Computational Molecular Biology and Bioinformatics

Sequence Alignment

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

October, 2021

Basics

Sequence alignment is used for finding out the `similarity` between sequences by `aligning` them with each other.

To understand this, we need to explain the following two terms.

- **Similarity:** This gives a measure of how similar the sequences are by recognizing which parts of the sequences are alike and which parts differ.
- **Alignment:** This is a way of placing one sequence above the other in order to make clear the correspondence between similar characters or substrings from the sequences.

Importance

Sequence alignment is useful in different types of bioinformatics approaches. Some of these include the following.

- If the same gene is sequenced by two different labs and they want to compare the results.
- If the same long sequence is typed twice into the computer and we are looking for typing errors.
- Performing fragment assembly in programs to help large-scale DNA sequencing.
- To search for local similarities using large biosequence databases.
- For deriving the phylogenetic relationship between different organisms by comparing their DNA or protein sequences.
- To identify the sequence homologies that might establish the existence of a shared ancestry.

Varieties of sequence alignment

We are often interested in finding the best alignments between two sequences. This can be categorized into different types of problems as listed below.

- **Global alignment:** It refers to the alignment of the entire sequence pairs.
- **Semi-global alignment:** It refers to the alignment of prefixes and suffixes of the given pair of sequences not any arbitrary substrings.
- **Local alignment:** It refers to the alignment of just the substrings of a pair of sequences.
- **Multiple alignment:** It refers to the alignment of multiple (more than a pair) sequences.

Basics

Consider the two DNA sequences given by $\mathrm{GACGGATTAG}$ and $\mathrm{GATCGGAATAG}$. It is obvious that we can align them one above the other as follows.

GA–CGGATTAG

GATCGGAATAG

Note that the only two differences that are distinguishable in the above alignment are given below.

1. There appears an extra T in the second sequence (gap), and
2. There is a change from A to T in the fourth position from right to left (`mismatch`).

Scoring the similarity

Given an alignment between two sequences, each column of the alignment will receive a certain value depending on its contents and the total score for the alignment will be the sum of the values assigned to its columns. We can assign a score to it as follows.

- **Match:** If a column has two identical characters, it will receive value $+1$.
- **Mismatch:** Different characters will give the column value -1 (a mismatch).
- **Gap:** A space in a column drops down its value to -2.

The best alignment will be the one with the maximum total score.

Note: The choice of assignment of scores (parameters) has a significant impact over the algorithm.

Deriving similarity of sequences

We can store the different possible alignment scores in an array 'a' of dimension $(m+1) \times (n+1)$ by aligning the sequence prefixes.

The value for an entry (i, j), represented as $a[i, j]$, can be computed by looking at just three previous entries: those for $(i-1, j)$, $(i-1, j-1)$, and $(i, j-1)$. The reason is that there are just three ways of obtaining an alignment between $s[1 \ldots i]$ and $t[1 \ldots j]$, and each one uses one of these previous values.

In fact, to get an alignment for $s[1 \ldots i]$ and $t[1 \ldots j]$, we have the following three choices:

1. Align $s[1 \ldots i]$ with $t[1 \ldots j-1]$ and match a gap with $t[j]$.
2. Align $s[1 \ldots i-1]$ with $t[1 \ldots j-1]$ and match $s[i]$ with $t[j]$.
3. Align $s[1 \ldots i-1]$ with $t[1 \ldots j]$ and match $s[i]$ with a gap.

Basic constructions

We can formally define the similarity ($\mathcal{S}$) between two subsequences as follows:

$$\mathcal{S}(s[1\dots i], t[1\dots j]) = \max \begin{cases} \mathcal{S}(s[1\dots i], t[1\dots j-1]) + g \\ \mathcal{S}(s[1\dots i-1], t[1\dots j-1]) + p(i,j) \\ \mathcal{S}(s[1\dots i-1], t[1\dots j]) + g \end{cases}$$

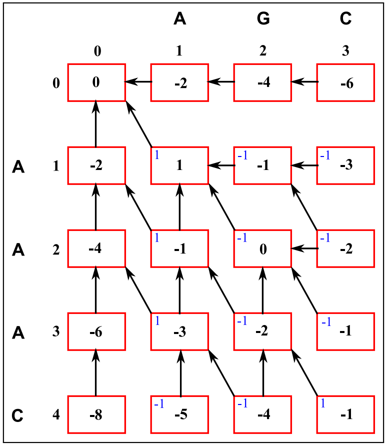
Based on this, we can have the following reformulation.

$$a[i,j] = \max \begin{cases} a[i,j-1] + g \\ a[i-1,j-1] + p(i,j) \\ a[i-1,j] + g \end{cases}$$

Here, g denotes the gap score and $p(i,j)$ denotes the match/mismatch score.

