# Computational Biology Project
# for the attention of Prof. Frank Neven, UHasselt

—

# Sequence Alignment : Theory, Algorithms and Practice in Python

Joiret Marc (UHasselt 1541822)

April 12, 2017

# 1 Optimal pairwise alignment

## 1.1 Dynamic Programming

We borrow from Jones and Pevzner [1], the following basic concepts of Bioinformatics algorithms.

An *algorithm* is a sequence of instructions that one must perform in order to solve a well-formulated problem. Problems are specified in terms of their inputs and their outputs, and the algorithm will be the method of translating the inputs into the outputs.

Some algorithms break a problem into smaller subproblems and use the solutions of the subproblems to construct the solution of the larger one. During this process, the number of subproblems may become very large, and some algorithms solve the same subproblem repeatedly, needlessly increasing the running time.

*Dynamic programming* organizes computations to avoid recomputing values that you already know, which can often save a lot of time.

### 1.1.1 The Fibonacci Example

To describe more precisely what is meant by dynamic programming, let us take a very simple example from Elkner, J. ([2], Chap. 12.5) to compute the Fibonacci sequence in Python code.
Recall that the Fibonacci sequence is the sequence of integers such that the last one is the sum of the two previous ones ($1, 1, 2, 3, 5, 8, 13, 21, 34, 55$ are the 10 first such integers). The $n^{ieth}$ Fibonacci integer $F(n)$ can be defined recursively by :

$$F(0) = 1 \tag{1}$$
$$F(1) = 1 \tag{2}$$
$$F(n) = F(n-2) + F(n-1) \tag{3}$$

A recursive algorithm to find the Fibonacci number $F(n)$ can easily be implemented in Python :

```
def fibonacci (n):
    if n == 0 or n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

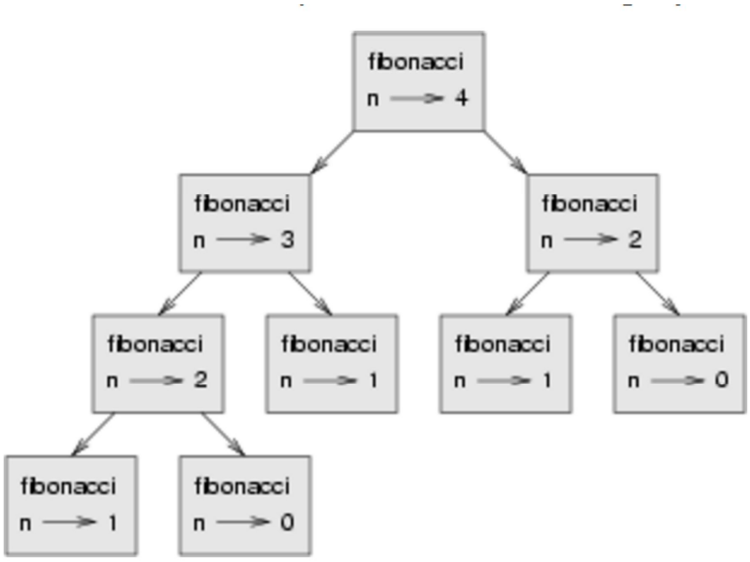


Figure 1: Recursive Fibonacci calls prove to be inefficient.

Such an algorithm will provide an output with a correct answer but the runtime increases very quickly with $n$.
A call graph is schematically described on Fig. 1.
A call graph shows a set of function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, fibonacci with n = 4 calls fibonacci with n = 3 and n = 2. In turn, fibonacci with n = 3 calls fibonacci with n = 2 and n = 1. And so on. We see that fibonacci(0) and fibonacci(1) are called five times. This is an inefficient solution to the problem, and it gets far worse as the argument gets bigger.
A good alternative solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a hint. Here is the implementation of fibonacci using hints:

```
previous = {0: 0, 1: 1}

def fibonacci(n):
    if n in previous:
        return previous[n]
    else:
        new_value = fibonacci(n-1) + fibonacci(n-2)
        previous[n] = new_value
        return new_value
```

The dictionary named `previous` keeps track of the Fibonacci numbers we already know. We start with only two pairs: 0 maps to 0; and 1 maps to 1. Whenever fibonacci is called, it checks the dictionary to determine if it contains the result. If it's there, the function can return immediately without making any more recursive calls. If not, it has to compute the new value. The new value is added to the dictionary before the function returns.

This is an example of dynamic programming.

With the first recursive algorithm, the runtime for $n = 25$ is 38.78 milliseconds on a laptop computer, while with the dynamic programming algorithm, the runtime was about 0.02 millisecond, i.e. 2000 times faster for $n = 25$.

## 1.2 Alignments

What is an *alignment* and how do we *score* the goodness of an alignment ? The alignment of the strings $\mathbf{S_1}$ (of $m$ characters) and $\mathbf{S_2}$ (of $n$ characters, with $m$ not necessary the same as $n$) is a two row matrix such that the first row contains the characters of $\mathbf{S_1}$ in order while the second row contains the characters of $\mathbf{S_2}$ in order, where spaces may be interspersed throughout the strings in different places. As a result, the characters in each string appear in order, though not necessarily in an adjacent way. No column of the alignment matrix contains spaces in both rows, so that the alignment may have at most $n+m$ columns.

Columns that contain the same letter in both rows are called *matches*.
Columns that contain different letters in both rows are called *mismatches*.
The columns of the alignment containing one space are called *indels*.
Columns containing a space in the top row are called *insertions*, while columns containing a space in the bottom row are called *deletions*. This is the adopted convention for insertions and deletions from the perspective of the sequence written at the top (the $S_1(m)$ sequence) which will also be written horizontally in the dynamic programming matrix (see below).
The number of matches plus the number of mismatches + the number of indels is equal to the length of alignment matrix and must be smaller than $n + m$. An example of an alignment is :

