T-COFFEE

Elena Montenegro, Nicolás Manosalva, Luis Cervera, Fernando Freire

January 5, 2019

Contents

1 Multiple Sequence Alignment: T-COFFEE

Authors: *Elena Montenegro, Nicolás Manosalva, Luis Cervera, Fernando Freire*

In this notebook we are going to explain a detailed description of T-COFFEE MSA algorithm, or more precisely, **the set of algorithms** that converge on T-COFFEE strategy to multiple sequence alignment.

We do so over a code implementation, not the T-COFFEE standard but an implementation that covers the basic aspects of the T-COFFEE approach.

Where necessary, we compare the T-COFFEE approach with the CLUSTALW one, in order to make more understandable the T-COFFEE method.

We are going to cover the topics in this order:

1. Pairwise alignment basis.
2. From pairs to sets.
3. Which order? Guide trees.
4. Putting all together: CLUSTAL way.
5. From CLUSTAL to T-COFFEE.
6. Beyond T-COFFEE.

Sample file

This is a sample file of protein sequences that we use in the across the explanation.

Script 1.0.1 (text)

```
%%writefile sample.fasta
>1aboA
NLFVALYDFVASGDNTLSITKGEKLRVLGYNHNGEWCEAQTKNGQGWVPS
NYITPVN
>1ycsB
KGVIYALWDYEPQNDDELPMKEGDCMTIIHREDEDEIEWWWARLNDKEGY
VPRNLLGLYP
>1pht
GYQYRALYDYKKEREEDIDLHLGDILTVNKGSLVALGFSDGQEARPEEIG
WLNGYNETTGERGDFPGTYVEYIGRKKISP
>1vie
DRVRKKSGAAWQGQIVGWYCTNLTPEGYAVESEAHPGSVQIYPVAALERI
N
>1ihvA
NFRVYYRDSRDPVWKGPAKLLWKGEGAVVIQDNSDIKVVPRRKAKIIRD
```

Output

```
Overwriting sample.fasta
```

1.1 Pairwise alignment basis

These type of algorithms are in the kernel of all MSA methods and participating in one or more steps.

Two approaches

1. Precise methods (dynamic programming):

- Global alignment (Needleman-Wunsch)
- Local alignment (Smith-Waterman)

2. Heuristic methods:

- BLAST
- ...

Optimization criteria: All searching the highest alignment score or the lowest penalty score
Scoring

1. Amino acid or base independent:

- fixed match, no match, gap init, gap continuation, gap close
- implemented as parameter variables

2. Amino acid or base dependent:

- match, no match based on biological concepts:
- BLOSUM, PAM, ... for amino acid
- JC, F81, K80, ... for bases
- implemented as matrices or dictionaries (hashes)
- no biological basics to score gaps
  - affine gaps
  - gap initiation
  - gap continuation
  - other functions
  - penalties based on gap position
  - CLUSTAL scores to 0 final gap sequences
  - increase the complexity of algorithms (memory and CPU)

A common set of parameters of pairwise align software.

```
"""
        mode (str): Computation mode:
            'GLOBAL'           Global Alignment
            'LOCAL'            Local Alignment
            'LONG_SUBSTRING'   Obtain long common substring
        score_match (int): Score of match characters.
        score_no_match (int): Score of no match characters.
        score_gap_ini (int): Score of gap init.
        score_gap_cont (int): Score of gap continuation.
        subst_matrix (dict: tuples, float): Substitution matrix
"""
```

Some gap complexities

```
"""
 For each cell, which coordinates are (i, j, ini_gap), calc the maximum score path from
        three alternative displacements:

        1) To (i + 1, j + 1, 1), that is, matching or no matching the seq0(i) and seq1(i) characters.
        This is a diagonal displacement.
        2) To (i, j + 1, 0), that is, seting a gap in seq0 and advance seq1. Horizontal displacement.
        3) To (i + 1, j, 0), that is, seting a gap in seq1 and advance seq0. Vertical displacement.
"""
```

1.1.1 Pairwise software

From the class below we'll take the global and local alignment methods that our implementation of t-coffee will use.