<u>Note</u>: An arrow is used to highlight which cell contributes to the maximum value.

Deriving similarity of sequences



Deriving optimal global alignment between AAAC and AGC (the corner of a cell (i, j) reflects whether $s[i] = t[j]$)

Finding the best global alignment score

The following dynamic programming approach (known as Needleman-Wunsch algorithm) can be used to compute the best alignment score (which is -1 here) for the previous example.

Input: Sequences s and t.
Output: Matrix a containing the similarities between s and t.

```
 1: $m \leftarrow |s|$                          // Length of $s$
 2: $n \leftarrow |t|$                          // Length of $t$
 3: **for** $i \leftarrow 0$ to $m$ **do**
 4:     $a[i,0] \leftarrow i \times g$          // Filling up the first column
 5: **end for**
 6: **for** $j \leftarrow 0$ to $n$ **do**
 7:     $a[0,j] \leftarrow j \times g$          // Filling up the first row
 8: **end for**
 9: **for** $i \leftarrow 1$ to $m$ **do**
10:     **for** $j \leftarrow 1$ to $n$ **do**
11:         $a[i,j] \leftarrow \max(a[i-1,j] + g, a[i-1,j-1] + p(i,j), a[i,j-1] + g)$
12:     **end for**
13: **end for**
14: return $a[m,n]$
```

Complexity analysis

Time complexity:
This algorithm consists of four *for* blocks. The first two blocks (initialization steps) consume time $O(m)$ and $O(n)$, respectively. The last two nested *for* blocks are used to fill the rest of the matrix. The number of operations performed depends essentially on the number of entries that must be computed, that is, the size of the matrix. Thus, we spend time $O(mn)$ in this part and the time complexity becomes

$$O(m) + O(n) + O(mn) = O(mn).$$

Space complexity:
As we need to fill the entries of the matrix 'a', the space complexity becomes $O(mn)$.

Improving the space complexity

It is possible to improve the space complexity from quadratic to linear and keep the same generality as follows.

Input: Sequences s and t.
Output: Matrix a containing the similarities between s and t.

```
 1: $m \leftarrow |s|$                                   // Length of $s$
 2: $n \leftarrow |t|$                                   // Length of $t$
 3: for $j \leftarrow 0$ to $n$ do
 4:     $a[j] \leftarrow j \times g$                     // Filling up the $j^{th}$ row
 5: end for
 6: for $i \leftarrow 1$ to $m$ do
 7:     $old \leftarrow a[0]$
 8:     $a[0] \leftarrow i \times g$
 9:     for $j \leftarrow 1$ to $n$ do
10:         $temp \leftarrow a[j]$
11:         $a[j] \leftarrow \max(a[j] + g, old + p(i,j), a[j-1] + g)$
12:         $old \leftarrow temp$
13:     end for
14: end for
15: return $a[m, n]$
```

Deriving the optimal global alignment

Input: Indices i, j, and the array a given by the previous algorithm.
Output: Alignments in align-s, align-t, and length in *len*.

```
 1: if $i = 0$ and $j = 0$ then
 2:     $len \leftarrow 0$
 3: else
 4:     if $i > 0$ and $a[i,j] = a[i-1,j] + g$ then
 5:         Recursive-call($i-1, j, len$)
 6:         $len \leftarrow len + 1$
 7:         Set align-$s[len] \leftarrow s[i]$ and align-$t[len] \leftarrow -$
 8:     else
 9:         if $i > 0$ and $j > 0$ and $a[i,j] = a[i-1,j-1] + p(i,j)$ then
10:             Recursive-call($i-1, j-1, len$)
11:             $len \leftarrow len + 1$
12:             Set align-$s[len] \leftarrow s[i]$ and align-$t[len] \leftarrow t[j]$
13:         else
14:             Recursive-call($i, j-1, len$)
15:             $len \leftarrow len + 1$
16:             Set align-$s[len] \leftarrow -$ and align-$t[len] \leftarrow t[j]$
17:         end if
18:     end if
19: end if
```

Deriving the optimal global alignment

The optimal global alignment for the example shown earlier can be derived using this algorithm as follows.

