ECOLE POLYTECHNIQUE

From ADN to formation of proteins: how to align sequences?

Programming Project INF421

Jenny Mansour and Manon Romain

06/01/2017

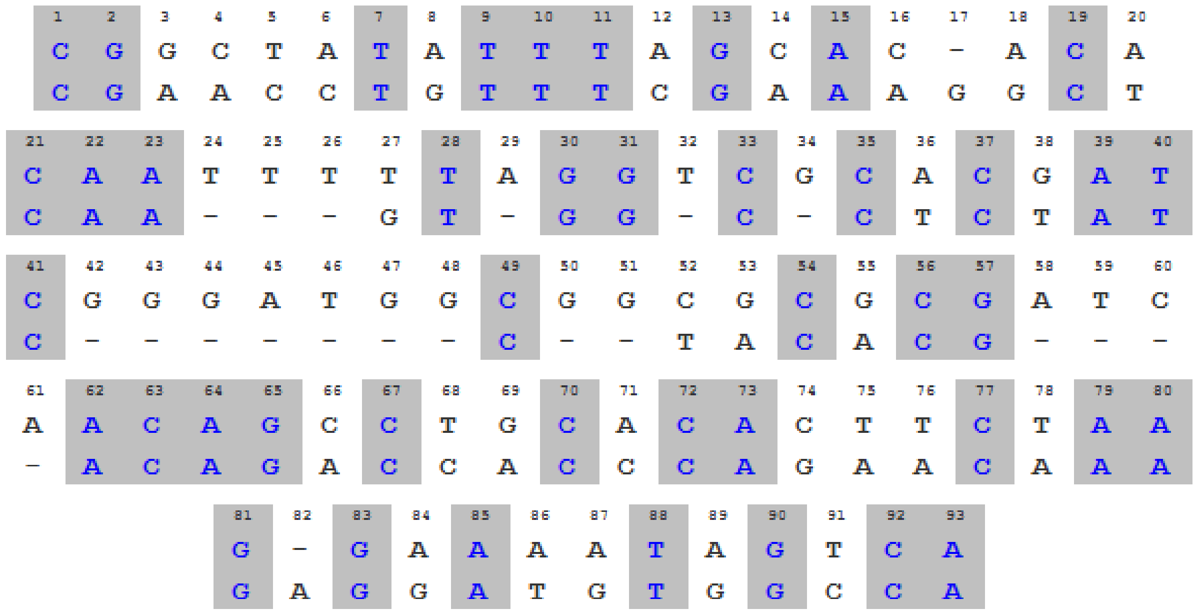

Table of contents

I. Auxiliary considerations

Recursive versus Iterative Implementation

At first, we did the implementation of all tasks using recursive algorithms, as it seemed more natural to use. However, when testing on the (long) test files provided our programs were raising a Stack Overflow error, even when we were just using tail-call. Doing a bit of research, we learnt that Java did not support the tail call optimization - that transforms tail call into loop when compiling the code - so we had to resort to using iterative algorithms for all our programs.

How to Read and Print Files?

In the package `InputOutput`, we implemented two classes, `ReadFile` and `PrintOptimalAlignment`.

- `ReadFile` class needs a String with the name of a file (existing at the root of the project) to be constructed. This file should contain two lines: the first one represents the first sequence and the second one the second sequence. This function thus separates those two lines and saves them in two different class variables first_line and second_line.

- `PrintOptimalAlignment` takes in arguments a list of Booleans and two lists of characters that give the list of letters and hyphens of best alignment for the task considered. Thus, this function prints a window with the two sequences, one above the other, with a special color when the letters match in the alignment.

II. Longest common subsequences and edition distance

Task 1. Naïve algorithm

```
Let L1 be the first sequence, with n1 its length
Let L2 be the second sequence, with n2 its length.
Let Sub1 and Sub2 be arrays of length $2^{n1}$ and $2^{n2}$.
Add all subsequences possible for L1 and L2 in Sub1 and Sub2, ordered by decreasing length of the subsequences.
For an integer i going from 0 to $2^{n1}$:
    For an integer j going from 0 to $2^{n2}$:
        If Sub1[i] and Sub2[j] have the same length, compare the two strings :
            If they are identical, call it *subsequence* and return *subsequence*
Return *subsequence* being the empty list
```

Complexity

The two loops have a 2^{n1+n2} complexity. In each loop, we compare two strings of length an integer between 0 and max (n1, n2).
Hence, the complexity of this naïve algorithm is more than 2^{n1+n2}. It is in the range of $S*2^{n1+n2}$, S being the sum from 1 to max (n1, n2) of all integers.