Script 1.1.1 (python)

```python
"""This module shows alternative recursive implementations of global sequence alignments:
    Global alignment (Needleman-Wunsch based)
    Local (Smith-Waterman based)
    Finding of the longest common substring.
Todo:
    * Return all the solutions of the alignments. Now it only returns one solution
    * Control of errors
    * Implement multi-alignments
    * Implement heuristic algorithms
"""
import time
import sys

MIN = -sys.maxsize - 1
COMPAC = 100000
"""int: Constant to compact max score."""
SCORE_MATCH = 2
"""int: Default match score."""
SCORE_NO_MATCH = -3
"""int: Default no match score."""
SCORE_GAP_INI = -10
"""int: Default gap ini in affine gap penalty."""
SCORE_GAP_CONT = -2
"""int: Default gap continuation in affine gap penalty"""
DEFAULT_SUBST_MATRIX = {('A', 'A'): 0, ('A', 'C'): 1, ('A', 'G'): 1, ('A', 'T'): 1, ('C',
 ↪  'A'): 1, ('C', 'C'): 0, ('C', 'G'): 1, ('C', 'T'): 1, ('G', 'A'): 1, ('G', 'C'): 1,
 ↪  ('G', 'G'): 0, ('G', 'T'): 1, ('T', 'A'): 1, ('T', 'C'): 1, ('T', 'G'): 1, ('T', 'T'): 0}
"""dict: Default substitution matrix (for "ACGT" common nucleotide alphabet)"""

sys.setrecursionlimit(5000)

class AlignSequences:
    """Recursive implementation of global, local and long substring alignments methods.

    Attributes:
```

```
        sequences (list of str): Contains the wo sequences to align. The first
            one (index 0) is the query sequence (BLAST concept) or bottom sequence on
    ↪  alignment prints
            or vertical sequence in the common graphical representation of score matrix.
        len_seq0 (int): Sequence 0 length.
        len_seq1 (int): Sequence 1 length.
        mode (str): Computation mode:
            'GLOBAL'           Global Alignment
            'LOCAL'            Local Alignment
            'LONG_SUBSTRING'   Obtain long common substring
        score_match (int): Score of match characters.
        score_no_match (int): Score of no match characters.
        score_gap_ini (int): Score of gap init.
        score_gap_cont (int): Score of gap continuation.
        score (int): Score of last computed alignment.
        gaps (int): Number of gaps of the last computed alignment.
        matches (int): Number of matches of the last computed alignment.
        unmatches (int): Number of unmatches of the last computed alignment.
        align_seq0 (str): Sequence 0 with the gaps necessary for the alignment.
        align_seq1 (str): Sequence 1 with the gaps necessary for the alignment.
        matching (str): Printable line with the align relations ('|, '.', ' ') between
            both align_seq, necessary for printing the alignment.
        ini_time (int): Initial time of computation, for profiling purposes
        finish_time (int): Final time of computation, for profiling purposes
        score_store (dict of tuple int): Store of scores, for each calculated cell with
    ↪  tuple (i,j,g)
            where i is the coordinate of the bottom sequence, j the coordinate of the top
    ↪  sequence
            and g has the value 1 if the cell is a gap init cell and 0 if it's a gap
    ↪  continuation.
            For a explanation of calculared cell see align method.
        matches_store (dict of tuple int): Store of the number of matches in the calculated
    ↪  cell
        gaps_store (dict of tuple int): Store of the number of gaps in the calculated cell
        max_score_index (tuple of int): Cell coordinate tuple of the cell with the maximun
    ↪  score
        max_score (int): maximum computed score
        forward_arrow (dict of str): Store of the optimal displacements accomplished at a
    ↪  cell
            to guarantee an optimal score: 'v' vertical (down), 'h' horizontal (rigth),
            'd' diagonal.
        stacks (list of list of str): Stacks of sequences related to principal sequences in
    ↪  a msa
        stacks_indexes (list of str): Indexes of the sequences of stack relatives to
    ↪  original sequences
        stacks_refs (list of dict): References of the char in sequence os stack relatives to
        char positions on original sequences
        subst_matrix (dict: tuples, float): Substitution matrix
        matrix_mode (str): If "SUBST" it's a substitution matrix, if not it's a weight matrix
            with the keys
                i = position of first sequence in stack
                j = position of second sequence on stack
```

```
                pos_i = coordinate of char on first sequence
                pos_j = coordinate of char on second sequence
            and the value is the weight to score this position
            if not match, the score is 0.
    """

    def __init__(self, sequences, mode="ALIGN", score_match=SCORE_MATCH,
    ↪  score_no_match=SCORE_NO_MATCH,\
                score_gap_ini=SCORE_GAP_INI, score_gap_cont=SCORE_GAP_CONT, subst_matrix={}
):
        """Init parameters of alignment"""
        self.set_sequences(sequences)
        self.set_stacks()
        self.len_seq0 = len(self.sequences[0])
        self.len_seq1 = len(self.sequences[1])
        self.init_stores()
        self.set_scores(score_match, score_no_match, score_gap_ini, score_gap_cont)
        self.set_mode(mode)
        self.score = 0
        self.matches = 0
        self.unmatches = 0
        self.gaps = 0
        self.align_seq0 = ""
        self.align_seq1 = ""
        self.matching = ""
        self.ini_time = 0
        self.finish_time = 0
        self.set_subst_matrix(subst_matrix)
        self.set_matrix_mode()

    def init_stores(self):
        """Init dictionary that store temp data of the alignment"""
        self.score_store = {}
        self.matches_store = {}
        self.gaps_store = {}
        self.max_score_index = (0, 0, 0)
        self.max_score = 0
        self.forward_arrow = {}

    def set_sequences(self, sequences):
        """Update the target sequences of the alignment"""
        self.sequences = sequences

    def set_stacks(self, stack_0=[], stack_1=[],\
                   stack_0_indexes=[], stack_1_indexes=[], stack_0_refs=[], stack_1_refs=[]):
        """Update the stacks for msa"""
        self.stacks = [stack_0, stack_1]
        self.stacks_indexes = [stack_0_indexes, stack_1_indexes]
        self.stacks_refs = [stack_0_refs, stack_1_refs]

    def set_matrix_mode(self, mode="SUBST"):
        """Update matrix mode"""
```

```
127         self.matrix_mode = mode
128 
129     def set_subst_matrix(self, subst_matrix={}):
130         """Update the score matrix"""
131         self.subst_matrix = subst_matrix
132 
133     def set_scores(self, score_match=SCORE_MATCH, score_no_match=SCORE_NO_MATCH,\
134                    score_gap_ini=SCORE_GAP_INI, score_gap_cont=SCORE_GAP_CONT):
135         """Update the weigth scores of the alignment"""
136         self.score_match = score_match
137         self.score_no_match = score_no_match
138         self.score_gap_ini = score_gap_ini
139         self.score_gap_cont = score_gap_cont
140 
141     def set_mode(self, mode="ALIGN"):
142         """Set computation mode"""
143         self.mode = mode
144 
145     def forward_track(self, index):
146         """Calc alignments in forward direction.
147 
148             The alignment strings are calculated from init cell (0,0) in global
149             alignments or maximum score cell in local alignments.
150 
151             In local mode it's necessary to extend the alignments (local) to the total
    ↪ length of
152             the sequences to show the location of the alignment, and in order to compare with
153             BioPython outputs.
154 
155             Args:
156                 index (tuple of int): Cell coordinates of the starting cell
157 
158             Returns:
159                 string: align sequence 0 (bottom) for printing purposes
160                 string: align sequence 1 (top) for printing purposes
161                 tuple of int: Coordinates of the last cell
162         """
163         ret_align_seq0, ret_align_seq1 = "", ""
164         (i, j, gap_ini) = index
165         ret_final_pos = (self.len_seq0, self.len_seq1)
166         while i < self.len_seq0 or j < self.len_seq1:
167             if self.mode == "LOCAL" and self.score_store[(i, j, gap_ini)] == 0:
168                 ret_final_pos = (i, j)
169                 break
170             arrow = self.forward_arrow[(i, j, gap_ini)]
171             if arrow == "d":
172                 ret_align_seq0 += self.sequences[0][i]
173                 ret_align_seq1 += self.sequences[1][j]
174                 i, j, gap_ini = i + 1, j + 1, 1
175             elif arrow == "h":
176                 ret_align_seq0 += "-"
177                 ret_align_seq1 += self.sequences[1][j]
```

```
                i, j, gap_ini = i , j + 1, 0
            elif arrow == "v":
                ret_align_seq0 += self.sequences[0][i]
                ret_align_seq1 += "-"
                i, j, gap_ini = i + 1, j, 0
        #compute the complete align in local mode
        if self.mode == "LOCAL":
            ret_align_seq0 = self.sequences[0][0:index[0]] +\
                             ret_align_seq0 + self.sequences[0][ret_final_pos[0]:]
            ret_align_seq1 = self.sequences[1][0:index[1]] +\
                             ret_align_seq1 + self.sequences[1][ret_final_pos[1]:]
            diff_pos_ini = index[1] - index[0]
            if diff_pos_ini > 0:
                ret_align_seq0 = '-' * diff_pos_ini + ret_align_seq0
            else:
                ret_align_seq1 = '-' * -diff_pos_ini + ret_align_seq1
            diff_len = len(ret_align_seq1) - len(ret_align_seq0)
            if diff_len > 0:
                ret_align_seq0 += '-' * diff_len
            else:
                ret_align_seq1 += '-' * -diff_len
        return ret_align_seq0, ret_align_seq1, ret_final_pos

    def calc_matching(self, align_seq0, align_seq1, ini_pos=(), final_pos=()):
        """Calc matching string

            The matching string is the string line to print between the top and
            bottom alignment strings. It contains the match (|), no match (.) and
            gap ( ) indicators.

            Args:
                align_seq0 (string): Bottom sequence
                align_seq1 (string): Top sequence
                ini_pos (tuple of int): Initial cell coordinates
                final_pos (tuple of int): Final cell coordinates

            Returns:
                string: Matching string

        """
        count = 0
        ret_matching = ""
        diff_pos_ini = ini_pos[1] - ini_pos[0]
        if diff_pos_ini > 0:
            delta_pos = diff_pos_ini
        else:
            delta_pos = 0
        for n, (i, j) in enumerate(zip(align_seq0, align_seq1)):
            if self.mode == "LOCAL" and not (n >= ini_pos[0] + delta_pos and n <
            ↪  final_pos[0] + delta_pos):
                ret_matching += ' '
            else:
```

```
                if i == j: ret_matching += '|'
                elif i != j and i != '-' and j != '-': ret_matching += '.'
                else: ret_matching += ' '
            count += 1
        return ret_matching

    def store(self, key, score, matches, gaps):
        """Store info related to a computed cell
        The maximum score is computed having into account the number of matches, if there are
        most than one solution. If the score are equal, the path with more matches is
    ↪ selected.
             Args:
                key (tuple of int): Cell coordinates
                score (int): Cell score
                matches (int): Cell matches
                gaps (int): Cell gaps
        """
        self.score_store[key] = score
        super_score = score * COMPAC + 10 * matches
        if super_score > self.max_score:
            self.max_score_index = key
            self.max_score = super_score
        self.matches_store[key] = matches
        self.gaps_store[key] = gaps

    def calc_score_binary(self, seq_0, seq_1, i, j, seq_0_index=0, seq_1_index=1, pos_0=0,
    ↪ pos_1=0):
        """Compute alignment scores for two sequences
        If there are a substitution matrix (actually dictionary) defined,
        the scores are computed from the dictionary.
            Args:
                seq_0 (int): Sequence 0
                seq_1 (int): Sequence 1
                i (int): Sequence 0 char index
                j (int): Sequence 1 char index
                seq_0_index (int): Sequence 0 index on original sequences (MSA)
                seq_1_index (int): Sequence 1 index on original sequences (MSA)
                pos_0 (int): Sequence 0 index on stack 0
                pos_1 (int): Sequence 1 index on stack 0
        """
        if self.subst_matrix:
            if self.matrix_mode == "SUBST":
                #print("PAIR", i, j, seq_0[i], seq_1[j])
                subst_matrix_index = (seq_0[i], seq_1[j])
                subst_matrix_index_swap = (seq_1[j], seq_0[i])
                if subst_matrix_index in self.subst_matrix:
                    matrix_score = self.subst_matrix[subst_matrix_index]
                elif subst_matrix_index_swap in self.subst_matrix:
                    matrix_score = self.subst_matrix[subst_matrix_index_swap]
            else: #weight matrix
                if pos_0 in self.stacks_refs and i in self.stacks_refs[pos_0]:
                    i_orig = self.stacks_refs[pos_0][i]
```

```
                else:
                    i_orig = i
                if pos_1 in self.stacks_refs and i in self.stacks_refs[pos_0]:
                    j_orig = self.stacks_refs[pos_1][j]
                else:
                    j_orig = j
                if i_orig in self.subst_matrix[seq_0_index][seq_1_index] and \
                    j_orig in self.subst_matrix[seq_0_index][seq_1_index][i_orig]:
                    matrix_score = self.subst_matrix[seq_0_index][seq_1_index][i_orig][j_ori
g]
                else:
                    matrix_score = 0
        # Gaps in almost one of the sequences. This case only arises in MSA
        # There is no matrix related entry. If matrix is a weight matrix we compute
        # as zero (as defined in T-Coffee)
        if seq_0[i] == "-" or  seq_1[j] == '-':