Step 1: Start from $a[m = 4, n = 3]$ and align C with C.
Step 2: Move diagonally to $a[m = 3, n = 2]$ and align A with G.
Step 3: Move up to $a[m = 2, n = 2]$ and align A with a gap $(-)$.
Step 4: Move diagonally to $a[m = 1, n = 1]$ and align A with A.
Step 5: Move diagonally to $a[m = 0, n = 0]$ and stop.

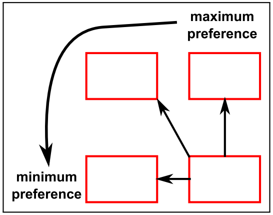
Thus, the final alignment becomes the following:

AAAC

A–GC

Deriving the optimal global alignment

Note that, many optimal alignments may exist for a given pair of sequences. The presented algorithm returns just one of them, giving preference to the edges leaving (i, j) in counterclockwise order as shown below.



So, when there is choice, a column with a gap in t has precedence over a column with two symbols, which in turn has precedence over a column with a gap in s. This can be changed by changing the order of the *if-else* blocks in the algorithm.

Complexity analysis

Time complexity:
This algorithm consumes time $O(len)$, where len is the size of the returned alignment, which is essentially $O(m+n)$.

Space complexity:
Given the already filled matrix a as input, the space complexity becomes $O(mn)$.

Alignment with gap penalty functions

Let us redefine a *gap* as a consecutive number of $k > 1$ spaces. The formation of such (consecutive) gaps with k spaces is more probable than k isolated spaces during mutations.

As of now, no distinction has been made between the consecutive and isolated gaps. The gaps were penalized in the previous cases through a linear function given by

$$f(k) = kg,$$

where g is the score associated with a single space and k is the number of spaces.

We introduce an algorithm that computes similarities with respect to general gap penalty functions that consider non-additive scores.

Alignment with gap penalty functions

In this algorithm, we cannot break an alignment in two parts and expect the total score to be the sum of the partial scores. However, score additivity is still valid if we break the alignment in block boundaries.

Every alignment can be uniquely decomposed into a number of consecutive blocks. There can be three kinds of blocks as listed below.

1. Two aligned characters from the alphabet set.
2. A maximal series of consecutive characters in t aligned with spaces in s.
3. A maximal series of consecutive characters in s aligned with spaces in t.

Alignment with gap penalty functions

Consider the following global alignment between the pair of sequences $\mathrm{AACAATTCCGACTAC}$ and $\mathrm{ACTACCTCGC}$.

AAC——AATTCCGACTAC

ACTACCT————CGC—-

Now, consider the same alignment shown block by block as follows.

```
A | A | C | –  –  – | A | A  T  T  C  C  G | A | C | T | A  C
A | C | T | A  C  C | T | –  –  –  –  –  – | C | G | C | –  –
```

In the former case, the scoring of an alignment is done at the column level, but in the latter one scoring is done at the block level.

Alignment with gap penalty functions

To compare sequence s of length m to sequence t of length n, we use three arrays of size $(m+1) \times (n+1)$, one for each type of ending block. Array a is used for alignments ending in character-character blocks; b is used for alignments ending in spaces in s and c is used for alignments ending with spaces in t.

