Measuring DNA sequence alignment

Inas KACI

December 2020

1 Introduction

For an academic project, we had six weeks to find a solution for measuring the similarity between two DNA sequences. A sequence can be seen as a word that is obtained from the alphabet $\sum = \{A, T, G, C\}$. We had two algorithmic problems to solve:

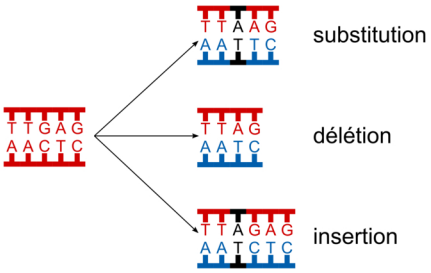
- Computing the edit distance (minimal cost of an alignment) between two words;
- Finding an optimal alignment corresponding to this distance.

We have started by implementing a naive algorithm. Then, decreased the time consumption with a dynamic approach implemented by my project partner and a divide and conquer approach I have implemented. Finally, we have compared their memory consumption performances.

2 Definitions:

2.1 Alignment:

It consists of transforming a word x to a word y by insertion (y obtained by inserting a new letter in x, encoded by a gap in $\overline{x}$), deletion (y obtained by deleting a letter from x encoded by a gap in $\overline{y}$) or substitution (replacing a letter in x by a new letter in y encoded by two different letters in $\overline{x}$ and $\overline{y}$)



Let's consider $|x|$, the length of the word x that is equal to the number of characters in it and $\overline{x_i}$, the letter at position i in $\overline{x}$.
Let's denote a gap as $\{-\}$ and $\overline{\sum} = \sum \bigcup \{\_\}$. The function π associates the word x to the word $\overline{x}$ by deleting all its gaps :

$$\pi : \overline{\sum} \to \sum$$
$$\overline{x} \mapsto x$$

$(\overline{x}, \overline{y})$ is an alignment of (x, y) if:

$$\begin{cases} \pi(\overline{x}) = x \\ \pi(\overline{y}) = y \\ |\overline{x}| = |\overline{y}| \\ \forall i \in [1, ..., |\overline{x}|], \overline{x}_i \neq \{-\} \text{ or } \overline{y}_i \neq \{-\} \end{cases} \tag{1}$$

2.1.1 Example for $x = ATTGTA$ and $y = ATCTTA$:

$\overline{x}$: A T T G T A	$\overline{x}$: A T − T G T A	$\overline{x}$: A T T G T − A	$\overline{x}$: − − − − − A T T G T A
$\overline{y}$: A T C T T A	$\overline{y}$: A T C T − T A	$\overline{y}$: A T − C T T A	$\overline{y}$: A T C T T A − − − − −

2.2 Cost of an alignment:

The cost of an alignment $(\overline{x}, \overline{y})$ is defined by $C(\overline{x}, \overline{y}) = \sum_{k=1}^{|\overline{x}|} c(\overline{x}_k, \overline{y}_k)$ where :

$$c(a,b) = \begin{cases} c_{ins}, & \text{if } a = -. \\ c_{del}, & \text{if } b = -. \\ c_{sub}(a,b), & \text{otherwise} \end{cases} \tag{2}$$

2.3 Edit distance:

The edit distance from x to y is given by:

$$d(x,y) = min\{C(\overline{x}, \overline{y}) \ |(\overline{x}, \overline{y}) \text{ is an alignment of} (x,y)\}$$

(3)

Example: Let's consider:

$$c_{del} = c_{ins} = 2 \tag{4}$$

and

$$c_{sub}(a,b) = \begin{cases} 3, & \text{if } \{a,b\} = \{C,G\} \text{ or } \{a,b\} = \{A,T\}. \\ 4, & \text{otherwise} \end{cases} \tag{5}$$

The solution for the previous example is d(x, y) = 4 and the minimal alignment corresponding to this edit distance is the second configuration.