```
A T - G T T A T -
A T C G T - A - C
```

From the perspective of the sequence written at the top, the space in position 3 is an insertion (C has been inserted and the bottom sequence results), while the space in the 6th position in the bottom sequence is a deletion of the T in the top sequence.

### 1.2.1 Scoring Alignments

The minimal unit of alignment consists of one position from each of the two sequence being compared. We distinguish pairs with indels and pairs with two residues.

For pair of indels, we use the so called *affine gap model* : the score of a gap is computed as

$$G = g_0 + l \cdot g_e$$

where $g_0$ is the cost of gap opening and $g_e$ is the gap extension. A reasonable combination is $g_0 = -5, g_e = -2$, i.e., gap opening is penalized at least twice as much as gap extension. For a match of nucleotide, we might score +1 and a mismatch might be scored -3.
In contrast to nucleotides, different amino-acids mutate at different rates and so, for amino-acids, substitution matrices are used [3], like the PAM and BLOSUM matrices.
Considering an alphabet which contains the gap character, any two members $x, y$ of this alphabet have a score $s(x, y)$ attached. If we consider the strings

- $S_1 = S_1[1]S_1[2]\cdots S_1[m]$
- $S_2 = S_2[1]S_2[2]\cdots S_2[n]$

and denoting $S_1^{'}$ and $S_2^{'}$ their gap-containing versions (like on the two rows aligned above), then the score of an alignment of these two sequences of length $l$ is simply the sum of scores of pairs of characters :

$$\text{Score} = \sum_{i=1}^{l} s(S_1^{'}[i],\ S_2^{'}[i])$$

### 1.2.2 Number of Possible Alignments

Haubold et. al [3] writes the number of alignments using the following recursion with the boundary condition :

$$f(m,n) = f(m-1,n) + f(m-1,n-1) + f(m,n-1) \quad (4)$$
$$f(m,0) = f(0,n) = f(0,0) = 1 \quad (5)$$

The dynamic programming solution consists of evaluating (4) by computing first the number of possibilities of alignment for the shortest alignments. The runtime algorithm is O(m, n) with such a dynamic programming approach.

We start by building a matrix whose rows are the indexed characters of string $S_2$ and whose columns are the indexed characters of string of $S_1$. The matrix is initially filled with empty results. We initialize by filling the first row and the first column according to a rule that depends whether you carry out a global or a local or semi-global alignment: i.e the boundary condition 5. Now, we further fill the rows by applying the recursion formula (4). And we quickly fill the matrix completely.

## 1.3 Global Alignment (Needleman and Wunsch, 1970)

There are three types of alignments : global, local and semi-global (overlap). There are variations of the same underlying dynamic programming algorithm. A two dimensional table is built, which is known as a dynamic programming matrix[1]. This structure is subjected to 3 steps :

1. Initializing the dynamic programming matrix.
2. Filling in the dynamic programming matrix with scores of optimal partial alignments.
3. Traceback : extracting one or more alignments from the dynamic programming matrix, via a traceback pointer.

### 1.3.1 Scoring Scheme

A scoring scheme must be adopted.
We took in our example program :

- match : +4
- mismatch : -1
- gap extension : -2 (no gap opening penalty, the affine gap model has not been implemented)

It is worth emphasizing that gaps and mismatches should be penalized differently. The final result of the alignment algorithm may depend on the scoring scheme (particularly with completely unaligned sequence like AAAA and GGGGG for instance).
We adopt the same nomenclature as the one in [3] for substring, prefix, suffix and subsequence.

### 1.3.2 Initializing

In the global alignment, the boundary condition is not zero. The scores for gaps penalize gaps at the extremities of both sequences, i.e. the boundary condition (5) is adopted : the gaps are penalized at the extremities of a sequence proportionally to the length of the gap.

**Global Alignment Initialization :**

$$\begin{aligned} F(i,0) &= i \cdot g_e \text{ where} i \in [0, \cdots, n] \\ F(0,j) &= j \cdot g_e \text{ where} j \in [0, \cdots, m] \end{aligned} \tag{6}$$

[1] Recall our convention that $i$ indexes the second sequence $S_2[i]$, as rows. There are $n$ rows for a string $S_2$ of length $n$, while $j$ indexes the first sequence $S_1[j]$ as columns. There are $m$ columns for a string $S_1$ of length $m$. This way the resulting dynamic programming matrix is $n \times m$ ($n$ rows by $m$ columns matrix) and the string $S_1$ is displayed horizontally while string $S_2$ is displayed vertically. The insertion and deletion are considered from the perspective of $S_1$ which is the "top" sequence to be aligned and $S_2$ is the "bottom" sequence. Hence, a space '-' in the top sequence is an insertion, while a space '-' in the bottom sequence is a deletion.

### 1.3.3 Filling the Dynamic Matrix

**Global Alignment Dynamic Programming Matrix filling :**

$$F(i,j) = \max \begin{cases} F(i,j-1) + g_e & : \text{(left move)} \\ F(i-1,j-1) + s(S_1[j], S_2[i]) & : \text{(upper left move)} \\ F(i-1,j) + g_e & : \text{(top move)} \end{cases} \quad (7)$$

### 1.3.4 Traceback

Having obtained the optimal score (depending on the scoring scheme), the corresponding alignment must be extracted. This is done via a traceback pointer : it must be remembered which cell was the precursor of the current calculated value $F(i,j)$. If the precursor (from which of the case in 7 was the maximum) was on the left, we flag the current cell as "LEF"; if the precursor was on top, we flag the current cell as "TOP"; and "ULE", if from the upper left.

An example of global alignment is presented hereafter : The global alignment algorithm implemented in Python is provided in the file "AlignmentMarc.py". DNA strings inputs were :
S1 = 'ACCGTT' (horizontally disposed sequence in the matrix)
S2 = 'AGTTCA' (vertically disposed sequence in the matrix)

The scoring scheme is :

- match : +4
- mismatch : -1
- gap extension : -2 (no gap opening penalty, the affine gap model has not been implemented)

The dynamic programming matrix, traceback matrix and alignments results are displayed below.
It is important to note that the solution is not necessarily unique as a current calculated cell can have ties in the maximum search in the recursion 7. It cases of ties (up to three traceback status in a particular cell can arise), there is no way to distinguish the resulting co-optimal alignments under the model. Most implementations ignore this problem and return an arbitrarily chosen solution.