The initialization is done as follows:

$$a[0,0] = 0$$

$$b[0,j] = f(j)$$

$$c[i,0] = f(i)$$

Alignment with gap penalty functions

The following recurrence relations are used for updating the cells:

$$a[i,j] = p(i,j) + \max \begin{cases} a[i-1,j-1] \\ b[i-1,j-1] \\ c[i-1,j-1] \end{cases}$$

$$b[i,j] = \max \begin{cases} a[i,j-k] + f(k), 1 \leq k \leq j \\ c[i,j-k] + f(k), 1 \leq k \leq j \end{cases}$$

$$c[i,j] = \max \begin{cases} a[i-k,j] + f(k), 1 \leq k \leq i \\ b[i-k,j] + f(k), 1 \leq k \leq i \end{cases}$$

Complexity analysis

Time complexity:
For computing $a[i,j]$, $b[i,j]$, and $c[i,j]$, we need to perform $(3+2j+2i)$ accesses. So, the total worst case time complexity becomes

$$\begin{aligned} & \sum_{i=1}^{m}\sum_{j=1}^{n}(3+2j+2i) \\ = & \sum_{i=1}^{m}(2ni+n^2+4n) \\ = & O(mn^2+m^2n). \end{aligned}$$

Introduction to Hidden Markov models

Processes are of two types – deterministic (e.g., rolling the SHOLAY coin) and stochastic (e.g., rolling a dice).

Definition

A stochastic process $X = (X_t : t \in I)$ on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ is said to process the Markov property if, $\forall A \in \mathcal{F}$ and $s, t \in I$, $s < t$, we have

$$\mathbb{P}(X_t \in A | \mathcal{F}_s) = \mathbb{P}(X_t \in A | \sigma(X_s)),$$

where $\{\mathcal{F}\}_{t \in I}$ is the natural filtration.

If the process takes discrete values and is indexed by a discrete time, this can be reformulated as follows.

$$\mathbb{P}(X_n = x_n | X_{n-1} = x_{n-1} \cdots X_0 = x_0) = \mathbb{P}(X_n = x_n | X_{n-1} = x_{n-1}).$$

Alignment using hidden Markov models

We can prepare a matrix of transitional probabilities (where an entry at cell (i, j) denotes the probability of occurrence of j immediately after i), from the given sequences. E.g., consider the sequence $\mathrm{AAGGAATTAGC}$ and the corresponding transitional probabilities as shown below.

	–	**A**	**T**	**C**	**G**
–	–	1	0	0	0
A	0	2/5	1/5	0	2/5
T	0	1/2	1/2	0	0
C	1	0	0	0	0
G	0	1/3	0	1/3	1/3

The alignment can be generated from these transition probabilities. We use the relation $\arg\max_{i \in \{A,T,C,G\}} P(X_n = i | X_{n-1} = x_{n-1})$ to derive the aligned sequence.

Basics

In a semi-global comparison, we score alignments ignoring some of the end gaps (that appear before the first or after the last character) in the sequences. With a slight modification to the already presented algorithms, we can control the penalty associated with end gaps.

The end gaps are welcome because they might provide more acceptable alignments. E.g., consider the sequences AGCACTTGGATTCTCGG and CAGCGTGG, and their following two possible alignments.

```
-AGCACTTGGATTCTCGG    AGCA-CTTGGATTCTCGG
CAGC-----G-T----GG    --CAGCGTGG--------
```

[Match = 7, Mismatch = 0, Gap = 11]   [Match = 6, Mismatch = 1, Gap = 11]

Basics

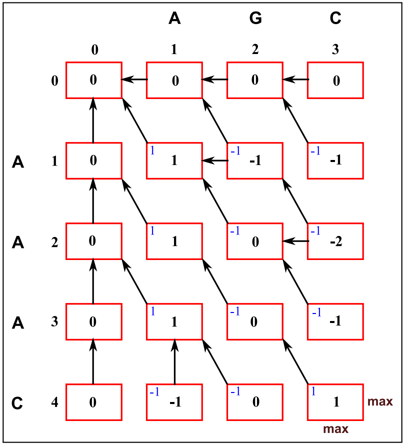
Using the already adopted scoring scheme, the first alignment turns out to be better, however, the second one appears to be more appropriate due to its *continuity*. This continuity can be effectively quantified based on the count of end gaps.

To obtain the score of the optimal alignment between s and t without penalizing the gaps after the end of s (final gaps), all we need to do is to find the best similarity between s and a prefix of t. In the previous algorithms, the entry (i, j) of matrix a contains the similarity between $s[1 \dots i]$ and $t[1 \dots j]$. Therefore, it suffices to take the maximum value in the last row (or column) of the array, i.e.,

$$\mathcal{S}(s, t) = \max_{j=1}^{n} a[m, j].$$

<u>Note</u>: Here $\mathcal{S}(s, t)$ indicates the similarity score ignoring the end gaps.

