Programming Project P2 – X2015

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

Project proposed by Marie Albenque – albenque@lix.polytechnique.fr

Lucas Lugão Guimarães – lucas.lugao-guimaraes@polytechnique.edu
Alexandre Ribeiro João Macedo – alexandre.macedo@polytechnique.edu

Task 1. We can solve the longest common subsequence (LCS) for two sequences $s = s_1 \dots s_n$ and $t = t_1 \dots t_m$ in a naive way by considering all the 2^n subsequences of s and determining if it is a subsequence of t. The last step can be done with a complexity of $\mathcal{O}(m)$, which leaves the naive algorithm with a complexity of $\mathcal{O}(2^n m)$.

Task 2. In order to solve the LCS in a more efficient way we can use dynamic programming (DP). We denote $LCS(s,t)$ the solution of the problem given the inputs $s = s_1 \dots s_n$ and $t = t_1 \dots t_m$. In the case where $s_m = t_n = u$, the solution will be u appended to the end of the solution of $LCS(\hat{s} = s_1 \dots s_{n-1}, \hat{t} = t_1 \dots t_{m-1})$. In the case where $s_m \neq t_n$, the solution is the longest sequence between $LCS(\hat{s} = s_1 \dots s_{n-1}, t)$ and $LCS(s, \hat{t} = t_1 \dots t_{m-1})$.

In a more straight-forward way, we can solve it as described in [**?**]. We use a matrix $A_{m,n}$ such that the elements $a_{i,j}$ corresponds to the length of a LCS of $\hat{s} = s_1 \dots s_i$ and $\hat{t} = t_1 \dots t_j$. It is clear that we initialize the matrix as $a_{i,0} = a_{0,j} = 0$ for all $0 \leq i \leq n$ and $0 \leq j \leq m$. Then we compute the recurrence

$$a_{i,j} = \max \begin{cases} a_{i-1,j} \\ a_{i,j-1} \\ a_{i-1,j-1} + 1, & \text{if } s_i = t_j \end{cases} .$$

The first case corresponds to the case when s_i is not present in the LCS, the second corresponds to the case when t_j is not present in the LCS and the third case when both are present. The longest length is given by $a_{n,m}$. In order to get the LSC, we need to keep track of which of the cases were chosen. We can do this by creating a new matrix

$$b_{i,j} = \begin{cases} \uparrow, & \text{if } a_{i,j} = a_{i-1,j} \\ \leftarrow, & \text{if } a_{i,j} = a_{i-1,j} \\ \nwarrow, & \text{if } a_{i,j} = a_{i-1,j-1} + 1 \end{cases}$$

and print the character that corresponds to the case $\nwarrow$. That is, the LCS will composed by all s_i such that $b_{i,j} = \nwarrow$. This algorithms has a time complexity of $\mathcal{O}(nm)$.

Task 3. We can show the equivalence between the editing distance and the number of inserted hyphens added to the number mismatches, letters that are different between sequences, by interpreting the action of adding an hyphen to a sequence, lets say s, as an insertion in this sequence or as a deletion in t and interpreting a mismatch as a transform in either of these sequences.

The editing distance $ED(s,t)$ can be computed using the same idea of DP but in this case we initialize the matrix as $a_{i,0} = i$, $a_{0,j} = j$ to account for the not matches. As we want to minimize the distance the recurrence becomes

$$a_{i,j} = \min \begin{cases} a_{i-1,j} + 1 \\ a_{i,j-1} + 1 \\ a_{i-1,j-1} \qquad \text{if } s_i = t_j \end{cases}$$

and the traceback will be similar in a manner where gaps are added to the sequences when we do not have a match.

Task 4. To compute the alignment using the Blosum50 score matrix, we use the Needleman-Wunsch algorithm as named in [**?**] and described in [**?**]. In that case the initialization becomes $a_{0,0} = 0$, $a_{i,0} = a_{i-1,0} + Sc(s_i, -)$, $a_{0,j} = a_{0,j-1} + Sc(-, t_j)$ where $Sc(x,y)$ is the score between the alignment of x and y and $-$ is a gap. The recursion is $a_{i,j} = \max[a_{i-1,j} + Sc(s_i,-), a_{i,j-1} + Sc(-,t_j), a_{i-1,j-1} + Sc(s_i,t_j)]$ and the traceback will be similar to the described above. It is a more general case of ?THM? **??** where we define a penalty for adding a gap in a sequence, lets say s and aligning this gap to a base of t and we also define a score for aligning a base of s with t. The value of Sc that gives the same result as the LCS problem is $Sc(s_i,-) = Sc(-,t_j) = 0$, $Sc(s_i,t_j) = 1$ if $s_i = t_j$ and $Sc(s_i,t_j) = -\infty$ if $s_i \neq t_j$.