            inc_matches = 0
            if self.subst_matrix:
                if self.matrix_mode == "SUBST":
                    inc_score = self.score_gap_cont
                else:
                    inc_score = 0
            else:
                inc_score = self.score_gap_cont
        else:
            if seq_0[i] == seq_1[j]:
                if self.subst_matrix:
                    inc_score = matrix_score
                else:
                    inc_score = self.score_match
                inc_matches = 1
            else:
                if self.subst_matrix:
                    inc_score = matrix_score
                else:
                    inc_score = self.score_no_match
                inc_matches = 0

        return inc_score, inc_matches

    def calc_score(self, i, j):
        """Compute alignment scores.
        If there are stacks associated with the sequence, we compute the score weigthing the
        scores of the stacks (SOP: Score of Pairs). Stacks contains also the guiding
 ↪  sequences.
            Args:
                i (int): Sequence 0 index
                j (int): Sequence 1 index
        """
        if self.stacks == [[],[]]:
            return self.calc_score_binary(self.sequences[0], self.sequences[1], i, j, 0, 1,
             ↪  0, 0)
```

```
328         else:
329             computed_score = 0
330             computed_matches = 0
331             nvalues = 0
332             for pos_0, (seq_0, index_0) in enumerate(zip(self.stacks[0],
                ↪  self.stacks_indexes[0])):
333                 for pos_1, (seq_1, index_1) in enumerate(zip(self.stacks[1],
                    ↪  self.stacks_indexes[1])):
334                     score, matches = self.calc_score_binary(seq_0, seq_1, i, j, index_0,
                        ↪  index_1, pos_0, pos_1)
335                     computed_score += score
336                     computed_matches += matches
337                     nvalues += 1
338             ret_score = computed_score /  nvalues
339             ret_matches = computed_matches / nvalues
340             return ret_score, ret_matches
341 
342     def align(self, i=0, j=0, ini_gap=1):
343         """Recursive align of sequences
344         For each cell, which coordinates are (i, j, ini_gap), calc the maximum score path
    ↪  from
345         three alternative displacements:
346 
347         1) To (i + 1, j + 1, 1), that is, matching or no matching the seq0(i) and seq1(i)
    ↪  characters.
348         This is a diagonal displacement.
349         2) To (i, j + 1, 0), that is, seting a gap in seq0 and advance seq1. Horizontal
    ↪  displacement.
350         3) To (i + 1, j, 0), that is, seting a gap in seq1 and advance seq0. Vertical
    ↪  displacement.
351 
352         The scores of these displacements are calculated adding the score ot the target cells
353         (that are computed recursively) and the matrix, default of gap scores in each case.
354 
355         The score, matches, gaps and forward_arrow are stored at related dictionary entry
    ↪  based on
356         coordinates (i, j, ini_gap), all of them asociated to the maximum score of
357         the three possible paths starting from the cell, avoiding recomputation
358         of the cell if it's called from another recuersive path.
359 
360         Each cell has a third score coordinate, because a cell could be called from a cell
    ↪  with yet
361         has a gap (only from horizontal or vertical prior displacement) or from a cell with
    ↪  has a match/no match.
362         Then we need to store two scores, matches, gaps and forward_arrows related to the
    ↪  two possible
363         cell incarnations at coordinates (i, j, 0) and (i, j, 1).
364 
365         We store matches and gaps in order to have one aditional criterion to tiebreaker
366         if some of the scores are equal. We are using this aproach in local alignment
    ↪  computation. If two
367         scores are equal we choose the solucion with the greatest number of matches.
```

```
368
369         We store the displacement directions in forward_arrow dict to compute the alignment.
370         It's posible to avoid this, using only the score information, but we have let this
    ↪  aproach
371         as proof of concept and for clarity in the algorithm.
372
373         In this scenario we observe that the differences between the global,
374         local and long substring algorithms are minimal.
375
376         Local algorithm:
377
378             Starting from the global algorithm, which would be the most general,
379             the local algorithm only changes two aspects:
380                 1. Rejection of the roads with negative values of the score, equaling these
    ↪  values
381                 to 0, that is, not letting previous alignments of poor quality affect the
    ↪  final result.
382
383                 2. Use as cell of beginning of the alignment the one with the highest scores.
384                 In our implementation we also take into account the number of matches,
385                 as we have already mentioned.
386
387         Finally, but outside the algorithm of alignment itself (at forward_track and
    ↪  matching methods)
388         it only remains to extend the alignment obtained to show its location within the
    ↪  chains to be aligned.
389
390         Search algorithm of the long common substring:
391
392             Modify the global algorithm in the following aspects:
393                 1. Only computes matches between characters or gaps in one or another
    ↪  initial sequence.
394
395          Args:
396                 i (int): Sequence 0 index
397                 j (int): Sequence 1 index
398                 ini_gap (int): 1 if gap initiation, 0 if gap continuation
399         """
400         score_diag, score_hor, score_ver = MIN, MIN, MIN
401         matches_diag, matches_hor, matches_ver = MIN, MIN, MIN
402         gaps_diag, gaps_hor, gaps_ver = MIN, MIN, MIN
403         #align and advance seq0 and seq1
404         #in long_substring mode only matches are processed
405         if i < self.len_seq0 and j < self.len_seq1 and\
406         (self.mode != "LONG_SUBSTRING" or self.sequences[0][i] == self.sequences[1][j]):
407             inc_score, inc_matches = self.calc_score(i, j)
408             key = (i + 1, j + 1, 1)
409             if key in self.score_store:
410                 score_diag, matches_diag, gaps_diag = \
411                 self.score_store[key] + inc_score, self.matches_store[key] + inc_matches,
                     ↪  self.gaps_store[key]
412             else:
```

```
413                 score, matches, gaps = self.align(i + 1, j + 1, 1)
414                 self.store(key, score, matches, gaps)
415                 score_diag, matches_diag, gaps_diag = score + inc_score, matches +
                     ↪  inc_matches, gaps
416             #don't align and gap in seq0 (advance seq1)
417             if j < self.len_seq1:
418                 gap_score = self.score_gap_cont + ini_gap * self.score_gap_ini
419                 key = (i, j + 1, 0)
420                 if key in self.score_store:
421                     score_hor, matches_hor, gaps_hor = self.score_store[key] + gap_score,\
422                     self.matches_store[key], self.gaps_store[key] + 1
423                 else:
424                     score, matches, gaps = self.align(i, j + 1, 0)
425                     self.store(key, score, matches, gaps)
426                     score_hor, matches_hor, gaps_hor = score + gap_score, matches, gaps + 1
427             #don't align and gap in seq1 (advance seq0)
428             if i < self.len_seq0:
429                 gap_score = self.score_gap_cont + ini_gap * self.score_gap_ini
430                 key = (i + 1, j, 0)
431                 if key in self.score_store:
432                     score_ver, matches_ver, gaps_ver =\
433                     self.score_store[key] + gap_score, self.matches_store[key],
                     ↪  self.gaps_store[key] + 1
434                 else:
435                     score, matches, gaps = self.align(i + 1, j, 0)
436                     self.store(key, score, matches, gaps)
437                     score_ver, matches_ver, gaps_ver = score + gap_score, matches, gaps + 1
438             #choose the high score path
439             matcher_diag, matcher_hor, matcher_ver = score_diag, score_hor, score_ver
440             if i < self.len_seq0 or j < self.len_seq1:
441                 if self.mode == "LOCAL" and matcher_diag < 0 and matcher_hor < 0 and matcher_ver
                     ↪  < 0:
442                     score_diag, score_hor, score_ver = 0, 0, 0
443                     #matcher_diag, matcher_hor, matcher_ver  = 0, 0, 0
444                 if matcher_diag > matcher_hor and matcher_diag > matcher_ver:
445                     ret_score, ret_matches, ret_gaps, ret_arrow =\
446                     score_diag, matches_diag, gaps_diag, "d"
447                 elif matcher_hor > matcher_ver:
448                     ret_score, ret_matches, ret_gaps, ret_arrow =\
449                     score_hor, matches_hor, gaps_hor, "h"
450                 else:
451                     ret_score, ret_matches, ret_gaps, ret_arrow =\
452                     score_ver, matches_ver, gaps_ver, "v"
453             else:
454                 ret_score, ret_matches, ret_gaps, ret_arrow =\
455                 0, 0, 0, ""
456             self.forward_arrow[(i, j, ini_gap)] = ret_arrow
457             if i == 0 and j == 0:
458                 self.store((0, 0, 1), ret_score, ret_matches, ret_gaps)
459                 if self.mode in ["GLOBAL", "LONG_SUBSTRING"]: self.max_score_index = (0, 0, 1)
460                 else: ret_score = self.max_score // COMPAC
461                 ret_matches = self.matches_store[self.max_score_index]
```

```
462             ret_gaps = self.gaps_store[self.max_score_index]
463 
464         return ret_score, ret_matches, ret_gaps
465 
466     def compute(self, mode="LOCAL", silent=False):
467         """Calc alignment
468             Args:
469                 mode (str): Type of algorithm (local, global or long substring)
470                 silent (bool): If true don't show alignment output
471         """
472         self.ini_time = time.time()
473         self.init_stores()
474         self.set_mode(mode)
475         self.score, self.matches, self.gaps = self.align()
476         self.align_seq0, self.align_seq1, final_pos = self.forward_track(self.max_score_inde
    x)
477         self.matching = self.calc_matching(self.align_seq0, self.align_seq1,
            ↪  self.max_score_index, final_pos)
478         self.unmatches = self.matching.count('.')
479         self.gaps = self.matching.count(' ')
480         self.finish_time = time.time()
481         if not silent:
482             self.view()
483 
484     def get_len_long_common_substring(self):
485         """Getter for the len of the common substring
486         That is equal to the number of matches of the alignment
487         """
488         return self.matches
489 
490     def get_long_common_substring(self):
491         """Returns the longest common substring
492         whitout alignment (positional) information
493         """
494         long_common_substring = ""
495         for (char, match_char) in zip(self.align_seq1, self.matching):
496             if match_char == '|':
497                 long_common_substring += char
498         return long_common_substring
499 
500     def view(self):
501         """Prints the alignment data"""
502         #unmatches = self.matching.count('.')
503         #gaps = self.matching.count(' ')
504         if self.matching:
505             gap_groups = self.matching.count('| ') + self.matching.count('. ') +
                ↪  self.matching[0].count(' ')
506         else:
507             gap_groups = 0
508         print(" ")
509         if self.mode == "LOCAL":
510             print("### AlignSequences. Local alignment (Smith-Waterman)")
```

```
        elif self.mode == "LONG_SUBSTRING":
            print("### AlignSequences. Long substring finder")
        else:
            print("### AlignSequences. Global alignment (Needleman-Wunsch)")
        if self.subst_matrix:
            print("\tUsing score matrix with matrix mode",self.matrix_mode)
        print(self.align_seq1)
        print(self.matching)
        print(self.align_seq0)
        print("\tScore:", self.score)
        print("\tSimilarity (wo gaps):", self.matches / (self.matches + self.unmatches))
        print("\tDistance (wo gaps):", self.unmatches / (self.matches + self.unmatches))
        print("\tDistance:", self.unmatches / (self.matches + self.unmatches + self.gaps))
        print("\tInit index:", self.max_score_index)
        print("\tMatches:", self.matches, " Unmatches:", self.unmatches, " Gaps:",
        ↪  self.gaps, " Gap groups:", gap_groups)
        #simple scoring verification todo: apply to matrix
        if not self.subst_matrix:
            print("\tScore verified:", self.matches * self.score_match + self.unmatches *
            ↪  self.score_no_match \
                + self.gaps * self.score_gap_cont + gap_groups * self.score_gap_ini)
        print("\tFinish. Execution milliseconds:", round((self.finish_time - self.ini_time)
        ↪  * 1000))
        print("\tScore Dictionary Size", len(list(self.score_store.keys())))

    def edit_distance(self, score_match=0, score_no_match=-1, score_gap_ini=0,
    ↪  score_gap_cont=-1):
        """Calculates an edit distance as requested in questions 1 and 3
        It's the same computation as a global alignment with -1 penalities applied to
        score_gap_cont and score_nomatch and 0 in score_match and score_gap_ini

            Args:
                score_match (int): Score of match characters.
                score_no_match (int): Score of no match characters.
                score_gap_ini (int): Score of gap init.
                score_gap_cont (int): Score of gap continuation.
        """
        self.set_scores(score_match, score_no_match, score_gap_ini, score_gap_cont)
        self.compute("GLOBAL", True)
        return abs(self.score)
```