Now let's improve this complexity with dynamic programming. For that, we tried - in task 2 - to compute a matrix with the score of alignments. This matrix permits to the function `computeOptimalAlignment` to solve our problem by solving sub-problems and from their solutions obtain the original solution.

Task 2. Longest common subsequence

Equivalence

To go from a word *s* to a word *t*, there is a minimal number of operations, knowing that the possible operations are inserting, deleting or transforming a letter. Inserting a letter in a list means adding a hyphen at the same position in the other list (to keep the letters of the common subsequence above each other). In addition, for the same reason, deleting a letter in a list adds a hyphen in the other one. Eventually, transforming a letter means having two letters above each other that are different and thus that do not belong in the common subsequence.
Thus, having a minimal number of those three operations is equivalent to having a minimal sum of hyphens (inserting/deleting) or different letters in both lists (transforming).

Input A file in which line one is the first sequence s_1 (length n_1) and line two the second one s_2 (length n_2).
Output None - A window is printed representing the subsequence that scores best using the Blosum50 matrix

Algorithm

See Task 3. Optimal Alignment. Here we just compute the subsequence (without '-' when necessary).

Task 3. Optimal Alignment

Input A file in which line one is the first sequence s_1 (length n_1) and line two the second one s_2 (length n_2).
Output None - A window is printed representing the subsequence that scores best using the Blosum50 matrix

Algorithm
The algorithm is the same than in Task 4. Substitution matrices, the only difference is that instead of the score that's in the Blosum50 matrix, the score is 1 if the letters are the same and 0 if we need to replace, insert or delete a letter.

$$\boldsymbol{M_{ij}} = \begin{cases} \boldsymbol{M_{i-1,j-1} + 1} \text{ if } \boldsymbol{a_i = b_j} \\ \max\begin{cases}\boldsymbol{M_{i,j-1}} \\ \boldsymbol{M_{i-1,j}}\end{cases} \text{ if not} \end{cases}$$

III. Refinements on alignment

We used very similar functions in all tasks:

- In tasks 4, 5 and 6, `computeAlignmentMatrix` computes two matrices : the first matrix contains the score of transformations – the score of transforming the sequence s_1[0 … i-1] into s_2[0 … j-1] is in the coefficient (i,j) – and the second one contains the last operation (*e* for equality, *r* for replacement, *d* for deletion and *i* for insertion) for this transformation.
- `computeOptimalAlignment` traces back the optimal alignment – in our case, two linked lists of characters (letters or "-") and a linked list of Booleans indicating whether the letters match – from the matrix computed before.
- `OptimalAlignment` initializes both matrices, uses the two functions described above, and prints – nicely – the optimal alignment.

Task 4. Substitution matrices

Input A file in which line one is the first sequence s_1 (length n_1) and line two the second one s_2 (length n_2).
Output None - A window is printed representing the subsequence that scores best using the Blosum50 matrix.

Algorithm
Let M be a empty square matrix of size $(n_1+1)*(n_2+1)$, each element will contain the score of the best alignment between s_1[1…i] and s_2[1…j].
Let Op be a empty square matrix of size $(n_1+1)*(n_2+1)$, each element will contain the last operation to perform to transform s_1[1…i] into s_2[1…j].
Implementation: char[][] (*e* for equality, *r* for replacement, *d* for deletion and *i* for insertion).

Initialization:
We initialize the first line and first column of our matrix: if we try to compute the best alignment between an empty string and a string s, the score will be the

$\sum_{a_i \in s} \mathbf{score}(a_i, -)$ and the operation will be deleting (if s is the first sequence) or inserting (if it is the second one) all letters.

Recurrence:
We fill both matrix using the following recurrence formula:

$$M_{ij} = max \begin{cases} M_{i-1,j-1} + \mathbf{score}(a_i, b_j) \text{ then } Op_{i,j} = r \text{ or } e \\ M_{i,j-1} + \mathbf{score}(a_i, -) \text{ then } Op_{i,j} = d \\ M_{i-1,j} + \mathbf{score}(-, b_j) \text{ then } Op_{i,j} = i \end{cases}$$

Implementation: two for-loops.