Deriving similarity of sequences



Deriving optimal semi-global alignment between AAAC and AGC by ignoring the end spaces at the beginning and end

Deriving the best semi-global alignment

The maximum similarity scores over the rows (or columns) basically gives the score of the best alignment. To recover the alignment itself, we proceed just as in the previous algorithms, but starting at (k, m) (or (m, k) for columns) where k is such that $\mathcal{S}(s, t) = a[k, m]$.

However, the initializations will be different based on the different versions of the same problem as follows.

Where gaps are not charged	**Action**
Beginning of first sequence	Initialize first row with zeros
End of first sequence	Look for maximum in last row
Beginning of second sequence	Initialize first column with zeros
End of second sequence	Look for maximum in last column

Basics

A local alignment between s and t is an alignment between a substring of s and a substring of t.

To find out the highest scoring local alignments between two sequences, we use an $(m+1) \times (n+1)$ array as used earlier for obtaining the global alignment.

But here the interpretation of the array values is different. Each entry (i, j) will hold the highest score of an alignment between a suffix of $s[1 \dots i]$ and a suffix of $t[1 \dots j]$. The first row and the first column are initialized with zeros.

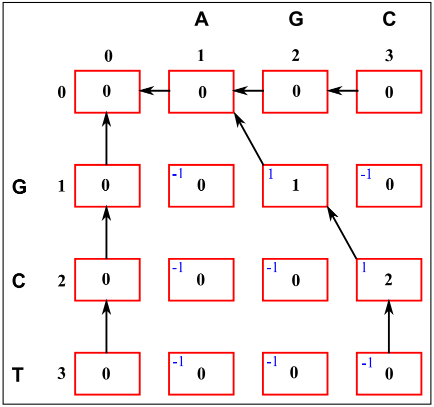
Basic constructions

Following initialization, the array can be filled in the usual way, with $a[i, j]$ depending on the value of three previously computed entries as shown below.

$$a[i, j] = \max \begin{cases} a[i, j-1] + g \\ a[i-1, j-1] + p(i, j) \\ a[i-1, j] + g \\ 0 \end{cases}$$

For any entry (i, j), there is always the alignment between the empty suffixes of $s[1 \ldots i]$ and $t[1 \ldots j]$, which has the score zero. Therefore, the array will have non-negative entries only.

Deriving similarity of sequences



Deriving optimal local alignment between the sequences AGC and GCT

Finding the best local alignment score

The following dynamic programming approach (known as Smith-Waterman algorithm) can be used for the previous example.

Input: Sequences s and t.

Output: Similarity between s and t.

```
1: $m \leftarrow |s|$                      // Length of $s$
2: $n \leftarrow |t|$                      // Length of $t$
3: for $i \leftarrow 0$ to $m$ do
4:     $a[i, 0] \leftarrow 0$              // Filling up the first column
5: end for
6: for $j \leftarrow 0$ to $n$ do
7:     $a[0, j] \leftarrow 0$              // Filling up the first row
8: end for
9: for $i \leftarrow 1$ to $m$ do
10:     for $j \leftarrow 1$ to $n$ do
11:         $a[i,j] \leftarrow \max(a[i-1,j]+g, a[i-1,j-1]+p(i,j), a[i,j-1]+g, 0)$
12:     end for
13: end for
14: return $\max_{\forall m,n} a[m, n]$
```

Deriving the optimal local alignment

The cell containing the best local alignment score (maximum value) is used as a starting point to get the optimal local alignment. The rest of the alignment is obtained by tracing back into the matrix as done before. The algorithm stops if either we have reached to an entry with the value zero or we have reached to an entry with no arrow going out.

The optimal local alignment for the example shown earlier can be derived using this algorithm as follows.

Step 1: Start from $a[m = 2, n = 3]$ and align C with C.
Step 2: Move diagonally to $a[m = 1, n = 2]$ and align G with G.
Step 3: Move diagonally to $a[m = 0, n = 1]$ and stop.

Thus, the final local alignment is at GC.

Complexity analysis

Time complexity:
The algorithm for finding the best local alignment score consumes time $O(mn)$ and the algorithm for deriving the optimal local alignment consumes $O(m + n)$.