Script 1.1.2 (python)

```
1
```

Tests

Script 1.1.3 (python)

```
import re
from Bio import pairwise2
from Bio.pairwise2 import format_alignment
```

```
from Bio.SubsMat import MatrixInfo

failed = 0
passed = 0
launched = 0

def test_alignment(number, s1, s2, verbose=False, tipus="local", matrix={},\
                           score_match=2, score_no_match=-3, score_gap_ini=-5,
                           ↪  score_gap_cont=-2):
    """Comparisons of global and local alignments between Biopython and AlignSequences
    ↪  implementation.

    Args:
        number (int): The number(identifier) of the test.
        s1 (str): Query string to align.
        s2 (str): Subject string to align
        verbose (bool): If True print outputs, default False
        tipus (str) : If 'local' the alignment is local (Smith), if 'global' Waterman.
        matrix (dict of int) : Substitution matrix
        score_match (int): Score of character match
        score_no_match (int): Score of character no match
        score_gap_ini (int): Score of gap initiation
        score_gap_cont (int): Score of gap continuation

    """
    global failed, passed, launched
    try:
        launched += 1
        align = AlignSequences([s1, s2])
        align.set_scores(score_match, score_no_match, score_gap_ini, score_gap_cont)
        if matrix != {}:
            method = getattr(pairwise2.align, tipus + 'ds')
            alignments = method(s2, s1, matrix,\
                                            score_gap_ini + score_gap_cont, score_gap_cont)
            align.set_subst_matrix(matrix)
        else:
            method = getattr(pairwise2.align, tipus + 'ms')
            alignments = method(s2, s1, score_match, score_no_match,\
                                             score_gap_ini + score_gap_cont, score_gap_cont)

        align.compute(tipus.upper(), silent=not verbose)
        m = re.match(r".*Score=([-1234567890]*)",
        ↪  format_alignment(*alignments[0]).replace("\n", ""))
        score = int(m.group(1))

        #search AlignSequences alignment in all possibles alignments fron Biopython
        found = False
        for a in alignments:
            if verbose:
                print()
                print("BioPython alignment:")
                print(format_alignment(*a))
```

```
            if align.align_seq0 == a[1] and align.align_seq1 == a[0]:
                    if not verbose: print(format_alignment(*a))
                    found = True
                    break
        assert(align.score == score)
        print ("Passed test %s: scores are equal '%s'" % (number, align.score ))
        assert(found)
        print ("Passed test %s: alignments are equal '%s'" % (number, align.align_seq0 ))
        passed += 1

    except AssertionError:
        print ("Failed test %s: alignments differ: \nBiopython:\n'%s'\nScore = %s \
        \nAlignSequences\n'%s'\nScore = %s"\
               % (number, alignments[0][1], score, align.align_seq0, align.score ))
        failed += 1
        exit(1)

prot1 = "GYQYRALYDYKKEREEDIDLHLGDILTVNKGSLVALGFSDGQEARPEEIGWLNGYNETTGERGDFPGTYVEYIGRKKISP"
prot2 = "NLFVALYDFVASGDNTLSITKGEKLRVLGYNHNGEWCEAQTKNGQGWVPSNYITPVN"

test_alignment(1, prot1, prot2, False, "global", MatrixInfo.blosum62, 0, 0, 0, -8)

prot1 = "GARFIELD THE LAST FAT CAT"
prot2 = "GARFIELD THE FAST CAT"

test_alignment(2, prot1, prot2, True, "global", {}, 3, -2, 0, -8)
# test_alignment(2, prot1, prot2, False, "global", MatrixInfo.blosum62, 0, 0, 0, -8)
# test_alignment(3, prot1, prot2, True, "global", MatrixInfo.blosum40, 0, 0, 0, -8)

print(" ")
if launched == passed: print('Passed All Test')
else: print("ERROR: There are failed tests")
```

Output

```
Passed test 1: scores are equal '-88'
Failed test 1: alignments differ:
Biopython:
'GYQYRALYDYKKEREEDIDLHLGDILTVNKGSLVALGFSDGQEARPEEIGWLNGYNETTGERGDFPGTYVEYIGRKKISP'
Score = -88
AlignSequences
'GYQYRALYDYKKEREEDIDLHLGDILTVNKGSLVALGFSDGQEARPEEIGWLNGYNETTGERGDFPGTYVEYIGRKKISP'
Score = -88

### AlignSequences. Global alignment (Needleman-Wunsch)
GARFIELD THE FAS----T CAT
|||||||||||||.||    |||||
GARFIELD THE LAST FAT CAT
        Score: 26
        Similarity (wo gaps): 0.9523809523809523
        Distance (wo gaps): 0.047619047619047616
```

```
        Distance: 0.04
        Init index: (0, 0, 1)
        Matches: 20  Unmatches: 1  Gaps: 4  Gap groups: 1
        Score verified: 26
        Finish. Execution milliseconds: 5
        Score Dictionary Size 1097

BioPython alignment:
GARFIELD THE FAST ----CAT
||||||||||||||.||||    |||
GARFIELD THE LAST FAT CAT
  Score=26


BioPython alignment:
GARFIELD THE FAST---- CAT
||||||||||||||.|||    ||||
GARFIELD THE LAST FAT CAT
  Score=26


BioPython alignment:
GARFIELD THE FAS----T CAT
||||||||||||||.||    |||||
GARFIELD THE LAST FAT CAT
  Score=26

Passed test 2: scores are equal '26'
Passed test 2: alignments are equal 'GARFIELD THE LAST FAT CAT'

ERROR: There are failed tests
```

1.2 From pairs to sets

In each step we need to align two groups of sequences, of length $n >= 1$ and $m >= 1$.

Each group is aligned as a whole, in the sense that the gaps entered in one of the sequences of the group must be introduced in the same positions in the rest of the sequences of their group.

To assign a score to a position, the combined score of all the residuals of that position is used. To do this we produce the Cartesian product nxm of all the characters of that position and calculate the average of scores:

$$\frac{\sum_{\substack{0 \le i < n \\ 0 \le j < m}} matrix(i,j)}{nm}$$

If in any of the positions we have a gap, we have chosen to penalize it as the sum of penalties assigned to the start of the gap plus gap continuation penalty. It is a criterion, *CLUSTAL* we know that it uses another one.

If we already have a pairwise development, as it was my case, it would be easy to extend it to address MSA?. The answer is affirmative. With slight modifications in the class **AlignSequences**, we have managed to address an MSA, in the following way: 1. Generalize the one-position scoring algorithm to take into account all the sequences of both groups, averaging the scores as indicated above.

2. Take a sequence from each group (the first) to perform a simple pairwise alignment (but with the scores calculated as indicated in 1).

3. Compare the sequences resulting from the pairwise alignment with their originals from each group, compute where the gap is introduced and introduce the gap at the same positions in the rest of the sequences of each group.

1.3 Which order? Guide trees.

But The method of progressive alignment based on a guide tree is used.

The guide tree can be obtained in two alternative ways: **Unweighted Pair Group Method with Arithmetic Mean (UPGMA)** and **Neighbor Join (NJ)**, the same options present in *CLUSTAL* software.

The alignment has three known phases. These are the particularities of this implementation on each phase:

1. Perform pairwise alignments between all the sequences involved and assign them a score.

In the case of **UPGMA** we use the proportion (in percentages) between matches and matches plus no matches (without taking gaps into account). That is, we use a measure of the identity between the two sequences involved. You can also use the distance, which would be the complement to 100 of identity, but we wanted to do so to be able to compare with the information that *CLUSTAL* throws at the beginning of his output. It does not affect the result, we simply have to look for maximum identities to build the guide tree, instead of minimum distances. In the case of **NJ**, we have chosen to use distances, computed also in percentages. Also not taking into account the number of gaps in the denominators.

2. Build the guide tree. As I said, we can do it using UPGMA or NJ. The NJ method generates an unrooted tree. As we need a root, for purposes of the subsequent alignment, we have chosen to root it by clustering the two nodes that have no relation. There are other approaches.

1.4 Putting it all together: the CLUSTAL way.

The guide tree can be obtained in two alternative ways: **Unweighted Pair Group Method with Arithmetic Mean (UPGMA)** and **Neighbor Join (NJ)**.

The alignment has three phases. These are the particularities of this implementation on each phase:

1. Perform pairwise alignments between all the sequences involved and assign them a score.

In the case of **UPGMA** we use the proportion (in percentages) between matches and matches plus no matches (without taking gaps into account). That is, we use a measure of the identity between the two sequences involved. You can also use the distance, which would be the complement to 100 of identity, but we wanted to do so to be able to compare with the information that *CLUSTAL* throws at the beginning of his output. It does not affect the result, we simply have to look for maximum identities to build the guide tree, instead of minimum distances. In the case of **NJ**, we have chosen to use distances, computed also in percentages. Also not taking into account the number of gaps in the denominators. 2. Build the guide tree. The NJ method generates an unrooted tree. As we need a root, for purposes of the subsequent alignment, we have chosen to root it by clustering the two nodes that have no relation. There are other strategies.