Let's explain this recursive formula. When computing the best score of an alignment between $s_1[1...i]$ and $s_2[1...j]$, we have three options to consider:

- We can align a_i and b_j and find the best alignment $s_1[1...i-1]$ and $s_2[1...j-1]$, Then, we need to replace 'r' a_i by b_j (we write 'e' for equality if a_i and b_j are the same just to color them properly when printing).

score = $M_{i-1,j-1}$	a_i
	b_j

Total score = $M_{i-1,j-1} + \mathbf{score}(a_i, b_j)$

- We can delete a_i and find the best alignment $s_1[1...i-1]$ and $s_2[1...j]$, Then, we need to delete 'd' a_i .

score = $M_{i-1,j}$	a_i
	$-$

Total score = $M_{i-1,j} + \mathbf{score}(a_i, -)$

- We can insert b_j at position I and find the best alignment $s_1[1...i]$ and $s_2[1...j-1]$, Then, we need to insert 'i' .

score = $M_{i,j-1}$	$-$
	b_j

Total score = $M_{i,j-1} + \mathbf{score}(-, b_j)$

When we are finished filling both M and Op, we need to backtrack to computes to best alignment. We store it in two list of characters (letters or '-') and one list of Booleans stating whether the letters are matching (we will color them later).
Implementation: LinkedList<Character> and LinkedList<Boolean>

Starting from the lengths of both strings in Op, and while i and j are not both equal to zero, if $Op_{i,\,j}$ is:

- 'r' or 'e' we add a_i and b_j to the appropriate lists and decrease i and j.
- 'i' we add '-' and b_j to the appropriate lists and decrease j.
- 'd' we add a_i and '-' to the appropriate lists and decrease i.

Complexity

As we fill the each element of the matrices, the complexity of computeAlignmentMatrix is $\theta(n_1 * n_2)$.
The complexity of the backtrack function is $\theta(n_1 + n_2)$ because we start from $i + j = n_1 + n_2$, and at each step $i + j$ decreases by one or two.
Therefore, the final time complexity is $\theta(n_1 * n_2)$.

Task 5. Affine gap penalty

Input A file in which line one is the first sequence s$_1$ (length n$_1$) and line two the second one s$_2$ (length n$_2$).
Output None - A window is printed representing the subsequence that scores best using the Blosum50 matrix and affine gap penalty

Algorithm

The algorithm is very similar to the one in Task 4. Substitution matrices, but the recurrence formula is now:

$$\boldsymbol{M_{ij} = max \begin{cases} M_{i-1,j-1} + \text{score}(a_i, b_j) \\ \max\limits_{k \in [\![1,j]\!]} M_{i,j-k} - \text{cost\_gap}(k) \\ \max\limits_{k \in [\![1,i]\!]} M_{i-k,j} - \text{cost\_gap}(k) \end{cases}}$$

$$\boldsymbol{\text{cost\_gap}(k) = \begin{cases} \text{0 if it is a end gap} \\ (\text{opening gap penalty} + (k-1) * \text{increasing gap penalty}) \text{ if not} \end{cases}}$$

Besides, we initialize the first line and column to 0 because we are told gap at the beginning doesn't count.

Let's explain the difference:

- If $\boldsymbol{a_i}$ is the letter that ends a k-gap in the second sequence, the score is as described below. We will maximize the score of such alignment over all possible gaps (k going from 1 to i). Besides, if $\boldsymbol{a_i}$ is the last letter of the first word, the gap doesn't count. If the maximum is found there, the corresponding operation will be deleting 'd'.

score $= \boldsymbol{M_{i-k,j}}$	$\boldsymbol{a_{i-k+1}}$	$\cdots$	$\boldsymbol{a_i}$
	$-$	$-$	$-$

Total score $=$ $\boldsymbol{M_{i-k,j} - \text{cost\_gap}(k)}$

Complexity

Each step of the loop does $\theta(i + j)$ operations, so according to the calculation below, the final complexity will be $\theta(\boldsymbol{n_1 n_2 (n_1 + n_2)})$.

$$\sum_{i=1}^{n_1}\sum_{j=1}^{n_2} i+j = \sum_{i=1}^{n_1}\left(n_2 i + \sum_{j=1}^{n_2} j\right) = \frac{n_1(n_1+1)n_2}{2} + \frac{n_1 n_2(n_2+1)}{2} = \frac{n_1(n_1+n_2+2)n_2}{2}$$