3 Technical Challenges:

3.1 Data structure choice:

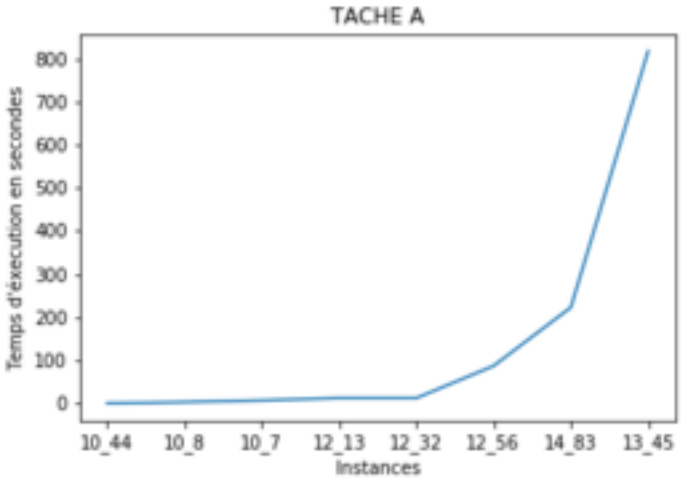
The operation I was going to use the most was accessing the subwords $x[i, j]$ when computing the edit distance. So, I chose an array to represent the DNA sequences because the access complexity is $O(1)$.

3.2 Time complexity:

3.2.1 Naive approach

I started by a recursive algorithm enumerating all possible paths from x to y and returning the distance of the shortest path. As expected with the theoretical study, its complexity grew up exponentially ($O(2^n)$). The following graph shows the execution time in seconds for 8 instances (format:x length_instance number

)

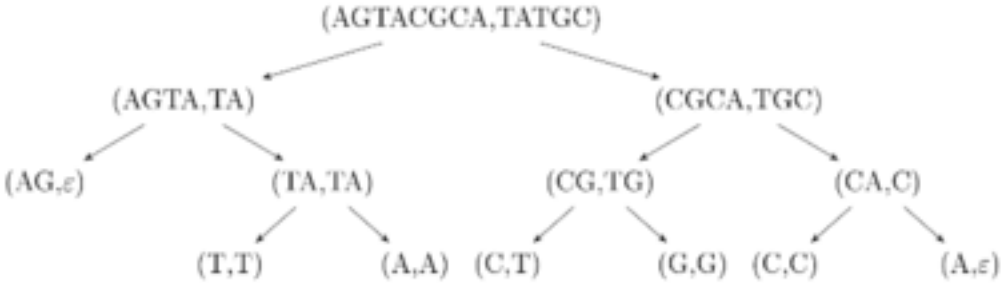


3.2.2 Dynamic approach

my project partner implemented the function DIST1 returning the edit distance and used it to implement the function PROG_DYN which returns the optimal alignment.
The time complexity and the memory complexity are both in **O(n*m)**

3.3 Divide and Conquer approach

The idea is to divide x into $x^1.x^2$ and y into $y^1.y^2$ and compute recursively $(\overline{s},\overline{t})$ the optimal alignment of (x^1,y^1) and $(\overline{u},\overline{v})$ the optimal alignment of (x^2,y^2), then concatenate the results to obtain $(\overline{s}.\overline{u},\overline{t}.\overline{v})$ which I have proved was the optimal alignment of (x,y). I divided x on its middle index $i^* = |x|/2$. To find the index j^* where to divide y, I implemented a function **cut**.



Cut function:
We introduce the following notation:$n = |x|$, $m = |y|$

$$D(i,j) = d(x_{[1,..,i]}, y_{[1,...,j]}) \tag{6}$$

$$D(n,m) = d(x,y) \tag{7}$$

$$Al(i,j) = (\overline{u},\overline{v}) | (\overline{u},\overline{v}) \text{ is an alignment of} (x_{[1,...,i]}, y_{[1,...,j]}) and l = |\overline{u}| \tag{8}$$

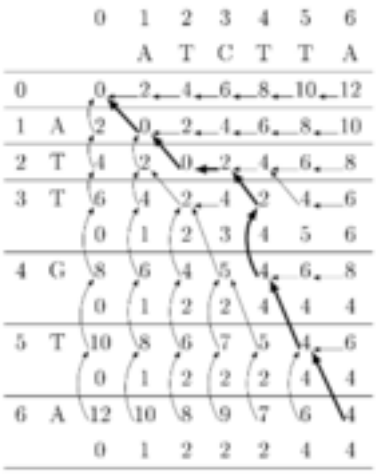
Then:

$$C(\overline{u},\overline{v}) = C(\overline{u}_{[1,..,l-1]}, \overline{v}_{[1,..,l-1]}) + C(\overline{u}_l, \overline{v}_l) \tag{9}$$

$$D(i,j) = min \begin{cases} \text{D(i-1,j-1)+ } c_{sub}(x_i, y_j) \\ \text{D(i,j-1)+}c_{ins} \\ \text{D(i-1,j)+}c_{del} \end{cases} \quad (10)$$

The following table represents an array containing the values of D for our first example.
The bold arrows represent the optimal path. (vertical arrow is a delete, horizontal arrow is an insertion, diagonal arrow is a substitution).
j^* is the index of the column in which the path intersects with the line i^*.Our cut function returns j^*
example: Let's consider $i^* = 3$, then $j^* = 4$

—



The time complexity of cut is **O(m*n)** , **The memory complexity** is **O(m+n)**.
I have implemented the function **SOL2** using the function cut and the divide and conquer strategy. Its memory complexity is **O(m+n)**.