3. Multiple alignment. Progressive alignment following the order indicated by the guide tree.

1.5 From CLUSTAL to T-COFFEE

1. Substitution matrices to weight matrices

- But some versions of CLUSTAL uses weight matrices

2. Weight matrices calculated combining information of several alignments:

- Global one (Waterman)
- Local one (Smith-Waterman and others)

3. Guide trees by NJ or UPGMA as CLUSTAL.
4. Revision of initial alignments. 3-tuples progressive alignment

1.6 Beyond T-COFFEE

TODO: general comments about MUSCLE or other approaches

1.7 MSA software

1.7.1 MSA generic methods

Script 1.7.1 (python)

```python
"""This methods shows alternative implementations of multiple sequence alignments, CLUSTAL
↪  and  T-COFFEE.

TODO:
    * Many more tests. Create a test battery.

    * Achieve that the results obtained are more similar to those of CLUSTAL (if they have
↪  to be).
    Given the lack of detailed information it will be necessary to resort
    to the sources (in C++) of CLUSTAL.

    * Allow to configure the initial alignment and the final multialignments with different
↪  parameters.

    * Draw the alignments in a more standard way.

    * Draw the phylogenetic trees.

    * Include all new methods in AlignSequences or in another class.
"""
from ete3 import Tree, TreeStyle
MIN_SCORE = 0

def draw_guide_tree(tree):
    """
    Draw guide tree with ETE library
    """
    t = Tree(tree + ";")
    ts = TreeStyle()
```

```
    ts.show_leaf_name = True
    ts.show_branch_length = False
    ts.show_branch_support = False
    ts.scale = 160
    ts.branch_vertical_margin = 40
    print(t)
    return t, ts

def readFasta(file):
    """
    Reads all sequences of a FASTA file
    Args:
        file (str): name of the imput FASTA file
    Returns:
        dict of str, str: sequences readed
    """
    ret_seqs = {}
    seq = ""
    key_found = False
    with open(file, 'r') as f:
        key = ""
        for line in f:
            line = line.replace('\n', '')
            if len(line) > 0:
                if line[0] == ">":
                    if key_found:
                        ret_seqs[key] = seq
                    key_found = True
                    key = line[1:].split(" ")[0]
                    seq = ""
                elif key_found:
                    seq += line
    if key_found:
        ret_seqs[key] = seq
    return ret_seqs

def pairwise_align(s1, s2, matrix, matrix_mode, mode, score_gap_ini=0, score_gap_cont=-8,\
                   score_match=3, score_no_match=-2):
    """
    Performs initial pairwise alignments against the class AlignSequences
    returning the %identity.

    Args:
        s1 (str): First sequence to compare.
        s2 (str): Second sequence to compare.
        matrix (dict of tuples of int): Substitution matrix, Biopython format
        matrix_mode (str): Type of matrix
            'SUBST'            Substitution matrix
            'WEIGHT'           Weight matrix
        mode (str): Computation mode:
            'GLOBAL'           Global Alignment
            'LOCAL'            Local Alignment
```

```
            'LONG_SUBSTRING'   Obtain long common substring
        score_gap_ini (int): Score of gap init.
        score_gap_cont (int): Score of gap continuation.
        score_match (int): Score of match characters (used if no matrix informed)
        score_no_match (int): Score of no match characters (used if no matrix informed)

    Returns:
        (int): % identity between sequences
    """
    align = AlignSequences([s1, s2])
    align.set_scores(score_match, score_no_match, score_gap_ini, score_gap_cont)
    align.set_subst_matrix(matrix)
    align.set_matrix_mode(matrix_mode)
    align.compute(mode.upper(), silent = True)
    return round((align.matches + 1) * 100 / (align.matches + align.unmatches + 2))

def pairwise_align_distance(s1, s2, matrix, matrix_mode, mode, score_gap_ini=0,
↪   score_gap_cont=-8):
    """
    Performs initial pairwise alignments against the class AlignSequences
    returning the distance between 0 and 100.

    Args:
        s1 (str): First sequence to compare.
        s2 (str): Second sequence to compare.
        matrix (dict of tuples of int): Substitution matrix, Biopython format.
        matrix_mode (str): Type of matrix
            'SUBST'            Substitution matrix
            'WEIGHT'           Weight matrix
        mode (str): Computation mode:
            'GLOBAL'           Global Alignment
            'LOCAL'            Local Alignment
            'LONG_SUBSTRING'   Obtain long common substring
        score_gap_ini (int): Score of gap init.
        score_gap_cont (int): Score of gap continuation.
        score_match (int): Score of match characters (used if no matrix informed)
        score_no_match (int): Score of no match characters (used if no matrix informed)

    Returns:
        (int): distance between sequences
    """
    align = AlignSequences([s1, s2])
    align.set_scores(0, 0, score_gap_ini, score_gap_cont)
    align.set_subst_matrix(matrix)
    align.set_matrix_mode(matrix_mode)
    align.compute(mode.upper(), silent = True)
    identity = round((align.matches + 1) * 100 / (align.matches + align.unmatches + 2))
    return 100 - identity

def guide_tree_UPGMA(sequences, matrix, matrix_mode, mode,\
                     score_gap_ini, score_gap_cont,\
                     score_match, score_no_match):
```

```
    """
    Performs initial pairwise alignments against the class AlignSequences
    returning the guide_tree derived from UPGMA method.

    Args:
        sequences (lit of str): Sequences to align
        matrix (dict of tuples of int): Substitution matrix, Biopython format.
        matrix_mode (str): Type of matrix
            'SUBST'             Substitution matrix
            'WEIGHT'            Weight matrix
        mode (str): Computation mode:
            'GLOBAL'            Global Alignment
            'LOCAL'             Local Alignment
            'LONG_SUBSTRING'    Obtain long common substring
        score_gap_ini (int): Score of gap init.
        score_gap_cont (int): Score of gap continuation.
        score_match (int): Score of match characters (used if no matrix informed)
        score_no_match (int): Score of no match characters (used if no matrix informed)

    Returns:
        (list of 3-tuples of int): guide three, the third position of the tuple contains the
    root
                               of the other two nodes.
        (dict of int, boolean = True): contains all nodes

    """
    tree = {} #initial tree
    guide_tree = [] #guided tree, pairs to align in sequence
    max_score = MIN_SCORE
    max_score_position = ()
    for i in range(0, len(sequences)):
        for j in range(0 , i):
            if (i,j) not in tree:
                score = pairwise_align(sequences[i], sequences[j], matrix, matrix_mode, mode,\
                                score_gap_ini, score_gap_cont, score_match, score_no_match)
                tree[(i,j)] = score
                if score >= max_score:
                    max_score = score
                    max_score_position = (i,j)

    print(tree)
    len_tree = len(sequences)
    guide_tree_nodes = {}
    # Generate guide tree. At every step we compute another row averaging the
    # most closer rows and removing all their row coordinates from the tree
    while len(tree) > 0:
        (imax, jmax) = max_score_position
        guide_tree.append((imax, jmax, len_tree))
        guide_tree_nodes[imax] = True
        guide_tree_nodes[jmax] = True
        guide_tree_nodes[len_tree] = False