```
++++++++++++++++++++++++++++++++++++++++++++
---------Dynamic Programming Matrix   :-------
++++++++++++++++++++++++++++++++++++++++++++
 0 -1 -2 -3 -4 -5 -6

-1  4  2  0 -2 -4 -6

-2  2  3  1  4  2  0

-3  0  1  2  2  8  6

-4 -2 -1  0  1  6 12

-5 -4  2  3  1  4 10

-6 -1  0  1  2  2  8
```

```
++++++++++++++++++++++++++++++++++++++++++++
---------TRACEBACK Matrix   :-----------------
++++++++++++++++++++++++++++++++++++++++++++
0   0   0   0   0   0   0

0 ULE LEF LEF LEF LEF ULE

0 TOP ULE ULE ULE LEF LEF

0 TOP ULE ULE TOP ULE ULE

0 TOP ULE ULE ULE ULE ULE

0 TOP ULE ULE LEF TOP TOP

0 ULE TOP ULE ULE TOP TOP

++++++++++++++++++++++++++++++++++++++++++++
---------Pairwise GLOBAL alignment  :---------
++++++++++++++++++++++++++++++++++++++++++++
ACCGTT--
A--GTTCA
```

That is an optimum global alignment of length 8, with the highest score of 8, with 4 matches, 2 insertions (2 spaces at the top) and 2 deletions (2 spaces at the bottom).

The implemented algorithm in Python is provided in the file "`AlignMarc.py`", see function `DynProgMat` (lines $64 - 100$ in Python file). The function `GlobalPair(S1, S2, Scoring)` is used to print the alignments. The program tests start at line 362 and below.

## 1.4 Semi-Global Alignment or Overlap Alignment

Semi-global alignment are used in the context of the sequencing strategy known as shotgun sequencing. It is like a global alignment except that the leading or trailing gaps do not contribute to the score (gaps at either ends of the alignment are not penalized). This is simply done by initializing the first row and first column of the dynamic programming matrix to zero (8) and filling the matrix exactly as for the global alignment. The initializing and recursion rules write :

**Semi-Global Alignment Dynamic Programming Matrix initializing and filling :**

$$F(i,0) = 0 \tag{8}$$

$$F(0,j) = 0 \tag{9}$$

$$F(i,j) = \max \begin{cases} F(i,j-1) + g_e & : \text{ (left move)} \\ F(i-1,j-1) + s(S_1[j], S_2[i]) & : \text{ (upper left move)} \\ F(i-1,j) + g_e & : \text{ (top move)} \end{cases} \tag{10}$$

The traceback step starts from the maximum score in the last row or column and the left positions unaccounted for are filled with gaps. There might be more than one cooptimal starting point for traceback, while in global alignment there was only one starting point.

The implemented algorithm in Python is provided in the file "`AlignMarc.py`", see function `OverlapDynProgMat` (lines $174 - 216$ in Python file). The function `OverlapPair(S1, S2, Scoring)` is used to print the alignments. The program tests starts at line 362 and below.

The semi-global alignment with our previous scoring scheme is :

```
++++++++++++++++++++++++++++++++++++++++++++
---------SEMI-GLOBAL (overlap) alignment  :---
++++++++++++++++++++++++++++++++++++++++++++
ACCGTT--
--AGTTCA
```

## 1.5 Local Alignment (Smith and Waterman, 1981)

The local alignment is appropriate for detection of local regions of high similarity within two sequences, or for probing a long sequence with a short fragment, because no gap penalties are imposed ouside the similar regions. The modifications of the basic dynamic programming algorithm to find optimal local alignments is simply two things : first thing : adding zero in the first row and column at intializing. As a result, either sequence can slide along the other before alignment starts, without incurring any gap penalty against the residues it passes by. Second thing : a fourth condition is added to the three in the global alignment: end the region being aligned if other options degrade the score even more. So the rules for filling in the matrix are in local alignment :

**Local Alignment Dynamic Programming Matrix initializing and filling :**

$$F(i,0) = 0 \tag{11}$$

$$F(0,j) = 0 \tag{12}$$

$$F(i,j) = \max \begin{cases} F(i,j-1) + g_e & : \text{ (left move)} \\ F(i-1,j-1) + s(S_1[j], S_2[i]) & : \text{ (upper left move)} \\ F(i-1,j) + g_e & : \text{ (top move)} \\ 0 \end{cases} \tag{13}$$

The extra option in the last line of (13) for the maximum search ensures that there are no negative values in the matrix. The maximum alignment can begin and end anywhere in the matrix. The procedure is to identify the highest value in the matrix. This represents the end (3' end for nucleic acid or carboy-terminus for proteins) of the alignment. This position is not necessarily in the lower-right end corner of the matrix as it was for global alignment. The trace-back procedure begins at this highest-value position and proceeds diagonally up to the left until a cell is reach with a value of zero.
In contrast to global or semi-global alignments, the local alignment consists of pair of substrings from the input sequences.

The implemented algorithm in Python is provided in the file "`AlignMarc.py`", see function `LocalDynProgMat` (lines $270-314$ in Python file). The function `LocalPair(S1, S2, Scoring)` is used to print the alignments. The program tests starts at line 362 and below.
The local alignment with our previous scoring scheme is :