Space complexity:
For both the above cases, space complexity becomes $O(mn)$.

Basic Local Alignment Search Tool (BLAST)

BLAST algorithms are used to search databases. BLAST can rapidly align and compare a query sequence with a database of sequences. Some of the variants of BLAST are listed below.

- BLASTn
- BLASTp
- BLASTx
- tBLASTn
- tBLASTx

BLAST increases the speed of alignment by decreasing the search space (number of comparisons). Specifically, instead of comparing every residue against each other, BLAST uses short `word` segments to create alignment `seeds`. BLAST also calculates the statistical significance for each sequence alignment result.

Basics

We are often required to align more than two sequences simultaneously in the best possible way. This refers to the problem of multiple sequence alignment.

Given a set of sequences over the same alphabet, a multiple alignment is obtained by inserting gaps in these sequences such that their sizes become the same. We generally place the extended sequences in a vertical list so that characters or gaps in the corresponding positions occupy the same column.

Such an example with protein sequences is shown below.

ADNMQPHLLL–

ADNMLR–LL–Y

ADNMK—-LLLY

–DNMPPVLHLY

Scoring the alignment of multiple sequences

As because scoring a multiple alignment is more complex than its pairwise counterpart, we therefore restrict ourselves to purely additive functions here, i.e., the alignment score is the sum of column scores.

We note the following two important requirements for deriving such a score.

1. The function must be independent of the order of arguments.
2. The function should reward the presence of many equal or strongly related characters and penalizes unrelated residues and gaps.

We use the sum-of-pairs (SP) function for this purpose.

Scoring the alignment of multiple sequences

The SP function is defined as the sum of pairwise scores of all pairs of symbols in the column.

For instance, consider the sixth column of the previously shown alignment of four sequences. The similarity score for this column comprising the character set $\{\mathrm{P},\ \mathrm{R},\ -,\ \mathrm{P}\}$ is given by

$$\text{SP-score}(\mathrm{P},\ \mathrm{R},\ -,\ \mathrm{P})$$

$$= \mathcal{S}(\mathrm{P},\ \mathrm{R}) + \mathcal{S}(\mathrm{P},\ -) + \mathcal{S}(\mathrm{P},\ \mathrm{P}) + \mathcal{S}(\mathrm{R},\ -) + \mathcal{S}(\mathrm{R},\ \mathrm{P}) + \mathcal{S}(-,\ \mathrm{P}),$$

where $\mathcal{S}(x, y)$ denotes the pairwise score for the pair of characters x and y.

Scoring the alignment of multiple sequences

To apply the SP-score, we need to define a value for $\mathcal{S}(-,-)$ which was not required earlier in pairwise alignment (global, local or semi-global). Conventionally, we take $\mathcal{S}(-,-) = 0$.

Note that, only when $\mathcal{S}(-,-) = 0$ we have the following interesting relation.

$$\text{SP-score}(\alpha) = \textstyle\sum_{i<j} \text{score}(\alpha_{ij}),$$

where α is denoting the multiple alignment, and α_{ij} is the pairwise alignment induced by α on the sequences s_i and s_j.

This is true because it reflects two ways of doing the same thing.

Deriving the best multiple alignment

Suppose, for simplicity, that we have k sequences, all of the same length n. We use a k-dimensional array a of length $n+1$ in each dimension to hold the optimal scores for multiple alignments of prefixes of the sequences.

After initializing with $a[0, \ldots, 0] \leftarrow 0$, we fill in this entire array by computing $a[i] \leftarrow \max_{b \neq 0}(a[i-b] + \text{SP-score}(Column(s, i, b)))$, where b ranges over all nonzero binary vectors of k elements. Here, $Column(s, i, b) = (c_j)_{1 \leq j \leq k}$ with $c_j = s_j[i_j]$, if $b_j = 1$, and $c_j = -$, otherwise.

Note: The cell $a[i_1, \ldots, i_k]$ holds the score of the optimal alignment involving $s_1[1 \ldots i_1], \ldots, s_k[1 \ldots i_k]$.