```

```
        # Average scores from i,j rows into new row in new_row_pos
        for j in range(0, len_tree):
            if j in [imax, jmax]:
                continue
            nscores = 0.0;
            for coordinate in [(imax, j), (j, imax), (jmax, j), (j, jmax)]:
                if coordinate in tree:
                    score = tree[coordinate]
                    nscores += 1
                    if (len_tree, j) not in tree:
                        tree[(len_tree, j)] = score
                    else:
                        tree[(len_tree, j)] += score
            if nscores > 0:
                tree[(len_tree, j)] = tree[(len_tree, j)] / nscores

        # Tree cleaning and calc max score
        max_score = MIN_SCORE
        max_score_position = ()
        for i in range(0, len_tree + 1):
            for j in range(0, len_tree + 1):
                if (i,j) in tree:
                    if i == imax or i == jmax or j == imax or j == jmax:
                        del(tree[(i,j)])
                    else:
                        if tree[(i,j)] >= max_score:
                            max_score =  tree[(i,j)]
                            max_score_position = (i,j)

        len_tree += 1

    return guide_tree, guide_tree_nodes

def q(i, j, nseq, n, dmatrix):
    """
    NJ method: calculate element of intermediate Q matrix.
    """
    d = (nseq - 2) * dmatrix[(i,j)]
    for k in range(0, n):
        if (i,k) in dmatrix:
            d -= dmatrix[(i,k)]
        if (j,k) in dmatrix:
            d -= dmatrix[(j,k)]
    return d

def calc_qmatrix(nseq, n, dmatrix):
    """
    NJ method: calculate intermediate Q matrix.
    """
    qmatrix = {}
    for (i,j) in dmatrix:
        qmatrix[(i,j)] = q(i, j, nseq, n, dmatrix)
```

```
233     return qmatrix
234
235 def smallest_q(qmatrix):
236     """
237     NJ method: returns the coordinates of the minimun score in intermediate Q matrix.
238     """
239     sq = ()
240     min_sq = - MIN
241     for key in qmatrix.keys():
242         if qmatrix[key] <  min_sq:
243             min_sq = qmatrix[key]
244             sq = key
245     return sq
246
247 def djoin(joined_pair, nseq, n, dmatrix):
248     """
249     NJ method: returns distances of joined nodes to the rooted node, so, it returns the
    ↪  branch lengths
250     """
251     (i, j) = joined_pair
252     d_i_1 = dmatrix[(i,j)] / 2.0
253     d_i_2 = 0
254     for k in range(0, n):
255         if (i,k) in dmatrix:
256             d_i_2 += dmatrix[(i,k)]
257         if (j,k) in dmatrix:
258             d_i_2 -= dmatrix[(j,k)]
259     d_i = d_i_1 - d_i_2 / (2*(nseq - 2))
260     d_j = dmatrix[(i,j)] - d_i
261     return d_i, d_j
262
263 def dnjoin(k, joined_pair, dmatrix):
264     """
265     NJ method: returns distance of sequence k to the new nodetht routes the joined_pair.
266     The distance is the mean of the distances ftom k to each of nodes joined.
267     """
268     (i, j) = joined_pair
269     d_k = 0
270     if (i,k) in dmatrix:
271         d_k += dmatrix[(i,k)]
272     if (j,k) in dmatrix:
273         d_k += dmatrix[(j,k)]
274     d_k = (d_k - dmatrix[(i,j)]) / 2.0
275     return d_k
276
277 def recalc_dmatrix(joined_pair, n, dmatrix):
278     """
279     NJ method: recalc distance matrix taking into account the joined pair
280     """
281     (i, j) = joined_pair
282     # Recalculate distances
283     for k in range(0, n):
```

```
284         if (i,k) in dmatrix and (j,k) in dmatrix:
285             dmatrix[(n + 1, k)] = dnjoin(k, joined_pair, dmatrix)
286             dmatrix[(k, n + 1)] = dmatrix[(n + 1, k)]
287     # Remove joined rows from dmatrix
288     for k in range(0, n + 1):
289         for l in range(0, n + 1):
290             if k == i or k == j or l == i or l== j:
291                 if (k, l) in dmatrix:
292                     del(dmatrix[(k, l)])
293     return
294 
295 def guide_tree_NJ(sequences, matrix, matrix_mode, mode,\
296                   score_gap_ini, score_gap_cont,\
297                   score_match, score_no_match):
298     """
299     Performs initial pairwise alignments against the class AlignSequences
300     returning the guide_tree derived from NJ method.
301 
302     Args:
303         sequences (lit of str): Sequences to align
304         matrix (dict of tuples of int): Substitution matrix, Biopython format.
305         matrix_mode (str): Type of matrix
306             'SUBST'             Substitution matrix
307             'WEIGHT'            Weight matrix
308         mode (str): Computation mode:
309             'GLOBAL'            Global Alignment
310             'LOCAL'             Local Alignment
311             'LONG_SUBSTRING'    Obtain long common substring
312         score_gap_ini (int): Score of gap init.
313         score_gap_cont (int): Score of gap continuation.
314         score_match (int): Score of match characters (used if no matrix informed)
315         score_no_match (int): Score of no match characters (used if no matrix informed)
316 
317     Returns:
318         (list of 3-tuples of int): guide three, the third position of the tuple contains the
    ↪  root
319                                 of the other two nodes.
320         (dict of int, boolean = True): contains all nodes
321 
322     """
323     dmatrix = {} #initial distance matrix
324     n = len(sequences)
325     for i in range(0, n):
326         for j in range(0 , i):
327             if (i,j) not in dmatrix:
328                 distance = pairwise_align_distance(sequences[i], sequences[j], matrix,
                     ↪  matrix_mode,\
329                                    mode, score_gap_ini, score_gap_cont)
330                 dmatrix[(i,j)] = distance
331                 dmatrix[(j,i)] = distance
332     nseq = n
333     new_nodes = n - 2
```

```
    guide_tree = [] #guided tree, pairs to align in sequence
    guide_tree_nodes = {} #guided tree rooted nodes to complete
    for i in range(0, n):
        guide_tree_nodes[i] = False
    while new_nodes > 0:
        qmatrix = calc_qmatrix(nseq, n, dmatrix)
        (joined_i, joined_j) = smallest_q(qmatrix)
        #print("JOIN:", (joined_i, joined_j))
        guide_tree.append((joined_i, joined_j, n + 1))
        guide_tree_nodes[joined_i] = True
        guide_tree_nodes[joined_j] = True
        guide_tree_nodes[n + 1] = False
        recalc_dmatrix((joined_i, joined_j), n, dmatrix)
        n += 1
        nseq -= 1
        new_nodes -= 1
    # Root the tree
    #print("DMATRIX:", dmatrix)
    rooting_tuple = []
    for node in guide_tree_nodes:
        if not guide_tree_nodes[node]:
            rooting_tuple.append(node)
    rooting_tuple.append(n + 1)
    guide_tree_nodes[n + 1] = True
    #print("Rooting tuple:", rooting_tuple)
    if len(rooting_tuple) == 3:
        guide_tree.append(tuple(rooting_tuple))
    assert len(rooting_tuple) == 3
    return guide_tree, guide_tree_nodes

def gapeator(a, a_gapped, b_stack, b_stack_refs):
    """
    Introduces gaps in all the sequences of b_stack taking into account the positions
    and the gaps introduced in sequence a to obtain sequence a_gapped
    Args:
        a (str): template sequence not gapped
        a_gapped (str): template sequence gapped
        b_stack (list of str): stack of b sequences ungapped
        b_stack_refs (list of dict): stack of references to original positions
    Returns:
        list of str: stack b gapped as a does
        list of dict: stack b coordinates refered to original sequence
    """
    ini_a_gapped = a_gapped
    b_gapped_stack = []
    b_references_stack = []
    len_a_gapped = len(a_gapped)
    for b, b_refs in zip(b_stack, b_stack_refs):
        b_gapped = ""
        b_gapped_references = {}
        a_gapped = ini_a_gapped
        base_ref = 0
```

```
        for k, (i, j) in enumerate(zip(a, b)):
            index = a_gapped.index(i)
            a_gapped = a_gapped[index + 1:]
            #print("a_gapped", a_gapped )
            #print(i, j, index)
            b_gapped += "-" * index + j
            if k in b_refs:
                b_gapped_references[base_ref + k + index] =  b_refs[k]
            base_ref += index
            #print("b_gapped", b_gapped )
        b_gapped += b[k+1:]
        remaining_gaps = "-" * (len_a_gapped - len(b_gapped))
        b_gapped +=  remaining_gaps
        b_gapped_stack.append(b_gapped)
        b_references_stack.append(b_gapped_references)
    return b_gapped_stack, b_references_stack

def pairwise_align_msa_step(stack_0, stack_1, sequences, matrix, matrix_mode,\
                            mode, stack_0_indexes, stack_1_indexes, stack_0_refs,
                             ↪  stack_1_refs,\
                            score_match, score_no_match, score_gap_ini, score_gap_cont):
    """
    Performs msa alignment of sequence stack 0 and 1.

    Args:
        stack_0 (list of str): First stack of sequences to align.
        stack_1 (list of str): Secong stack of sequences to align.
        sequences (list of str): Sequences to align.
        matrix (dict of tuples of int): Substitution matrix, Biopython format.
        matrix_mode (str): Type of matrix
            'SUBST'             Substitution matrix
            'WEIGHT'            Weight matrix
        mode (str): Computation mode:
            'GLOBAL'            Global Alignment
            'LOCAL'             Local Alignment
            'LONG_SUBSTRING'    Obtain long common substring
        stack_0_indexes(list of int) : Indexes of initial sequences related to stack
    ↪  sequences 0
        stack_1_indexes(list of int) : Indexes of initial sequences related to stack
    ↪  sequences 1
        stack_0_refs(list of dict) : stack_0 references to original sequences
        stack_1_refs(list of dict) : stack_1 references to original sequences
        score_match (int): Score of match characters (used if no matrix informed)
        score_no_match (int): Score of no match characters (used if no matrix informed)
        score_gap_ini (int): Score of gap init.
        score_gap_cont (int): Score of gap continuation.

    Returns:
        list of str: stack_0 gapped (with the gaps necessary for the alignment)
        list of str: stack_0 gapped (with the gaps necessary for the alignment)
        list of dict: stack_0 references to original sequences
        list of dict: stack_1 references to original sequences
```

```
    """
    align = AlignSequences([stack_0[0], stack_1[0]])
    align.set_scores(score_match, score_no_match, score_gap_ini, score_gap_cont)
    align.set_subst_matrix(matrix)
    align.set_matrix_mode(matrix_mode)
    align.set_stacks(stack_0, stack_1, stack_0_indexes, stack_1_indexes, stack_0_refs,
    ↪  stack_1_refs)
    align.compute(mode.upper(), silent = True)
    # align_seq0 align_seq1 are the seq0 and seq1 alignments
    # we need to deduce the rest of alignments.
    # what we do is perform the same gap insertions, if any, as the first sequence of the
    ↪  stacks
    # the gap insertions where performed taken into account the initial sequence
    # to compute the references to inital sequence in order to employ a weight matrix if
    ↪  informed
    stack_0_gapped, stack_0_references = gapeator(stack_0[0], align.align_seq0, stack_0,
    ↪  stack_0_refs)
    stack_1_gapped, stack_1_references = gapeator(stack_1[0], align.align_seq1, stack_1,
    ↪  stack_1_refs)
    return stack_0_gapped, stack_1_gapped, stack_0_references, stack_1_references

def get_name(index, sequence_names):
    """
    Obtain sequence name from index
    """
    name = ""
    if index < len(sequence_names):
        name = sequence_names[index]
    else:
        name = str(index)
    return name

def to_newick(tree, sequence_names):
    """
    Obtain guide tree in newick format
    """
    # Change format to intermediate roots
    roots = {}
    newick_tree = ""
    for branch in tree:
        (i, j, k) = branch
        name_i = get_name(i, sequence_names)
        name_j = get_name(j, sequence_names)
        name_k = get_name(k, sequence_names)
        if name_i in roots:
            new_root_i = roots[name_i]
        else:
            new_root_i = name_i
        if name_j in roots:
            new_root_j = roots[name_j]
        else:
            new_root_j = name_j
```

```
482         roots[name_k] = [new_root_i, new_root_j]
483 
484     for root in roots.values():
485         s_root = str(root)
486         if len(s_root) > len(newick_tree):
487             newick_tree = s_root.replace("[","(").replace("]",")").replace("'","")
488 
489     return newick_tree
```