```
++++++++++++++++++++++++++++++++++++++++++++
---------Pairwise LOCAL alignment  :---------
++++++++++++++++++++++++++++++++++++++++++++
GTT
GTT
```

## 1.6 Alignment using BioPython

The BioPython Alignment function that was used was `pairwise2` along with the SeqIO function to read in sequence files in FASTA format. The BioPython detailed code is written below line 427 in `AlignMarc.py`.

Here we compared 2 homologous DNA sequences.
The carbon fixation by photosynthesis starts with an enzyme called Ribulose biphosphate carboxylase (RUBISCO) in the Calvin cycle which takes place in the chloroplast of plants. This RUBISCO enzyme is made of different sub-unit proteins. The smallest protein sub-unit is 180 amino-acid residues long. We want to compare the small RUBISCO subunit of *Vitis vinifera* (Wine Grape) with the one of *Arabidopsis thaliana* (Thale Cress). The transcript sequences of both proteins were queried on the NCBI (National Center for Biotechnology Information) database.
The gene coding for RUBISCO in *Vitis vinifera* is on chromosome 17 while it is on chromosome 1 for *Arabidopsis thaliana*. The *Vitis* gene transcript is 945 bp from 3 exons. For the *Arabidopsis* gene transcript, the transcript is 754 bp. In both cases, the coding sequence (CDS) is 542 bp. We want to examine the similarity of both sequences with the BioPython alignment function (Global and Local). The code is detailed between line $423 - 505$ in the file `AlignMarc.py`.

The first nucleotides from the $5'-$CDS region aligned are presented below (*Arabidopsis* is the top sequence and *Vitis* is the bottom sequence).
The score for the global alignment is 19 with the scoring scheme of $1, -1, -2, -1$ for match, mismatch, gap opening and gap extension respectively.

```
Coding Sequence Alignments :
GLOBAL :
ATGGCTTCCTCTATGCTCTCTTCCGCTACTATGGTTGC-----------CT-CTCCGGCTCAGGCCACT
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
ATGGCTTCCTCCATGGTCTCCTCCGCCACCGTGGCTACAATCAACCGTGCTACCCCGGCTCAGGCCAAC
  Score=19

LOCAL :
ATGGCTTCCTCTATGCTCTCTTCCGCTACTATGGTTGC-----------CT-CTCCGGCTCAGGCCACT
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
ATGGCTTCCTCCATGGTCTCCTCCGCCACCGTGGCTACAATCAACCGTGCTACCCCGGCTCAGGCCAAC
  Score=76
```

# 2 Data structures

Suppose you have a text of length $n$ over some alphabet (the alphabet has 4 letters in the case of nucleic acids, 20 letters in the case of proteins). Suppose you are looking for a match of some string of length $m$ in this text of length $n$. With the Dynamic Programming alignment algorithms we saw previously, we could find an optimum exact local alignment of the string on the text in a runtime of $O(n, m)$. It is quadratic as it is the product of length $n$ by length $m$. The interesting question is : "Is there a method to obtain the match in a linear way ?" The answer is yes. We will see that we can even solve a matching problem in time proportional to $m$ only, i.e. the size of the pattern. The cost to pay is that we have to organize the data, i.e. the text of length $n$, into a specific structure that will take extra space memory. This is the purpose of Suffix Trees. Suffix Trees were invented by Peter Weiner, who also proposed a linear-time algorithm for their construction in 1973.

## 2.1 Suffix trees

**Suffix Tree :** it is a data structure for indexing a text such that it can be searched for a pattern in time proportional to the length of the pattern irrespective of the length of the text.

### 2.1.1 How suffix trees work ?

A suffix tree for a text of length $n$ can be constructed in $O(n)$ time, which leads immediately to a linear $O(n + m)$ algorithm for the pattern matching problem : construct the suffix tree for the text $\mathbf{t(n)}$, in $O(n)$ time, and then check whether the pattern $p(m)$ occurs in the tree, which requires $O(m)$ times once

the tree has been built. Note that for a very often queried database, like the "omic" databases, the effort to organize the data in a structured suffix trees is worth because it will assure that the queries are solved very fast (at least in a linear time $\sim m$).
The difficult part is the construction of the suffix tree in a linear time $O(n)$. There are at least three such algorithms in the literature but there are rather complicated (the 3 references are given in the book by Haubold [3]). We describe here how to build a suffix tree in a quadratic runtime, starting from a "keyword tree" (see [1], chap.9.5, p.322).

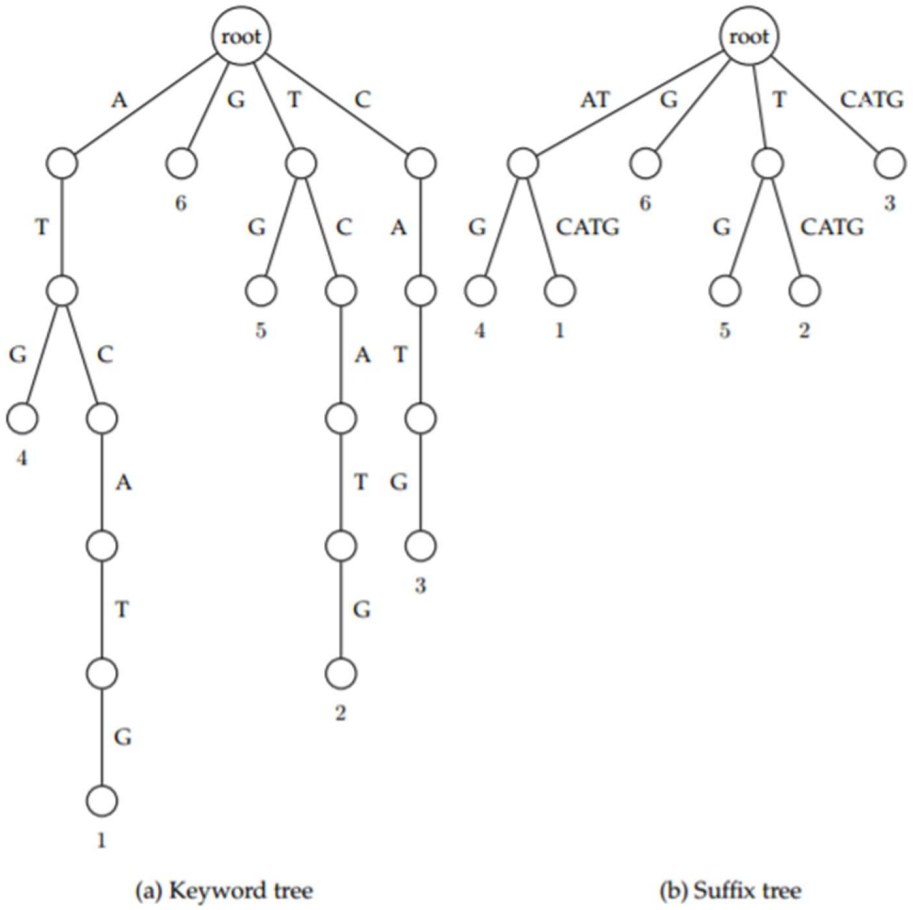


Figure 2: Constructing a suffix tree from a keyword tree.

Figure 2 shows the keyword tree (a) on the left panel for all six suffixes of the string "ATCATG" which are :

