligned) as well as a list of the symbols (‘+’plus, ’-’ minus, ’_’ space) and an integer value representing the score, which is taken from the last row and column of the scoring matrix.

```
        Output = {
                X′ = []
                Y′ = []
                Symbols = []
                Score = int
}
```

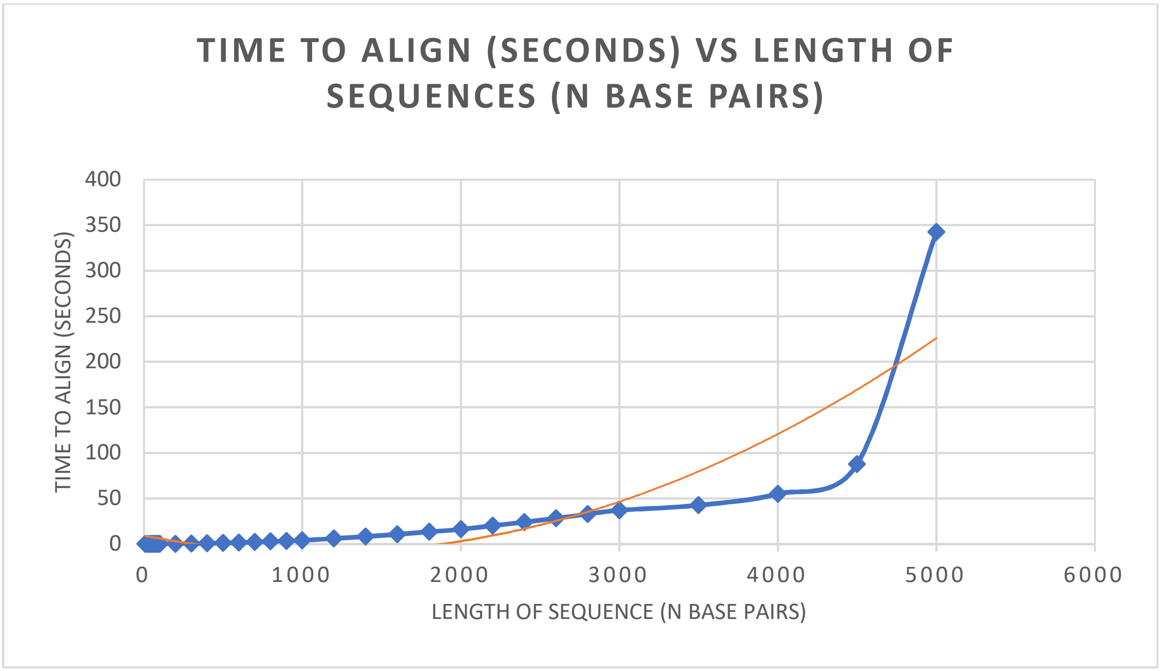
Intermediate results: While calculating the final alignment, I use a 2-dimensional array to save the scoring matrix.

```
        Intermediate = {
                scoringMatrix = [length of sequence X][length of sequence Y]

                each cell in the matrix is a 'Cell' class, which contains the
value at the specific index as well as a pointer to the cell the value was
propagated from
}
```

Differences: The only major differences between my implementation of the algorithm and the pseudo code previously described is the addition of a parent attribute to each cell in the scoring matrix which allows each cell to save its value as well as the direction of where the value was propagated from. This allowed for a simpler backtracking implementation and does not change the overall time complexity.

Graph of Time to Align vs N base Pairs in each sequence:



The blue dots represent data points of how long N base pair length sequences took to align. The orange line represents an n^2 regression line.

Table of results:

N base pairs	Time to align	N base pairs	Time to align	N base pairs	Time to align
10	.000326	300	.344391	1800	13.42341
20	0.001219	400	0.646	2000	16.11766
30	0.002545	500	1.052254	2200	20.09913
40	0.004872	600	1.501782	2400	23.92794

50	0.007339	700	2.046275	2600	28.11926
60	0.010547	800	2.676838	2800	32.89158
70	0.014121	900	3.374184	3000	37.00061
80	0.019459	1000	4.077233	3500	42.60179
90	0.024114	1200	5.939315	4000	55.03334
100	0.028911	1400	8.103638	4500	87.60493
200	0.134836	1600	10.62991	5000	342.587

This table represents the data points for the graph above. (sorry it is on two pages)

Analysis:

First and foremost, I would have liked to test more data, however, datasets above 5000 base pairs took too long to execute and were killed by the zsh terminal on the school lab computers. All tests were run on the VUW Computer Science department lab computers equipped with i7-6700 processors with 8gb of ram (this will become important later). As we can see by the graph provided, the overall time to execute an alignment as the data sizes increase roughly resembles an n^2 function, though not completely. The discrepancies in data sizes less than 5000 base pairs (where actual time for completion was less than n^2) could be due to interpreter and OS level optimizations for the implementation or being able to store data sets in faster cache memory rather than RAM or hard storage. As the data size increased, the overall RAM usage of the program increased rapidly, and for data sets above ~3500 base pairs all RAM and swap space was used, which means that portions of the scoring matrix had to have been stored in hard storage, which given the extraordinarily slow access times as compared to RAM or cache, which would explain the greater than n^2 execution times.

6. Edit Distance

6.1 Optimal Sub-Structure

The edit distance problem is a variant of the global sequence alignment problem. Instead of calculating alignment costs we are instead calculating the number of mutations required to transform a string, X, into a target string, T.