1.7.2 T-COFFEE methods

Script 1.7.2 (python)

```
1  # T-COFFEE specific methods
2  def pairwise_align_coffee(s1, s2, matrix, mode, score_gap_ini=0, score_gap_cont=-8,\
3                    score_match=3, score_no_match=-2):
4      """
5      Performs initial pairwise alignments against the class AlignSequences
6      returning the %identity and the alignments to construct the primary library
7 
8      Args:
9          s1 (str): First sequence to compare.
10         s2 (str): Second sequence to compare.
11         matrix (dict of tuples of int): Substitution matrix, Biopython format.
12         mode (str): Computation mode:
13             'GLOBAL'           Global Alignment
14             'LOCAL'            Local Alignment
15             'LONG_SUBSTRING'   Obtain long common substring
16         score_gap_ini (int): Score of gap init.
17         score_gap_cont (int): Score of gap continuation.
18         score_match (int): Score of match characters (used if no matrix informed)
19         score_no_match (int): Score of no match characters (used if no matrix informed)
20 
21     Returns:
22          int: % identity between sequences
23          str: sequence 1 aligned
24          str: sequence 2 aligned
25     """
26     align = AlignSequences([s1, s2])
27     align.set_scores(score_match, score_no_match, score_gap_ini, score_gap_cont)
28     align.set_subst_matrix(matrix)
29     align.compute(mode.upper(), silent = True)
30     return round((align.matches + 1) * 100 / (align.matches + align.unmatches + 2)), \
31             align.align_seq0, align.align_seq1
32 
33 def get_pos(k, seq_i, align_i):
34     """
35     Obtain position of a character in the original sequence given the
36     position in the alignment(k), the original sequence (seq_i)
37     and the align_i (gapped) sequence
38     """
```

```
    char = align_i[k]
    count_char = align_i[0:k+1].count(char)
    index = -1;
    for _ in range(0, count_char):
        index = seq_i.find(char, index + 1)
    return index

def update_weight_at_pos(weight_library, i, j, pos_i, pos_j, identity):
    """
    Update weight at pos i , j, pos_i, pos_j
    """
    if i not in weight_library:
        weight_library[i] = {}
    w_i = weight_library[i]
    if j not in w_i:
        w_i[j] = {}
    w_i_j = w_i[j]
    if pos_i not in w_i_j:
        w_i_j[pos_i] = {}
    w_i_j_pi = w_i_j[pos_i]
    if pos_j not in w_i_j_pi:
        w_i_j_pi[pos_j] = identity
    else:
        w_i_j_pi[pos_j] += identity

def update_weight_library(weight_library, i, j, identity,\
                  seq_i, seq_j, align_i, align_j):
    """
    Update weights library from alignments and %identity
    """
    for k, (c_i, c_j) in enumerate(zip(align_i, align_j)):
        if c_i != "-" and c_j != "-":
            pos_i = get_pos(k, seq_i, align_i)
            pos_j = get_pos(k, seq_j, align_j)
            #print("Position:", pos_i, pos_j)
            update_weight_at_pos(weight_library, i, j, pos_i, pos_j, identity)
            update_weight_at_pos(weight_library, j, i, pos_j, pos_i, identity)

def compute_library(sequences, matrix={}, weight_library={}, mode="GLOBAL",\
                  score_gap_ini=0, score_gap_cont=-8,\
                  score_match=3, score_no_match=-2):
    """
    Compute initial library of identities based on scores of PA
     Args:
        sequences (list of str): Seguences to compare.
        matrix (dict of tuples of int): Substitution matrix, Biopython format.
        mode (str): Computation mode:
            'GLOBAL'           Global Alignment
            'LOCAL'            Local Alignment
            'LONG_SUBSTRING'   Long substring alignment
        score_gap_ini (int): Score of gap init.
        score_gap_cont (int): Score of gap continuation.
```

```
        score_match (int): Score of match characters (used if no matrix informed)
        score_no_match (int): Score of no match characters (used if no matrix informed)

    Returns:
        list of str: primary library of alignments
    """
    primary_library = {}
    for i in range(0, len(sequences)):
        for j in range(0 , i):
            if (i,j) not in primary_library:
                identity, align_i, align_j = pairwise_align_coffee(sequences[i],
                 ↪  sequences[j],\
                               matrix, mode, score_gap_ini, score_gap_cont,\
                               score_match, score_no_match)
                update_weight_library(weight_library, i, j, identity,\
                              sequences[i], sequences[j], align_i, align_j)
                primary_library[(i,j)] = (align_i, align_j, identity)
    #print(weight_library)
    return primary_library

def extend_weigths(weight_library, i, k, j):
    """
    Extend weigths for pair of sequences (i,j) at pos (pos_i_posj)
    taken into account the routes using k as
    intermediate, by means of the alignments (i, k) and (k, j).
    """
    for pos_i, pos_i_j in weight_library[i][j].items():
        for pos_j in pos_i_j.keys():
            if pos_j in weight_library[j][k].keys():
                for pos_k in weight_library[j][k][pos_j].keys():
                    if pos_k in weight_library[k][i].keys():
                        for pos_i_new in weight_library[k][i][pos_k].keys():
                            if pos_i_new == pos_i:
                                #print("Extension", pos_i, pos_j, pos_k, weight_library[i][k
                                 ↪  ][pos_i][pos_k],weight_library[j][k][pos_j][pos_k])
                                m = min(\
                                    weight_library[i][k][pos_i][pos_k],\
                                    weight_library[j][k][pos_j][pos_k])
                                #print("++", m, weight_library[i][j][pos_i][pos_j])
                                weight_library[i][j][pos_i][pos_j] +=  m

def extend_library_weigths(sequences, weight_library):
    """
    Extend library for all triplets of sequences
    Taken into account the simetry i -> k -> j
    """
    len_sequences = len(sequences)
    for i in range(0, len_sequences):
        for k in range(0, len_sequences):
             if k != i:
                for j in range(0, len_sequences ):
                    if j != k and j != i:
```

```
                        #print("Triplet:", i, k, j)
                        extend_weigths(weight_library, i, k, j)


def compute_libraries(sequences, matrix,\
                      score_gap_ini, score_gap_cont, score_match, score_no_match):
    """
    Compute initial library of identities based on scores of pairwise alignments
     Args:
        sequences (list of str): Seguences to compare.
        matrix (dict of tuples of int): Substitution matrix, Biopython format.
        score_gap_ini (int): Score of gap init.
        score_gap_cont (int): Score of gap continuation.
        score_match (int): Score of match characters (used if no matrix informed)
        score_no_match (int): Score of no match characters (used if no matrix informed)

    Returns:
        dict : primary library of alignments
        dict : weight matrix
    """
    weight_library = {}
    primary_library = compute_library(sequences, matrix,\
                weight_library, "GLOBAL", score_gap_ini, score_gap_cont,\
                score_match, score_no_match)

    _ = compute_library(sequences, matrix,\
                weight_library, "LOCAL", score_gap_ini, score_gap_cont,\
                score_match, score_no_match)

#     _ = compute_library(sequences, matrix,\
#                 weight_library, "LONG_SUBSTRING", score_gap_ini, score_gap_cont,\
#                 score_match, score_no_match)

    extend_library_weigths(sequences, weight_library)

    return primary_library, weight_library
```