```
G        (position 6)
TG       (position 5)
ATG      (position 4)
CATG     (position 3)
TCATG    (position 2)
ATCATG   (position 1)
```

The suffix tree of a text can be obtained from the keyword tree of its suffixes by collapsing every path of nonbranching vertices into a single edge, as in the left panel of Figure 2.

The suffix tree for a text $t = t_1 \cdots t_n$ is a rooted labeled tree with $n$ leaves (numbered from 1 to $n$) satisfying the following conditions (the last letter should not appear anywhere and we fix it as an extra letter of the alphabet, e.g. the \$ sign).

- Each edge is labeled with a substring of the text
- Each internal vertex (except possibly the root) has at least two children
- Any two edges out of the same vertex start with a different letter
- Every suffix of text $\mathbf{t}$ is spelled out on a path from the root to some leaf

Suffix trees lead immediately to a fast algorithm for pattern matching. We define the threading of a pattern $\mathbf{p}$ through a suffix tree $\mathbf{T}$ as the matching of characters from $\mathbf{p}$ along the unique path in the tree until either all characters of $\mathbf{p}$ are matched, which is a complete threading, or until no more matches are possible, which is an incomplete threading.
The pattern matching algorithm is :

1. Compare first character in $\mathbf{p}$ : $p[1] \stackrel{?}{=}$ Edge labels$[1]$ from the root of $T$. If no match is found, stop (the pattern occurs nowhere in the text). If a match is found, follow the corresponding edge label comparing the next character in $\mathbf{p}$.
2. Every time a node is encountered, the correct edge for continuing the string comparison needs to be found.
3. Once a pattern is detected by the process, we know the text contains an exact match to the pattern $\mathbf{p}$.
4. The leaf labels in the subtree of the node in whose edge label the match ended, indicate the position of the pattern in the text.

Looking up the leaves of a subtree takes to a first approximation constant time. Given a suffix tree, the exact string matching problem is therefore solved in $O(m)$ because at most m comparisons are made between the pattern and the text.
Let us proceed the suffix tree in the right panel of Figure 2 (text = ATCATG) and try to find a pattern = TC.
The first letter in the pattern is T and matches the edge labeled T. So a match was found and we go on via the node (third node in level one).
The second letter of the pattern is C and matches with the first letter from the edge labeled CTAG. So a complete pattern has been detected and the position of this pattern is indicated by the leaf label of the edge where the match took place : the position is 2.
Indeed the pattern TC is to be found in the second position of the text ATCATG and there is only one such pattern in the text.
If you take AT as another pattern, the process is the following :

The first letter of the pattern matches the first edge labeled AT. The second letter as well before we reached the first node. So our match is completed at the node and there are 2 occurrences of the pattern because there are 2 leaf labels in the subtree below the node : position 4 and position 1. Indeed the pattern AT is found twice in the text at the right positions (1 and 4).

### 2.1.2 Two applications of suffix trees

The first obvious application of suffix trees is to rapidly find a string of length $m$ exactly matching a substring of a text of large length $n$, with $m \ll n$ and return the position of the match.
There are more than two applications of suffix trees but we limit ourselves in the following two.

**Detection of Repeated and Unique Substrings :** In a lot of eucaryotic genomes from different species, there are varying amount of repetitive DNA that are classified in 5 categories : transposon derived repeats (interspersed repeats with length between 100 and 6000 bp), processed pseudogenes, simple sequence repeats (e.g $A_n$ or $[GC]_n$, also known as microsatellites), segmental duplications (10 kb-300 kb chunks of DNA copied from one region to another), blocks of tandemly repeated sequences. It is of importance in genomics to detect exactly repeated sequence as well as unique substrings.
Any prefix of an internal node's path label is a repeat substring. The suffix tree depicted in the right panel of Figure 2 has two internal nodes with path label AT and T. These substrings together with the prefix A of AT are the only three repeats contained in the underlying text ATCATG. Indeed AT and A are both repeated twice (positions 4 and 1), while T is repeated twice (positions 5 and 2).

We define the string depth of an internal node as the length of its path label. String depths can be assigned to internal nodes in a single depth first traversal. By looking up the internal node with, for instance, the greatest string depth, we can rapidly locate the longest exact repeat in a string.

Moreover, trees only contains two kinds of nodes : internal nodes and leaves. Hence, a path label either ends at an internal node, in which case it is a repeat substring, or at a leaf, in which case it is a unique substring.

**Longest Common Substring Problem :** The longest common substring problem is motivated by elementary biological considerations. When comparing two or more coding DNA sequences, regions that are well conserved are often of particular biological interest. They might encode functionally important domains of the corresponding protein or point to novel regulatory elements. They represent also suitable sites for PCR-primers. There are many occasions motivating the wish to determine the longest substring shared by all members of a set of strings (Generalized Suffix Trees).

To find the longest common substring between two sequences, we need to find the lowest common ancestor of leaves containing a suffix from the first and a suffix from the second. One way to do this is to note for each internal node, the string identifier of the leaves in its subtree. Those nodes that are referring to both strings end at a path that is labeled by a substring common to both sequences. All we need to do is search for the internal node referring both sequences that has the greatest string depth.

In the next subsection, we address both problems : the identification and number of repeats in a DNA sequence and the longest common substring between two sequences.

Are there some repeated nucleotide dimers in the mitochondrial DNA of rats ?

The Longest Common Substring Problem supports the phylogenic relation between the Blue Whale, the Hippopotamus and the Cow.

## 2.2 Python Library implementing suffix trees and example program

A Suffix Tree library was imported in Python from `https://github.com/ptrus/suffix-trees` via the cmd line : `pip install suffix_trees`.

### 2.2.1 Detection of Repeated Substring

To find which nucleotide dimer and the number of its repeat in a DNA sequence we used Python code lines 9 to 20 in the file `SuffixTreeMarc.py`.

In the 258 bp mtDNA coding for a mRNA of a protein product in the mitochondrion of the rat *Rattus norvegicus*, we found that `CC` was the most frequently encountered dimer and was repeated 36 times.

### 2.2.2 Pairwise Longest Common Substring Problem

It is known in evolution biology that the Whale and the Hippopotamus have their latest common ancestor closer in the past than the latest common ancestor of the Hippopotamus and the Cow. This fact tends to prove that a far common ancestor of whales was probably a terrestrial animal and that the ancestors of whales were not always to be found in oceans. We will use the genomic NCBI (US National Center for Biotechnology Information) repository to compare the mitochondrial DNA of the 3 species and check for the pairwise similarities of their mtDNA sequences.

- Whale : *Balaenoptera musculus* mitochondrion complete genome is 16402 bp and has accession number `NC_001601.1` on NCBI.
- Hypo : *Hippopotamus amphibius* mitochondrion complete genome is 16407 bp and has accession number `NC_000889.1` on NCBI.
- Cow : *Bos taurus* mitochondrion complete genome is 16338 bp and has accession number `NC_006853.1` on NCBI.

The code is detailed between line $22 - 54$ in the file `SuffixTreeMarc.py` and includes the FASTA format file reading of the DNA sequences.

The results are the following :