The new operations are as follows, assume i, j, k are indexes in X, T and an output string Z respectively:

1. Copy – copies the character at X[i] to Z[k]
2. Replace - replaces the character at X[i] with a different character at Z[k]
3. Delete – advances i without does anything to Z
4. Insert – does not advance i,j inserts a character into Z at k and increments the pointer
5. Twiddle – copies the characters at x[i] and x[i+1] into Z in reverse order
6. Kill – advances the i pointer to the end of the X string, terminates the evaluation

We can assume that the copy, replace, and delete operations cost less than the insert and twiddle operations, which in tern cost less than the kill operation. This algorithm has a goal of minimizing the cost.

We can use the same proof by contradiction given in the original statement but change it slightly. If we let A be an optimal transformation for sequence X into target sequence T

Let a ⊆ A where a is an optimal sub transformation for the sequences X into T and $a_{0..k} + a_{k..n} = A$ for some value k

Proof by Contradiction:

Let $a_{0..k}'$ be some subset of A such that it costs less than a. This would result in a final solution $a'_{0..k} + a_{k..n} = A'$ where A′ is now a more optimal solution to the transformation of sequences X into T than A was. This is a contradiction since A (not A′) is the most optimal transformation of X and no other solution can have a more optimal transformation cost.

6.2 Greedy Algorithm

A greedy algorithm for the edit-distance problem would provide the least cost of each transformation with no 'forward thought' or backtracking to truly optimally align a sequence. Given a scoring schema for all the operations (copy, delete, replace, insert, twiddle, and kill_ a greedy algorithm would inspect, for each index i in both sequences, the elements at the specific index, perform the operation with the least transformation cost and update the output string Z and transformation cost accordingly. The pseudo-code is provided below:

```
input: string X and target string T
output: transformed string Z and score
//numbers are irrelevant here
//they just show hierarchy of operations
let copy = 2
let replace = 2
let delete = 2
let insert = 3
let twiddle = 3
let kill = 5

let i = 0
let j = 0
while i is less than length of X
    if copy/replace/delete can be performed:
        append correct value to string z[j]
        update score with cost (2)
        update indexes i, j

    else if twiddle/insert can be performed:
        append correct value/values to string z[j]
        update score with cost (3)
        update indexes i,j accordingly

    else
        //this is the kill situation
        //also always performed as last operation
        append necessary values to z[j]
        update score with cost (5)
```

While this is greedy algorithm will produce a transformed string, it will most likely not be the optimal solution. This can be shown with the strings below:

Original string: “abcdefg”

Target string: “acbedgf”

For reference, c stands for copy, r for replace, d for delete, i for insert, t for twiddle, and k for kill. And each score is taken from the schema above.

The greedy transformation will produce:

```
a c b e d g f

c r r r r r r k (score: 18)
```

While the optimal transformation is:

```
a c b e d g f

c t   t   t   k (score: 15)
```

The greedy solution here obviously does not create the optimal solution, meaning it is not the right algorithm for the problem.

6.3 Dynamic Algorithm

The general idea behind this problem is not substantially different from the global sequence alignment problem so only a few key points need to be changed.

The first of which is the scoring matrix. Since there are more operations and their costs are more intricately related we will create a scoring schema with semi-arbitrary values as follows:

- Copy – 2
- Replace – 2
- Delete – 2
- Insert – 3
- Twiddle – 3
- Kill – 4

This means that two copy operations will cost more than one twiddle, allowing operations that cost more than copy/replace/delete to have a valid place in this algorithm

The algorithm will still use a 2-dimensional array to store the values of each transformation. It will also still have two for loops, the outer looping through each index of the input string and the inner looping through each index of the target string in a similar manner to the global alignment problem.

Establishing the matrix will remain largely unchanged except we no longer need the first row and column to hold any propagated values.

Our control structure for determining the transformation cost at any given will change however. Instead of only calculating the cost for a gap or a match/mismatch, we will have to calculate the costs of copying, replacing, inserting, or twiddling. Pseudocode shown below, this will replace the decision structure of the original algorithm:

```
//numbers are irrelevant here
//they just show hierarchy of operations

let copy = 2
let replcae = 2
let delete = 2
let insert = 3
let twiddle = 3
let kill = 5

A[i,j] = Min{
    // copy event
    if X[i] == T[j] Then cost = A[i-1][j-1] + copy

    // replace event
    else if X[i] != T[j] Then cost = A[i-1][j-1] + replace

     // delete event
    A[i-1, j] + delete

    // insert event
    A[i, j-1] + insert

    // twiddle event
    if x[i] and x[i + 1] equal T[j+1] and T[j] respectively,
     then cost =  A[i-2, j-2] + twiddle

    //kill will be determined later
```

Lastly a new function will need to be added in order to calculate the proper time for the kill function. Pseudocode is as follows:

```
For i = 1 to length of X:
    minCost = A[i, length of T] + kill
        if minCost less than A[length of X][length of T] then
            A[length of X][length of T] = minCost
```

This function will be executed after the matrix is complete, but before backtracking is executed and will insert the kill function at the optimal point in execution. It relies on a completed score matrix to be inserted optimally which is why it is not implemented in the above control structure.

These are the only changes required in the overall algorithm, the rest can remain the same.

Other than this, nothing else truly changes. The backtracking algorithm can remain unchanged as the only thing needed to be updated is the scoring schema.

6.4 Analysis

The edit-distance problem will have nearly identical run time as the program still executes two loops of the length of the input and target strings and determining the value at an index is still constant. This means that in the worst case, where target string and input string are the same length, the time complexity for the edit-distance problem will still be $O(n^2)$