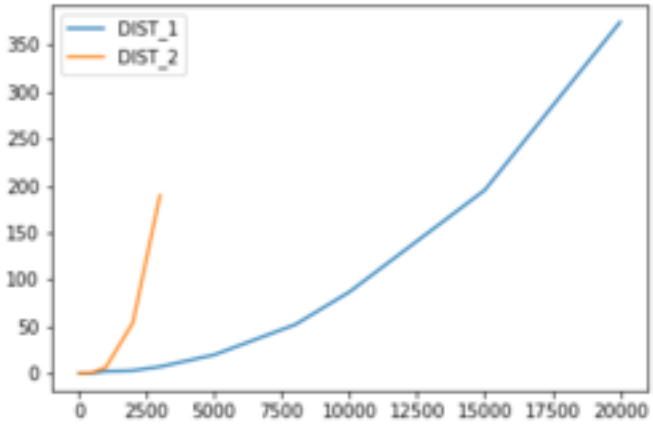
3.4 Memory complexity

3.4.1 Dynamic approach:

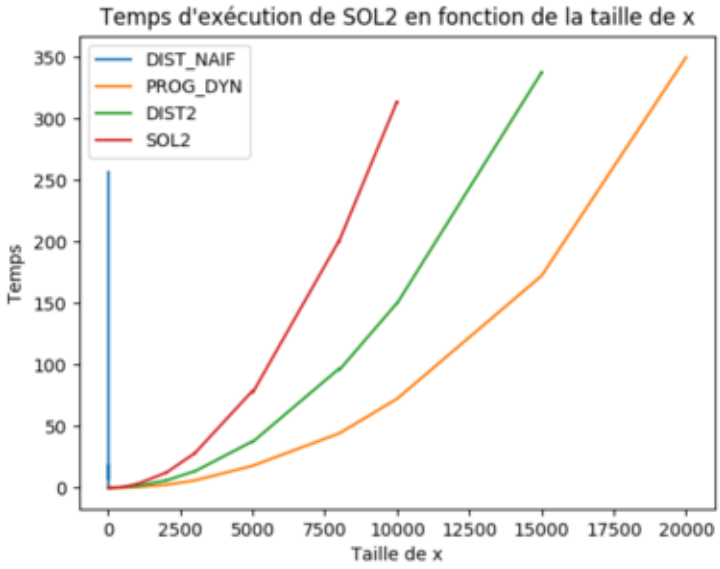
To fill the line i of the array D, we only need the information stored in the line i-1. My project partner managed to decrease the memory complexity of DIS1 to a linear complexity and implemented DIST2. But, she couldn't use DIST2 to compute the optimal alignment, because PROG_DYN needs access to the full table D.

3.4.2 Divide and conquer approach:

To find the corresponding cut j^*, as in DIST2, we only need the information stored on the lines i and i-1. The result is a linear complexity for memory consumption.
The following graph shows the memory consumption for 9 instances.



3.5 Experimental comparison



4 Conclusion:

The results show that the dynamic approach is faster than the Divide and Conquer Approach. This may be caused by the fact that the function **cut** is called at every recursive call of the function SOL2. We could improve this experimental time by parallelizing the recursive calls. The memory complexity of the divide and conquer approach is smaller than the dynamic approach. Our conclusion is that, The divide and conquer approach is the best solution for this problem.

5 What I gained from the experience

Learning the importance of studying the complexity of a problem to choose the right data structure and comparing different approaches to solve it efficiently.

6 References

- https://www.assistancescolaire.com/eleve/1re/sciences-de-la-vie-et-de-la-terre/reviser-le-cours/1_t_03/print?print=1printSheet=1
- https://www.youtube.com/watch?v=EUAVbQ1ZQW4
- https://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2018/ue/3I003-2018oct/qnc/ProgDyn.pdf
- https://github.com/InasK6/Measurement-of-DNA-sequence-alignment