```
LCS length for Whale and Hippo : 84
LCS for Whale and Hippo :
ACTATTCGCC AGAGTACTAC TAGCAACAGC TTAAAACTCA
AAGGACTTGG CGGTGCTTCA TACCCCTCTA GAGGAGCCTG TTCT

LCS length for Whale and Cow : 80
LCS for Whale and Cow :
GGTTCGTTTG TTCAACGATT AAAGTCCTAC GTGATCTGAG
TTCAGACCGG AGTAATCCAG GTCGGTTTCT ATCTATTACG

LCS length for Hippo and Cow : 62
LCS for Hippo and Cow :
ATAGCTAAGA CCCAAACTGG GATTAGATAC CCCACTATGC
TTAGCCCTAA ACACAGATAA TT
```

It can be seen that the Longest Common Substring (LCS) in mtDNA is 84 nucleotides between the Whale and the Hippo, larger than the 62 nucleotides shared in their mtDNA between the Cow and the Hippo. This supports what was already inferred from comparative anatomy and fossil records.

# 3 Fast Alignment

The suffix tree algorithm above, while fast, can only find exact rather than approximate, occurrences of a string in a database. If we are trying to find an approximate match to a gene of length 1 kbp in a database of size $10^{11}$, the quadratic programming algorithms may be too slow. With the exponential growth of Gene Banks, there is no other choice but to use fast heuristics similarity search algorithms as an alternative to quadratic sequence alignment algorithms.

A heuristic is an algorithm that will yield reasonable results, even if it is not provably optimal or lacks even a performance guarantee.

Many heuristics for fast database search in molecular biology use the same filtration idea. Filtration is based on the observation that a good alignment usually includes short identical of highly similar fragments. Thus, one can search for short exact matches, for example, by using a hash table or a suffix tree, and use these short matches as seeds for further analysis.

In 1985, the idea of filtration in computational molecular biology was used by David Lipman and Bill Pearson, in their FASTA algorithm. It was further developed in BLAST, now the dominant database search tool in molecular biology. BLAST stands for Basic Local Alignment Search Tool.

## 3.1 How BLAST works

The purpose is to find a local inexact match between a short *query* sequence and a large number of *subject* sequences that make up the text of the search. In this case we are looking for one or more local alignments between the query sequence and the database entries, like those repositoried in GenBank with billions of nucleotides or amino-acids sequences.
BLAST has two characteristics :

- The use of filtration or exclusion of unpromising areas of the alignment matrix from detailed searching. The central notion of the BLAST algorithm is the `MSP` : **Maximal Segment Pair**. This is a pair of segments (substrings) of identical length whose score cannot be improved by extension on either side.
- The use of Altschul-Dembo-Karlin statistics for estimating the statistical significance of found matches. The statistics rely on the distribution of extreme values, called the Gumbel distribution (extremum distribution). The idea is to determine the probability of observing a high scoring segment pair with score greater or equal to some threshold.

The BLAST algorithm has 3 steps :

1. **List** : Construct a list of high-scoring words. Given a protein query sequence, all substrings (or words) of a certain length $w$ that have a match with a score $\geq T$ (threshold) somewhere in the query are stored along with their match positions in the query. For instance, if the query entails '...VTALWGKVN...', words of length 3 from this query are hashed and write : VTA, TAL, ALW, LWG, ... Then a list of words of three amino-acids is built and are match tested with the hash table with a substitution matrix like BLOSUM62 or PAM120. The list of words to be tested is $20^3 = 8000$. These 8000 words will have some score that can be ranked. Only the ones above the threshold score T are retained to be member of the highest scored list. With PAM120, T=17, a list of 184 words contains approximately 50 entries for each residue in the query. A queried protein of length 100 would induce a word list with around 5000 entries.

   The preprocessing of a DNA sequence is simpler than that of proteins. Given a DNA query, it is hashed into words of length $w$ which is typically set to 12.

2. **Scan** : Solve the exact matching problem. If the database is stable, transform it into a suffix tree format and that would allow a rapid retrieval of all positions of word matches. The authors of BLAST chose the option of preprocessing the distinct query words into a keyword tree. The scan phase therefore takes $O(n + l + k) \approx O(n)$ time, where $n$ is the size of the database, $l$ the length of the word list, and $k$ the number of matches of members of the word list.

3. **Extend** : Matches returned from the scanning step are extended in either direction. This continues until the score of the MSP (Maximal Segment Pair) remains below a previous cutoff score $S$ for a number of residues. Scores are calculated with BLOSUM62 substitution matrix along with gap penalties. When the cutoff is reached, it is reported in the BLAST output.

The statistical significance of the BLAST output :

Some quantitative measure is needed of whether the alignments represent significant matches or whether they would be expected to occur by chance. For local assignments (BLAST searches), rigorous statistical tests have been developed. The scores can be considered as a random variable with a Gumbel distribution. This allows us to evaluate the likelihood that the highest score from a search (i.e. values at the right end tail of the distribution) occurred by chance. The expected number of HSP (High Scoring segment Pairs) having some score S (or better) by chance alone is defined by :

