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

Louis GUO, Benjamin PETIT - X2015

January 8, 2017

1 Longest common subsequence and edition distance

1.1 Length of the longest common subsequence

Let us consider two (DNA/amino acids/...) sequences $S = S_0...S_{n-1}$ and $S' = S'_0...S'_{m-1}$ represented by strings. One interesting characteristic we could look at to find out how similar these two sequences are is their longest common subsequence, a common subsequence being given by two increasing sequences of indices of same length, $i_1, ..., i_p$ and $j_1, ..., j_p$ such that:

$$S_{i_1}S_{i_2}...S_{i_p} = S'_{j_1}...S'_{j_p}$$

For instance, if $S = ACTGCCGT$ and $S' = CAGCCAAT$, $CGCCT$ is a common subsequence of S and S'. Two strings can have a very large number of common subsequences, especially when the letter set is small. A naive approach to compute one longest subsequence would be to compute all subsequences and find the longest ones, using the following algorithm:

Algorithm 1 $LCS(S, S')$: Naive approach to the longest common subsequence problem

if S or S' is empty **then**
  Return 0
else if $S_0 = S'_0$ **then**
  Return $max(1 + LCS(S_{1:}, S'_{1:}), LCS(S_{1:}, S'), LCS(S, S'_{1:})$
else
  Return $max(LCS(S_{1:}, S'), LCS(S, S'_{1:})$
end if

This algorithm has a terrible complexity. It basically goes through building all subsequences of the shortest of both strings - about $O(2^n)$ operations are required, which is barely usable. We thus have to go through a different approach, which will be based on dynamic programming. We propose to use the following method:

Let $l_{i,j}$ be the length of the longest common subsequence of $S_{:i}$ and $S_{:j}$ for $i \in \{0, ..., n\}$ and $j \in \{0, ..., m\}$. We prove that

$$l_{i,j} = max \begin{cases} l_{i-1,j-1} \ if \ S_{i-1} \neq S'_{j-1} \\ l_{i-1,j-1} + 1 \ otherwise \\ l_{i-1,j} \\ l_{i,j-1} \end{cases}$$

Having a longest common subsequence $S_{i_1}...S_{i_p} = S'_{j_1}...S'_{j_p}$ of $S_{:i} = S_0...S_{i-1}$ and $S'_{:j} = S'_0...S'_{j-1}$, we can indeed build longest common subsequences of their prefixes by distinguishing two different cases, depending on whether or not the last character is in the common subsequence.

We can easily build our matrix $(l_{i,j})_{i=0,...,n,\ j=0,...,m}$ line after line, storing all values in order to calculate them only once. At the same time, in a second matrix $(S_{i,j})$, we can keep track of the optimal subsequences that are built throughout the process, so that $S_{i,j}$ contains an optimal common subsequence of $S_{:i}$ and $S'_{:j}$. At the end of the execution, $l_{n,m}$ contains the optimal length, and one maximal length common subsequence of S and S' will be stored in $S_{n,m}$.

Algorithm 2 A dynamic programming solution to the longest common subsequence problem

```
SubSequences_{i,j} ← ""
for i = 1, ..., n do
    for j = 1, ..., m do
        if S_{i-1} = S'_{j-1} then
            SubSequences_{i,j} ← maxLengthString(SubSequences_{i-1,j-1}.S_i, SubSequences_{i-1,j}, SubSequences_{i,j-1})
        else
            SubSequences_{i,j} ← maxLengthString(SubSequences_{i-1,j-1}, SubSequences_{i-1,j}, SubSequences_{i,j-1})
        end if
    end for
end for
return SubSequences_{n,m}
```

Our dynamic programming implementation of the LCS problem is based on the previous remark, and can be launched using the following command (on Linux systems): *java LCS STRING1 STRING2*. It displays as an output the longest common subsequence of *STRING1* and *STRING2* and its length.

1.2 Editing distance and alignment of sequences

1.2.1 An interesting result on optimal alignments

The editing distance between S and S' is the smallest number of operations to change S into S'; an operation can be inserting a letter, deleting a letter or transforming a letter into another one. One can easily see that calcultating the editing distance and computing the longest common subsequence are two problems that are quite linked. It might indeed be useful to find the longest common subsequence to compute the editing distance so that as many letters as possible remain untouched and, thus, use as few operations as possible to turn S into S'. Finding the editing distance between S and S' is in fact equivalent to the following problem:

Finding where to insert hyphens in S and S' so that, when put next to each other, both sequences match on as many letters as possible, using as few hyphens as possible.