Complexity analysis

Time complexity:
This algorithm works on every cell of the array *a* to compute the values that consumes $O(n^k)$ time. Again, for each entry this computation depends on $2^k - 1$ entries, thereby requiring $O(2^k)$ time. Additionally, the algorithm uses SP-score for scoring the alignments by computing pairwise alignments. This consumes $O(k^2)$ time. Therefore, the total worst case complexity becomes $O(k^2 2^k n^k)$, where k is the number of sequences with length n.

Space complexity:
As we need to fill the entries of the matrix *a*, the space complexity becomes $O(n^k)$.

Improving the time complexity

It is possible to improve the time complexity by using an efficient heuristics. The heuristic is based on the relationship between a multiple alignment and its projections on two-sequence arrays.

The outline of the method is as follows. We have k sequences of length n_i, for $1 \leq i \leq k$, and we want to compute the optimal alignments according to the $\mathrm{SP\text{-}score}$. We will still use dynamic programming, but now we do not want to treat all cells, rather we will work on the cells that are *relevant* to optimal alignments, in some sense.

In a preprocessing step, we create (and use) conditions that will allow us to perform a test of relevance for arbitrary cells.

Pairwise projection of multiple alignments

Consider the following multiple sequence alignment.

ADNMQPHLLL–

ADNMLR–LL–Y

ADNMK—-LLLY

–DNMPPVLHLY

Suppose we take only the second and third one out of the above.

ADNMLR–LL–Y

ADNMK—-LLLY

By removing additional spaces, the induced pairwise alignment (projection) is obtained as follows.

ADNMLRLL–Y

ADNMK–LLLY

Relevance test

Theorem

A cell is relevant when each of its pairwise projections is part of an optimal alignment of the two sequences corresponding to the projection. Let α be an optimal alignment involving $s_1, \ldots, s_k$. If $SP\text{-}score(\alpha) \geq L$, then

$$score(\alpha_{ij}) \geq L_{ij},$$

where $L_{ij} = L - \sum_{x<y,(x,y)\neq(i,j)}(\mathcal{S}(s_x, s_y))$.

Relevance test

Proof.

The relation $\text{SP-score}(\alpha) \geq L$ can be written as

$$\begin{aligned}
& \sum_{x<y} (score(\alpha_{xy})) \geq L \\
\Rightarrow \quad & \sum_{x<y,(x,y)\neq(i,j)} (score(\alpha_{xy})) \geq L - score(\alpha_{ij}) \\
\Rightarrow \quad & \sum_{x<y,(x,y)\neq(i,j)} (\mathcal{S}(s_x, s_y)) \geq L - score(\alpha_{ij}) \\
\Rightarrow \quad & score(\alpha_{ij}) \geq L - \sum_{x<y,(x,y)\neq(i,j)} (\mathcal{S}(s_x, s_y)).
\end{aligned}$$

□

Heuristic alignment based on the relevance test

Initially, we create a matrix c in which each entry (i, j) contains the highest score of an alignment that includes the $cut(i, j)$. A pairwise alignment α contains $cut(i, j)$ when α can be divided into two subalignments, one aligning $s[1 \dots i]$ with $t[1 \dots j]$ and the other aligning the rest of s with the rest of t. To obtain the cut matrix $c = a + b$, we create a and b as follows

$$a[i, j] = \mathcal{S}(s[1 \dots i], t[1 \dots j])$$

$$b[i, j] = \mathcal{S}(s[i + 1 \dots m], t[j + 1 \dots n]).$$

We can identify the best alignments just by looking at c. We start with the cell at $(0, 0, \dots, 0)$, which is always relevant, and expand its influence to dependent relevant cells. Each one of these will in turn expand its influence, and so on, until we reach the final corner cell at $(n_1, \dots, n_k)$. Throughout this process, only relevant cells are analyzed.

Hands-on

1. BLASTn (Nucleotide BLAST) the reference genome sequence of SARS-CoV-2 using the NCBI database. BLAST is available at: `https://blast.ncbi.nlm.nih.gov/Blast.cgi`
2. Download the following paper and do the following:
   Li, H., Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics, 34(18), pp.3094-3100, 2018.
   - i) Get the implementation from: `https://github.com/lh3/minimap2`
   - ii) Apply it on some supplementary available at: `https://academic.oup.com/bioinformatics/article/34/18/3094/4994778`
   - iii) Could you identify some limitations of this implementation? How can you overcome that? Any suggestions?