Task 5. In this question, we use a variant of the previous algorithm that considers an affine gap penalty and ignores penalty gaps at the beginning and at the end of sequences, it can be classified as a semi-global alignment with affine gap penalty. This is a much complex case that we take by reference [**?**]. Let's start by calling the opening gap penalty d and the increasing gap penalty e. We will need three matrices to keep the algorithm time complexity in $\mathcal{O}(nm)$. These matrices are $k_{i,j} = \max[k_{i-1,j} + Sc(s_i,t_j), p_{i,j}, q_{i,j}]$, $p_{i,j} = \max[k_{i-1,j} - d, p_{i-1,j} - e]$ and $q_{i,j} = \max[k_{i,j-1} - d, q_{i,j-1} - e]$. We initialize them by $k_{i,j} = 0$, $p_{0,j} = -\infty$, $q_{i,0} = -\infty$, $p_{i,0} =$ "not used", $q_{0,j} =$ "not used". Since this line and column are not used, we initialize them to 0. The initialization of D is due to the semi-global aspect of the case. That is, when we get to the first line or column, the gap costs are zero because they are in the beginning of the sequence.

In order to justify why we are using three matrices we can think of our problem as having three different states as show in figure **??** extracted from [**?**]. In this figure, the matches are represented by the state M and the gaps by I_x and I_y. See ?THM? **??** The matrix K keeps track of the cases where there is the alignment of of the bases and the matrices P and Q keeps track of the cases where we introduced gaps, in one or in the other sequence. In order to make it more clear why we need three states, we can think of a case where at first, after we opened a gaps, it is the better option to close the gap and make the match, without knowing that in the future, it will be necessary to reopen the gap, what would have a bigger cost than not making the match and keep the cost of the increasing gap. Lets give an example for a global alignment with affine gaps got from [**?**]. For simplicity, let's consider a score: match=5, mismatch=-2, increasing gap=-1, opening gap =-10. If we make the best alignment between CART and CAT, we get (CART,CA-T) with a score of 5. If we align CARTS and CAT, using the result of the previous alignment, we would get (CARTS,CA-T-) with a score of -5 but the optimal result is actually (CARTS,CAT- -) with score

of -3. The good result comes from the keeping track of the possibilities of not closing the gap, hence three matrices.

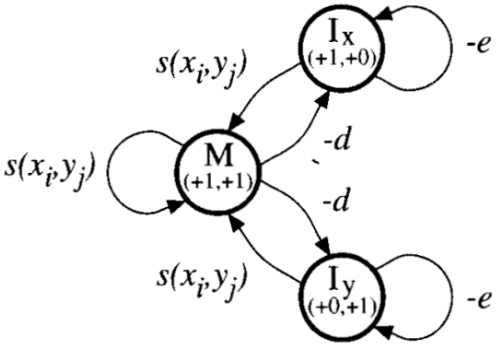

Figure 1: Diagram showing the three states.

Since we are on a semi-global alignment, we consider the best score equals to $\max[\max(k_{n,1}, \ldots, k_{n,j}), \max(k_{1,m}, \ldots, k_{i,m})]$ and we make the traceback from this point. In order to do it we need to consider, like in the problems above, which option was used to get the maximum value but this implies changing between states. Keeping in the state I_x or I_y means increasing the gap, and the state M means a match.

Task 6. For the local alignment with affine gaps, we can use an algorithm based in the Smith-Waterman algorithm, described in [**?**]. It is a particular case where we consider, in addition to the possibilities of adding a gap to one or the other sequence or not adding a gap, the possibility of stopping or starting at that point. This is implemented as a change in the DP, by adding the case 0 when getting the maximum score in the recursion. In other words, it is better to align the sequences locally than going with a negative score. This is done by changing the case in ?THM? **??** to $k_{i,j} = \max[k_{i-1,j} + Sc(s_i, t_j), p_{i,j}, q_{i,j}, 0]$.

Task 7. The algorithms presented so far are all exact, in a sense that they will return for sure the alignment with the best score but they all come with complexity of $\mathcal{O}(mn)$. If we use this to align a given sequence of size one thousand to a database of hundreds of millions of residues, the number of matrix entries will be the order of 10^{11} which would take a long time to compute, as explained in [**?**]. We should keep in mind that the human genome, for example, has $3,0.10^9$ base pairs. In order to solve this problem, we can use heuristic algorithms that not necessarily return the best alignment but that can run much faster. Such a example is BLAST.