Proof. (Proof of the equivalence)

Assume that we have a minimal and order list L of edits from S to S', containing i insertions, d deletions and t transformations, letting u letters untouched. We use the following process to turn it into a minimal and optimal hyphenation H:

- For each letter that is to be deleted in S, we insert an hyphen at that letter's position in S'.
- Every time we insert a letter in S, we insert an hyphen instead.

Now, suppose that there is some hyphenation H' that leads to (strictly) more letter matches with less or as many hyphens. Then, using this hyphenation and "reversing" the previous technique, we can build another list of edits (left to right):

- For each hyphen in S', we delete the corresponding letter in S.
- For each hyphen in S, we insert the corresponding in S.
- All untouched letters are either matches or have to be transformed.

This leads to a list of edits L' with $i' + d' \leq i + d$ (less hyphens) and $d < d'$ (we assume we have an hyphenation which is strictly better than H). Remark that we thus have $u' < u$, so that $Length(L) = i' + d' + u' < i + d + u = Length(L)$, whis is absurd because L is supposed to be an optimal list of edits.

Then finding an optimal sequence alignment (hyphenation, basically) is equivalent to computing the edit distance between these two sequences. □

Computing the edit distance between S and S' can be done easily, following a dynamic programming approach very close to what we did at (1.1). If we let $l_{i,j}$ be the length of the longest common subsequence of $S_{:i}$ and $S_{:j}$ for $i \in \{0, ..., n\}$ and $j \in \{0, ..., m\}$, we have:

$$l_{i,j} = max \begin{cases} l_{i-1,j-1} \ if \ S_{i-1} \neq S'_{j-1} \\ l_{i-1,j-1} + 1 \ otherwise \\ l_{i-1,j} \\ l_{i,j-1} \end{cases}$$

We can then easily compute the optimal alignment using the longest common subsequences. As we build the distance matrix, we store in a separate matrix - which we call *origin* - the value from which the max is obtained (*"TopLeft"* for $(i-1, j-1)$, *"Left"* for $(i, j-1)$ and *"Top"* for $(i-1, j)$). It is then quite easy to go backwards through the matrix from its end (m, n) to the origin $(0, 0)$ as we build the hyphenation. Basically, according to the above proof, we can proceed as follows:

- If the origin of value (i, j) is its top-left value, we simply add the two corresponding characters to their respective strings.

- If value (i, j) comes from its left value, we add the j-th character of string 2 to its corresponding string, and add an hyphen in front of it in the first string.

- If value (i, j) comes from its top value, we add the i-th character of string 1 to its corresponding string, and add an hyphen in front of it in the second string.

1.2.2 On our implementation and its complexity

We propose a Java implementation of this algorithm in a class called "*BasicAlignment*". It takes two strings as arguments and displays the optimal alignment. It can be called using the following scheme (on Linux systems): *java BasicAlignment STRING1 STRING2* . It displays one optimal alignment in a quite fancy way (using ANSI codes on Linux consoles / might not work on Windows).

Our program's complexity can be described as follows:

- **Space complexity:** Depending on our approach, we need to store one or two n, m-sized matrices. One contains integers, the other one has 2-tuples of strings of size smaller than n and m. Our approach has a space complexity in $O(nm)$.

- **Time complexity:** Depending on our approach, we will have to go once or twice across a n, m-sized matrix and do constant cost operations at each step ; whatever possibility we might choose, the overall complexity will be in $O(nm)$.

We observe that our program's theoritical complexity is of course way more efficient than the naive approach's one.

2 Refinements on alignment

2.1 Substitution matrices

All alignments are not equivalent from a biological point of view. All pairs of letters are given a (possibly negative) value, and we define the value of an alignment of two sequences as the sum of the values of the letters that are matched (the matching of a letter with an hyphen also has a value). For biological purposes, we may have to compute the optimal alignment - that is to say the alignment that has the highest score (with as few hyphens as possible). We only have to modify the previous algorithm a little bit, replacing our $0-1$ cost policy by the values of the couples of letters that get aligned when we apply the insertion/deletion/transformation operation. The min also turns out to become a max because we want to compute the largest score as possible.

$$d_{i,j} = max \begin{cases} d_{i-1,j-1} + Value(S_{i-1}, S'_{j-1}) \\ d_{i-1,j} + Value(S_{i-1}, '-') \\ d_{i,j-1} + Value(S_{j-1}, '-') \end{cases}$$

Our implementation is available in the *Alignment* class, and can be launched using the following command: *java Alignment STRING1 STRING2*.