$$E = K\, m\, n\, e^{-\lambda S} \tag{14}$$

where $E$ refers to the expected value, which is the number of different alignments with score equivalent or better than $S$ that are expected to occur by chance in a database search. This provides an estimate of false positive from a BLAST search. The parameters $K$ and $\lambda$ are referred to as Karlin-Altschul statistics. The relation of $E$ to the p-value is that $p$ (the p-value) of finding an HSP with a given E value is :

$$p = 1 - e^{-E} \tag{15}$$

## 3.2 BioPython BLAST search

Starting from the file with the mitochondrial DNA sequences (mtDNA) of the Whale, Hippotamus and the Cow, we queried the GenBank database of NCBI for matches with other targets. The code is detailed between line $01 - 27$ in the file `BlastSearchMarc.py` which includes the FASTA format file reading of the DNA sequences and the BLAST running over the Internet : `NCBIWWW.qblast()` calls the online version of BLAST at the NCBI website.

The results are stored in an XML file : `MyBlastResult.xml` for later parsing.
We found more than 248 other target sequences in the GenBank database, in a runtime of less than 2 minutes, with the closer relatives of those animals. An extract of the BLAST output is reproduced below showing 2 matched examples with the Hippopotamus mtDNA which are the mtDNA from a dwarf hippo and from a rhino.

```
****Alignment****
sequence: gi|347801547|gb|JN632625.1| Hexaprotodon liberiensis isolate CYTO mitochondrion, complete genome
length: 16396
e value: 0.0
GTTAACGTAGCTCAAACACCCAAAGCGAGGCACTGAAAATGCCTAGATGGGCTCACCCAGCCCCGTAAACATACA...
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| || |||||||||||| |||...
GTTAACGTAGCTCAAACACCCAAAGCGAGGCACTGAAAATGCCTAGATGGGCTCGCCTAGCCCCGTAAACACACA...
****Alignment****
sequence: gi|229473110|gb|FJ905816.1| Dicerorhinus sumatrensis mitochondrion, complete genome
length: 16466
e value: 0.0
GTTAACGTAGCTCAAACACCCAAAGCGAGGCACTGAAAATGCCTAGATGGGCTCACCCAGCCCCGTAAACATACA...
||||| |||||| ||  ||| ||||| |||||||||||||||||||| | ||  ||| ||| || |||||| | |...
GTTAATGTAGCTTAATAACC-AAAGCAAGGCACTGAAAATGCCTAGACGAGCCTACC-AGCTCCATAAACACATA...
```

# 4 Multiple Sequence Alignment

Multiple sequence alignments are often carried out for known members of a protein family. This can be useful in functional genomic and proteomic. The purpose is to highlight the conserved sequence motifs to infer functionally important domains or even pinpoint individual amino acids which play a decisive role in the way a particular enzyme works.

## 4.1 How CLUSTAL-W works ?

CLUSTAL-W uses a heuristic algorithm. It proceeds by reducing the multiple alignment problem to a series of pairwise alignments. The order in which these pairwise alignments are merged into the multiple alignment is the key of the algorithm. The pairwise alignments of the input sequences are used to calculate pairwise distances (number of pairwise mismatches). These distances serve as input data for construction of a guide tree (cluster diagram), which determines the order in which sequences are added to the multiple alignment : more similar sequences are given more weight in the overall alignment scheme than dissimilar sequences.

## 4.2 BioPython and CLUSTAL-W

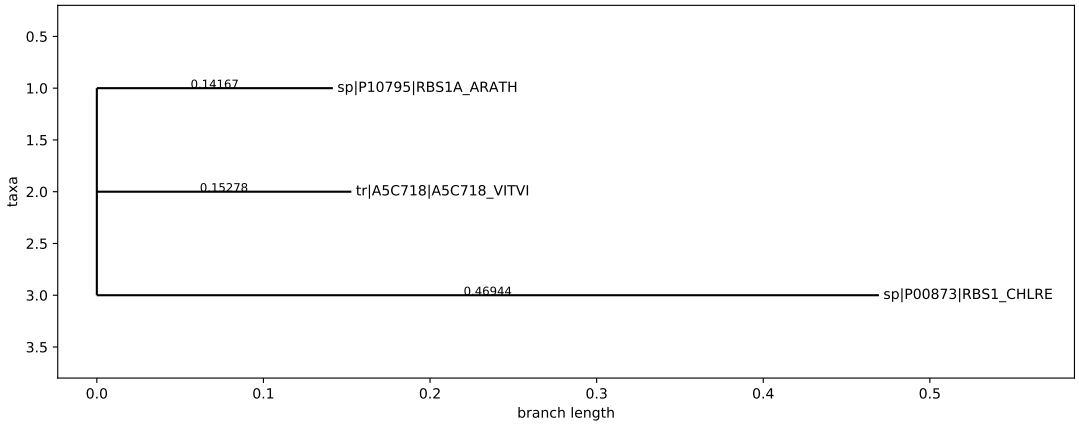


Figure 3: Inferred Phylogeneic Tree from the Rubisco protein small subunits of the 3 compared species (ARATH = *Arabidopsis thaliana*, VITVI = *Vitis vinifera*, CHLRE = *Chlamidomonas reinhardtii*).

The implemented example using BioPython addresses the similarity of proteins involved in photosynthesis of the three species :

- *Vitis vinifera*
- *Arabidopsis thaliana*
- *Chlamydomonas reinhardtii*

The protein sequences being compared are the small subunit of the enzyme Rubisco (first enzyme involved in carbon fixation by the photosynthetic organisms). This protein small subunit is coded in the DNA of the nucleus of eukaryotic cells with photosynthetic ability, is produced in the cytoplasm and then moves to the chloroplast where it is combined to larger subunits that are coded by chloroplastidial DNA (of some endosymbiotic probable phylogenic origin). It is an assumption of plant phylogeny that the gene coding for this small subunit initially came from a very primitive prokaryote cell that infected some common ancestor of all eukaryotic organisms which are able to fix carbon from photosynthesis. This question of this possible ancestral DNA horizontal transfer is not yet elucidated.

How well is the small subunit of the enzyme RUBISCO (Ribulose biphosphate carboxylase) conserved across unicellular eukaryotes like *Chlamydomonas* and higher multicellular eukaryotic plants like *Vitis* or *Arabidopsis* ?
We will try to answer the question by conducting a multiple alignment on the three sequences of the protein. The way to run CLUSTALw in BioPython is via a so called subprocess module : the `clustal w` executable programme was loaded from the internet in the `\Program file` directory, the path was located to BioPython so that Python could invoke Clustal w as a command line tool. The detailed BioPython code is provided in the attached file `MultipleAlignMarc.py`. The clustalw program produces two output files with extension `.asn` and `.dnd`. The former is used to print the multiple aligned sequences (see below) and the latter is used to display a phylogenic tree (see Figure 3).