In order to do this task we will use the Aho-Corasick, a classic exact match algorithm, as described in [**?**]. We start by creating the set $\mathcal{W}_g$ from g and from this set we create $\mathcal{S}_g$ by doing for each word of $\mathcal{W}_g$ the following procedure: we modify the first entry of this word by all the possibilities, meaning the entries of Blosum50, and we store those whose alignment score with the original sequence is greater than the defined threshold. And for each stored new word, we do the same procedure but instead of changing the first entry, we change the next one. We can check for changes only for the entries that had an alignment score above the threshold because an change in the sequence can only reduce the score when aligned to the original sequence. We keep doing this up to the point we modified all entries, that is, a number k of times. After we are done with the

creation of words based on the modification of the first entry of the original word, we do it for the next one, up to the end, that is k times. This procedure is illustrated in figure **??**. Once we have $\mathcal{S}_g$, we can run the Aho-Corasick, that will make the exact set matching of $\mathcal{S}_g$ and t. This is done by a refinement of a keyword tree. We construct the Aho-Corasick automaton, see [**?**], in $\mathcal{S}_g$, and we will run t in this automaton to get the results. The complexity will be $\mathcal{O}(h + m + z)$, where h is the some of the length of all words in $\mathcal{S}_g$ and z the number of occurrences of the pattern in t.

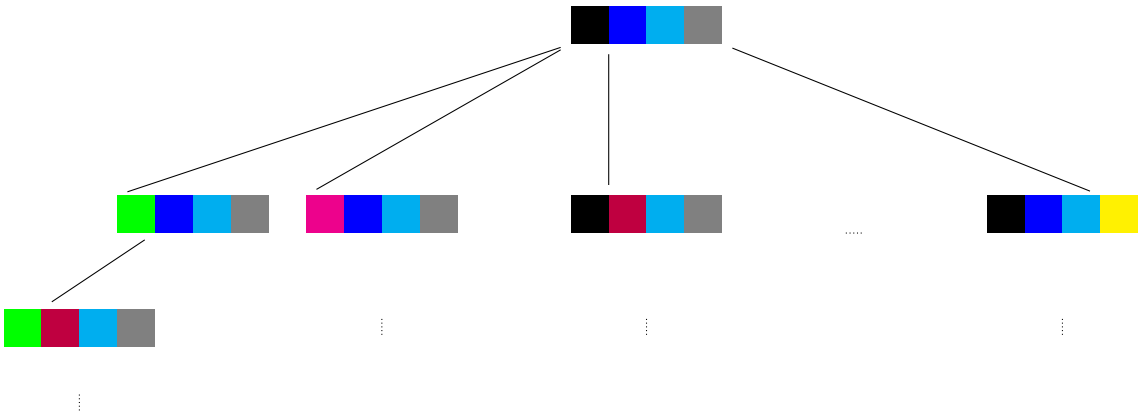

Figure 2: Initialization of $\mathcal{S}_g$

Task 8. In this task, once we have the working algorithm of ?THM? **??**, that give us the exact match, all we need to do is, once we have the match, we try to add an extra word from the original sequence g to the left and/or to the right. We keep doing this up to the point when we get an match that would have a negative score. From this sequences, we will keep those whose alignment score are above the set threshold. The meaning of the threshold, th_l, times [score of the alignment of g with itself] is how much "mutation" we allow for the local alignment to be still good. The output of the algorithm will finally be aligned sequences with its score, starting point in the sequence t and its length.

Comment 1. For ?THM? **??** and ?THM? **??** we used MATLAB and Bioinformatics Toolbox Release 2016a in order to test our algorithm. This toolbox has an implementation of the cases we were handling, making us able to validate our code and fine minor errors in the implementation.

Comment 2. We should keep in mind that as the diagram in figure **??** was extracted from [**?**], it does not use the same notation as us. The only purpose of this figure is to illustrate the three possibles states in the processes of finding the alignment.

References

[1] Neil C. Jones and Pavel a. Pevzner. *an Introduction to Bioinformatics algorithms*. MIT Press, 2004

[2] Richard Durbin, Sean R. Eddy, anders Krogh and Graeme Mitchison. *Biological Sequence analysis: Probabilistic Models of Proteins and Nucleic acids*. Cambridge University Press, 1998

[3] Carl Kingsford. *Lecture Slides*. Available at http://www3.cs.stonybrook.edu/ rp/class/549f14/lectures/CSE549-Lec04.pdf.

[4] Pekka Kilpelainen. *Lecture Slides*. Available at
https://www.cs.uku.fi/ kilpelai/BSA05/lectures/slides04.pdf