Task 6. Local Alignment

Input A file in which line one is the first sequence s_1 (length n_1) and line two the second one s_2 (length n_2).
Output None - A window is printed representing the local subsequence (not the whole sequence) that scores best using the Blosum50 matrix and affine gap penalty

Algorithm

The algorithm is the same as before (see Task 5. Affine gap penalty), expect that we just want the optimal local alignment, so if the alignment between s_1 and s_2 has a negative score we don't select it.

$$M_{ij} = max\begin{cases} 0 \\ M_{i-1,j-1} + \text{score}(a_i, b_j) \\ \max\limits_{k\in[\![1,j]\!]}(M_{i,j-k} - \text{cost\_gap}(k)) \\ \max\limits_{k\in[\![1,i]\!]}(M_{i-k,j} - \text{cost\_gap}(k)) \end{cases}$$

The backtrack function is also different: it starts from the global maximum of the score matrix and goes up until it meets a zero, then it will not consider the beginning or end of the sequences that don't improve the local score.

IV. Basic Local Alignment Search Tool

Task 7. Beginning Perfect Matches

Input A file in which line one is the new sequence g (length n) and line two the second one t (length m).
Output: None - Prints indices in the second sequence that correspond to beginning of perfect matches between an element of S_g and a subword of t.

Given the new sequence s, the database sequence t, the threshold *th* and an integer k, `task7` computes the list of indices that are beginning of perfect k-matches between a seed and a k-subword of t. The seeds are given by the function `findSeeds`.

To perform this, we need some auxiliary functions:

- `computeScore` computes the score between two strings using the Blosum50 matrix.

- `breakInto` takes an integer k and string s and computes the stack of all possible k-subwords from s (a k-subword being k consecutive letters) by adding them with a loop in the stack. We choose to put the k first letters on top of the stack.

- `computeAllWords` is a function that takes an integer k, computes from the alphabet all possible k-words, and puts them in a stack. In order to o do that, this function calls an auxiliary function:

- `computeAllWordsAux` takes in argument the length of the words we are looking for, the alphabet (containing all letters of Blosum50 matrix alphabet), the current word we are building and the Stack of possible words we already computed. The function adds all possible letters at the end of the word if it is not long enough and otherwise stops when the current word has the correct length.

`findSeeds` takes in argument k, s and the threshold *th*. It returns in a Map (Integer → String) all k words that have a score with a k-subword of s that is superior to *th* times the score of the subword with itself. When a seed is found, we store it in the Map its key being the index of the corresponding subword in *s*.

The function `task7` then creates a binary search tree, which contains all seeds, and computes all the k-subwords of t, with the function `breakInto`.

Optimization of the complexity
To optimize the complexity, we decided to store seeds in a Binary Search Tree. This implementation permits to search elements in logarithmic complexity, which is much better than what we had previously.

Task 8. BLAST

Input A file in which line one is the new sequence g (length n) and line two the second one t (length m).
Output None - Prints all local alignments with sufficiently high scores

Task 8 is computed through `findAlignments`.
`findAlignments` implements the BLAST algorithm. It takes in argument the length k of the subwords used, the new sequence *s*, the database sequence *t*, and the threshold *th* and *thl* used in the algorithm.

- First, the function finds the seeds of the new sequence and creates the Binary Search Tree (that scores seeds and their indexes in the sequence), through `findSeeds`.
- Then, it computes the k-subwords of the database sequence.

- It searches for k-words in the database matching our seeds. When it does, it stores the begin indexes of the match in both sequences in the Map `IndexesPerfectMatches`.
- Eventually, it tries to extend the alignments, by storing them in a list of `IndexesLengthScore` and comparing their score to the score of g with itself. The current match is the alignment of a seed and a k-subword of `t` and we try to extend backward and forward. If it has a sufficient score and hadn't already been computed, we store it.

  N.B: `IndexesLengthScore` is a new Data Structure that stores alignments as wanted; it stores the index of the first letter of the alignment in both sequences, its length and score (in four separate class variables respectively: `indexInS`, `indexInT`, `length` and `score`).

For this step, two auxiliary functions are needed: `front` and `back`.

- `front` takes two strings s and `t` and a current local match between them and updates curr_match until it reaches its maximal length (just from left to right). Until it reaches the end of one of the sequences, it continues and if we benefit from adding a letter, it does it.
- `back` does exactly the same thing but from right to left.