```
CLUSTAL 2.1 multiple sequence alignment :

sp|P10795|RBS1A_ARATH      MASSMLSSATMV----ASPAQATMVAPFNGLKSSAAFPATRKANNDITSI
tr|A5C718|A5C718_VITVI     MASSMVSSATVATINRATPAQANMVAPFTGLKSLSTFPGTRKANTDITSV
sp|P00873|RBS1_CHLRE       MAAVIAKSSVSAAV--ARPARS-SVRPMAALK-----PAVKAAP------
                           **: : .*:. .    * **::  * *: .**     *..: *

sp|P10795|RBS1A_ARATH      TSNGGRVNCMQVWPPIGKKKFETLSYLPDLTDSELAKEVDYLIRNKWIPC
tr|A5C718|A5C718_VITVI     ASNGGRVRCMKVWPTEGLMKFETLSYLPPLNDEQLCKEVDYLLRMKWIPC
sp|P00873|RBS1_CHLRE       VAAPAQANQMMVWTPVNNKMFETFSYLPPLTDEQIAAQVDYIVANGWIPC
                           .:  .:.. * **.. .   ***:**** *.*.::. :****::   ****

sp|P10795|RBS1A_ARATH      VEFELEH-GFVYREHGNSPG-----YYDGRYWTMWKLPLFGCTDSAQVLK
tr|A5C718|A5C718_VITVI     LEFTKLY-PYPHREHNRSPG-----YYDGRYWTMWKLPMFGCTDSAQVIK
sp|P00873|RBS1_CHLRE       LEFAEADKAYVSNESAIRFGSVSCLYYDNRYWTMWKLPMFGCRDPMQVLR
                           :**      :  .*     *     ***.**********:*** *. **::

sp|P10795|RBS1A_ARATH      EVEECKKEYPNAFIRIIGFDNTRQVQCISFIAYKPPSFTG---------
tr|A5C718|A5C718_VITVI     EVQEARTAYPDAHIRIIGFDNNRQVQCIAFIAYKPS-------------
sp|P00873|RBS1_CHLRE       EIVACTKAFPDAYVRLVAFDNQKQVQIMGFLVQRPKTARDFQPANKRSV
                           *:  . . :*:*.:*::.*** :*** :.*:. :*
```

It is very likely that the segment of the protein in the domain `RYWTMWKLP` which shows a complete similarity in a sub-sequence of nine residues (very highly conserved as marked by the high number of contiguous $\star$) is bound to play a key functional role in the final RUBISCO enzyme assembly. It is also worth noticing that the *Vitis* and *Arabidopsis* Rubisco small subunits are much more alike than the *Chlamidomonas* one (see Figure 3).

# References


[1] Jones, N.C & Pevzner, P.A. (2004). *An Introduction to Bioinformatics Algorithms*. MIT Press Series on Computational Molecular Biology, USA.

[2] Elkner, J. et al. (2012). *How to Think Like a Computer Scientist*. `http://openbookproject.net/thinkcs/python/english2e/`.

[3] Haubold, B. & Wiehe, T. (2006). *Introduction to Computational Biology : An Evolutionary Approach*. Birkhauser, Basel, Switzerland.