2.2 Affine penalty

Opening a gap in a sequence to insert or delete letters is quite a rare event in the world of genetics. It is then more probable to insert two letters at the same spot than at two different ones. We thus add a penalty of each gap opening, and an additional penalty for increasing the size of an already-existing gap when building an alignment. To keep track of gap openings, we generate one more (m, n)-sized boolean matrix called *inGap*, which contains true at position (i, j) if this position was reached creating or increasing a gap at the last step of the process. This matrix is constantly updated as we build our main score matrix, so that our equation becomes:

$$d_{i,j} = max \begin{cases} d_{i-1,j-1} + Value(S_{i-1}, S'_{j-1}) \\ d_{i-1,j} + Value(S_{i-1}, '-') + INC/DEC\ GAP\ PENALTY \\ d_{i,j-1} + Value(S_{j-1}, '-') + INC/DEC\ GAP\ PENALTY \end{cases}$$

The complexity of the implementation basically remains the same in order (even though its constant factor is increased by these refinements). Our program can be launched using the following command: *java AffinePenalty STRING1 STRING2 OPENING_GAP_PENALTY INCREASING_GAP_PENALTY*; the two last parameters have to be ints (the code can be easily modified to use doubles or floats instead).

2.3 Local alignments

The problem of finding optimal local alignments can basically be solved by using the previous algorithm on all subsequences of the two strings it takes as parameters to find the ones with the highest score. The complexity thus becomes quite higher: each string has $O(length^2)$ substrings, resulting in $O(m^3n^3)$ time complexity. Our implementation, which returns two pairs of optimal indices, can be launched using the following command: *java LocalAlignment STRING1 STRING2 OPENING_GAP_PENALTY INCREASING_GAP_PENALTY.*

3 Implementation of the BLAST algorithm

BLAST (Basic Local Alignment Search Tool) is an heuristic algorithm that is used by geneticists to find good enough local alignments in much lower complexity than the previous algorithm. It might not find an optimal matching, but works pretty well in practice, which is why it is still in use in labs today.

3.1 Data structure

As the charset in use in genetics is quite small, it might be way lighter - in order to avoid sequence repetitions and superfluous calculations - to store our set of subwords in a keyword tree: each subword is inserted in a tree which nodes are letters. The branch leading to some node is then the prefix to its child branches, so that each subword is uniquely represented by a whole branch of the keyword tree - leading to a uniquely defined leaf node. One huge advantage of this data structure is that score calculations are way lighter than they might be if we had used an ordinary list to store all subwords. To calculate the score of a subword against the whole subword set, we simply have to calculate the score of its first letter against the tree's root, then propagate this value to all of the root's child trees and add the score of the subword's second letter against every child tree's root, etc. until we reach the tree's leaves - and the end of the subword. It is then easy to get the score of the subword against every single subword in the set, by going through the scores that are stored in the tree's leaves.

3.2 Finding the seeds and extending local alignments

Using the keyword tree structure, it is very easy to go through the scores of every subword of t against the whole set of subwords of g and find whether or not it is higher than the score of the subword of g aligned with itself times th. If so, we simply add this subword's beginning index to the seed index list. Once all seeds have been found, we try to extend them on both sides until the score starts decreasing. We then check whether this alignment has a higher score than its g part aligned with itself times th_l. If so, we display this alignment as a potential optimal alignment found by BLAST.

Our version of BLAST can be launched using the following command: *java BLAST g t th thl*. It displays all the local alignments found by the algorithm.

4 Optimal foldings

In this final part, we would like to focus on finding optimal (minimum-energy) protein foldings using a 2D hydrophobic-hydrophilic model. Finding an optimal solution is a well-documented NP-hard problem : one must indeed enumerate all self-avoiding walks on a 2D lattice of given length. Solving such a problem can thus be a very resource-consuming task. We chose to optimize the process a little bit using folding trees (see the *FoldingTree.java*

file): instead of representing all elf-avoiding walks as separate objects, we chose to represent all of the solutions as part of a single tree. Each variation in the path is thus stored as a separate branch of the folding tree.

The folding tree is built using the *extend* method ; to add an amin acid to the sequence and calculate all the possible resulting foldings, we try to add it at the end of each branch of the tree, around the position of the last amin acid in the sequence. We thus make use of a backtracking procedure to check whether or not the position is available in the current working branch. The algorithm then calculates the scores over all branches of the folding tree, and (through the *OptimalFoldingIterator* class) returns the leaf node with the highest score. We then just have to backtrack that node's branch to obtain one optimal folding (there can be multiple optimal foldings, obviously).

One must keep in mind that processing an optimal folding is a very costly operation. Solving the folding problem over proteins which have a few dozens amino acids might take quite a long time - depending on the hardware you use. Even though it is quite hard to calculate the exact number of self-avoiding walks of length k on $\mathbb{R}^2$, one must keep in mind that such an algorithm will nevertheless remain exponential in terms of complexity.

The algorithm can be used as follows (on Linux systems): *java OptimalFolding SEQUENCE.*