1.7.3 Main MSA method

Script 1.7.3 (python)

```
# Generic MSA method
def do_msa_from_fasta(file, main_alg="CLUSTAL", method="NJ", matrix={}, matrix_mode="SUBST",\
                      mode="GLOBAL", score_gap_ini =-10, score_gap_cont=-5, score_match=3,\
                      score_no_match=-2, verbose=False):
    """
    Performs MSA alignments from fasta file
    Args:
        file (str): Name of the FASTA file.
        main_alg (str): Main algorithm:
            "CLUSTAL"       Clustal like
```

```
            "T-COFFEE"      T-COFFEE like
        method (str): NJ neighbor join / UPGMA
        matrix (dict of tuples of int): Substitution matrix, Biopython format.
        matrix_mode (str): Type of matrix
            'SUBST'             Substitution matrix
            'WEIGHT'            Weight matrix
        mode (str): Computation mode:
            'GLOBAL'            Global Alignment
            'LOCAL'             Local Alignment
            'LONG_SUBSTRING'    Obtain long common substring
        score_gap_ini (int): Score of gap init.
        score_gap_cont (int): Score of gap continuation.
        score_match (int): Score of match characters (used if no matrix informed)
        score_no_match (int): Score of no match characters (used if no matrix informed)
        verbose (bool): If True prints verbose info

    Returns:
        list of 3-tuples of int: guide three, the third position of the tuple contains the root
                            of the other two nodes.
        list of str: alignments
        list of str: sequence_names
        list of int: sequence indexes relating strings in alignment to original sequences
    """
    seq_fasta = readFasta(file)
    sequences = list(seq_fasta.values())
    sequence_names = list(seq_fasta.keys())
    print(sequence_names)
    return do_msa(sequences, sequence_names,\
                  main_alg, method, matrix, matrix_mode,\
                  mode, score_gap_ini, score_gap_cont, score_match,\
                  score_no_match, verbose)

def do_msa(sequences, sequence_names, main_alg="CLUSTAL", method="NJ", matrix={}, matrix_mode="SUBST",\
                     mode="GLOBAL", score_gap_ini=-10, score_gap_cont=-5, score_match=3,\
                     score_no_match=-2, verbose=False):
    """
    Performs MSA alignments from sequences
    Args:
        sequences (list of str): Sequences
        sequence_names (list of str): Names of sequences
        main_alg (str): Main algorithm:
            "CLUSTAL"       Clustal like
            "T-COFFEE"      T-COFFEE like
        method (str): NJ neighbor join / UPGMA
        matrix (dict of tuples of int): Substitution matrix, Biopython format.
        matrix_mode (str): Type of matrix
            'SUBST'             Substitution matrix
            'WEIGHT'            Weight matrix
        mode (str): Computation mode:
            'GLOBAL'            Global Alignment
```

```
            'LOCAL'            Local Alignment
            'LONG_SUBSTRING'   Obtain long common substring
        score_gap_ini (int): Score of gap init.
        score_gap_cont (int): Score of gap continuation.
        score_match (int): Score of match characters (used if no matrix informed)
        score_no_match (int): Score of no match characters (used if no matrix informed)
        verbose (bool): If True prints verbose info

    Returns:
        list of 3-tuples of int: guide three, the third position of the tuple contains the root
                                 of the other two nodes.
        list of str: alignments
        list of str: sequence_names
        list of int: sequence indexes relating strings in alignment to original sequences
    """
    if main_alg == "T-COFFEE":
        primary_library, weight_library = compute_libraries(sequences, matrix,\
                        score_gap_ini, score_gap_cont, score_match, score_no_match)
        #print("Primary library", primary_library)
        #print("Weight library", weight_library)
        # Matrix mode and other MSA parameters
        matrix = weight_library
        matrix_mode = "WEIGHT"
#         score_gap_ini = 0
#         score_gap_cont = 0
        # From here only we need is to compute a MSA with weight matrix as reference.
    else:
        matrix_mode = "SUBST"
    if method == "NJ":
        guide_tree, guide_tree_nodes =\
            guide_tree_NJ(sequences, matrix, matrix_mode,\
            mode, score_gap_ini, score_gap_cont,\
            score_match, score_no_match)
    else: #UPGMA
        guide_tree, guide_tree_nodes =\
            guide_tree_UPGMA(sequences, matrix, matrix_mode,\
            mode, score_gap_ini, score_gap_cont,\
            score_match, score_no_match)
    sequences_store = {}
    sequences_store_indexes = {}
    sequences_store_refs = {}
    print("Guide Tree",guide_tree, sequence_names)
    #return guide_tree, "", sequence_names
    # Create MSA
    for i in guide_tree_nodes.keys():
        if i < len(sequences):
            sequences_store[i] = [sequences[i]]
            sequences_store_indexes[i] = [i]
            sequences_store_refs[i] = []
            autorefs = {}
            for k in range(0, len(sequences[i])):
```

```
                autorefs[k] = k
            sequences_store_refs[i].append(autorefs)

    if verbose: print(sequences_store)
    for (i ,j ,k) in guide_tree:
        stack_i = sequences_store[i]
        stack_j = sequences_store[j]
        stack_i_indexes = sequences_store_indexes[i]
        stack_j_indexes = sequences_store_indexes[j]
        stack_i_references = sequences_store_refs[i]
        stack_j_references = sequences_store_refs[j]
        if verbose: print("Stack i", i, stack_i)
        if verbose: print("Stack j", j, stack_j)
        if verbose: print("Stack i_indexes", i, stack_i_indexes)
        if verbose: print("Stack j_indexes", j, stack_j_indexes)
        stack_0, stack_1, stack_0_references, stack_1_references =\
            pairwise_align_msa_step(stack_i, stack_j, sequences, matrix, matrix_mode,\
                                    mode, stack_i_indexes, stack_j_indexes,\
                                    stack_i_references, stack_j_references,\
                                    score_match, score_no_match, score_gap_ini,
                                     ↪  score_gap_cont)
        sequences_store[k] = []
        sequences_store_indexes[k] = []
        sequences_store_refs[k] = []
        if verbose: print("==================")
        for s in stack_0:
            if verbose: print(s)
            sequences_store[k].append(s)
        for s in stack_1:
            if verbose: print(s)
            sequences_store[k].append(s)
        for seq_index in stack_i_indexes:
            sequences_store_indexes[k].append(seq_index)
        for seq_index in stack_j_indexes:
            sequences_store_indexes[k].append(seq_index)
        for seq_refs in stack_0_references:
            sequences_store_refs[k].append(seq_refs)
        for seq_refs in stack_1_references:
            sequences_store_refs[k].append(seq_refs)
        if verbose: print("==================")
        if verbose: print("Sequences store indexes",sequences_store_indexes[k])
        if verbose: print("Sequences store references",sequences_store_refs[k])
        if verbose: print("New stack:", k, sequences_store[k])
    alignment = sequences_store[k]
    newick_tree = to_newick(guide_tree, sequence_names)
    return newick_tree, alignment, sequence_names, sequences_store_indexes[k]

def score(alignment, matrix, score_gap_ini=0, score_gap_cont=0):
    """
    Score based on sum of pair scores (SOP) taking into account substitution matrix
    Derived from the objetive score of MUSCLE refinement stage
    """
```

```
    msa_score = 0
    for k in range(0, len(alignment[0])): #columns of msa
        score_column_k = 0
        nvalues = 0
        for i in range (0, len(alignment)):
            for j in range (i + 1, len(alignment)):
                if alignment[i][k] == "-" and alignment[j][k] != "-":
                    score_column_k += score_gap_cont
                    if k == 0 or alignment[i][k-1] != "-":
                            score_column_k += score_gap_ini
                if alignment[j][k] == "-" and alignment[i][k] != "-":
                    score_column_k += score_gap_cont
                    if k == 0 or alignment[j][k-1] != "-":
                            score_column_k += score_gap_ini
                elif (alignment[i][k], alignment[j][k]) in matrix:
                    score_column_k += matrix[(alignment[i][k], alignment[j][k])]
                    nvalues += 1
                elif (alignment[j][k], alignment[i][k]) in matrix:
                    score_column_k += matrix[(alignment[j][k], alignment[i][k])]
                    nvalues += 1
        if nvalues > 0:
            #score += score_column_k / nvalues
            msa_score += score_column_k
    return msa_score

def score_from_fasta(file, matrix, score_gap_ini=0, score_gap_cont=0):
    seq_fasta = readFasta(file)
    sequences = list(seq_fasta.values())
    return score(sequences, matrix, score_gap_ini, score_gap_cont)
```

1.8 Some runs

1.8.1 T-COFFEE RUNS

Script 1.8.1 (python)

```
matrix = MatrixInfo.blosum80
file = "sample.fasta"
quality_score_gap_ini = -10
quality_score_gap_cont = -5

guide_tree_upgma, align, sequence_names, indexes = do_msa_from_fasta(file,\
                main_alg = "T-COFFEE", method = "UPGMA", \
                matrix = matrix, matrix_mode = "SUBST",\
                mode = "GLOBAL", score_gap_ini = -10,\
                score_gap_cont = -5, score_match = 3, score_no_match = -2, verbose = False)

print("# Guide Tree:", guide_tree_upgma)
t_upgma, ts_upgma = draw_guide_tree(guide_tree_upgma)
print("# Alignment:")
for i, s in enumerate(align):
```

```
    print(">" + sequence_names[indexes[i]])
    print(s)
print()
print("Score", score(align, MatrixInfo.blosum62, quality_score_gap_ini,
 ↪  quality_score_gap_cont))
print()

guide_tree_nj, align, sequence_names, indexes = do_msa_from_fasta(file,\
                main_alg = "T-COFFEE", method = "NJ", \
                matrix = matrix, matrix_mode = "SUBST",\
                mode = "GLOBAL", score_gap_ini = -10,\
                score_gap_cont = -5, score_match = 3, score_no_match = -2, verbose = False)
print("# Guide Tree:", guide_tree_nj)
t_nj, ts_nj = draw_guide_tree(guide_tree_nj)
print("# Alignment:")
for i, s in enumerate(align):
    print(">" + sequence_names[indexes[i]])
    print(s)
print()
print("Score", score(align, matrix, quality_score_gap_ini, quality_score_gap_cont))
```

Output

```
['1aboA', '1ycsB', '1pht', '1vie', '1ihvA']
{(1, 0): 12, (2, 0): 9, (2, 1): 8, (3, 0): 8, (3, 1): 9, (3, 2): 8, (4, 0): 10, (4, 1): 6,
 ↪  (4, 2): 10, (4, 3): 16}
Guide Tree [(4, 3, 5), (1, 0, 6), (5, 2, 7), (7, 6, 8)] ['1aboA', '1ycsB', '1pht', '1vie',
 ↪  '1ihvA']
# Guide Tree: (((1ihvA, 1vie), 1pht), (1ycsB, 1aboA))

         /-1ihvA
      /-|
   /-|   \-1vie
  |  |
--|   \-1pht
  |
  |   /-1ycsB
   \-|
      \-1aboA
# Alignment:
>1ihvA
NFR---VY-YRDSRD------PVWKGPAKLLWKGEGAVVIQDNSDIKVVPRR-------------KAKIIRD-------------
>1vie
D---------------------RVRKKSGAAWQGQIVGWYCTNLTPEGYAVE-------------SEAHPGSVQIYPVAALERIN
>1pht
GYQYRALYDYKKEREEDIDLHLGDILTVNKGSLVALGFSDGQEARPEEIGWLNGYNETTGERGDFPGTYVEYIG-----RKKISP
>1ycsB
KGV-IYALWDYEPQNDDELPMKEGDCMTIIHREDE-DEIEWWWARLNDK---------------EGYVPRNLLG---LYP-----
>1aboA
N-L-FVALYDFVASGDNTLSITKGEKLRVLGYNHN-G--EWCEAQTKNG---------------QGWVPSNYIT---PVN-----
```

```
Score -1648

['1aboA', '1ycsB', '1pht', '1vie', '1ihvA']
Guide Tree [(4, 3, 6), (6, 2, 7), (7, 0, 8), (1, 8, 9)] ['1aboA', '1ycsB', '1pht', '1vie',
↪  '1ihvA']
# Guide Tree: (1ycsB, (((1ihvA, 1vie), 1pht), 1aboA))

   /-1ycsB
  |
  |         /-1ihvA
--|       /-|
  |    /-|   \-1vie
  |   |  |
   \-|   \-1pht
     |
      \-1aboA
# Alignment:
>1ycsB
KGVIYALWDYEPQNDDELPMKEGDCMTIIHREDEDEIEWWWARLNDKEGYVP-----------------RNLL------GLYP-----
>1ihvA
NFR----VY-YRDSRD------PVWKGPAKLLWKG--EGAVVIQDNSDIKVVPRR-------------KAKIIRD-------------
>1vie
D----------------------RVRKKSGAAWQG--QIVGWYCTNLTPEGYAVE-------------SEAHPGSVQIYPVAALERIN
>1pht
GYQYRA-LYDYKKEREEDIDLHLGDILTVNKGSLV--ALGFSDGQEARPEEIGWLNGYNETTGERGDFPGTYVEYIG-----RKKISP
>1aboA
N-LFVA-LYDFVASGDNTLSITKGEKLRVLGYNHN--G-EWCEAQTKNGQGW---------------VPSNYITPVN-----------

Score -1885
```

1.8.2 CLUSTAL homemade

Script 1.8.2 (python)

```python
matrix = {}
file = "sample.fasta"
quality_score_gap_ini = -10
quality_score_gap_cont = -5
guide_tree_upgma, align, sequence_names, indexes = do_msa_from_fasta(file,\
                main_alg = "CLUSTAL", method = "UPGMA", \
                matrix = matrix, matrix_mode = "SUBST",\
                mode = "GLOBAL", score_gap_ini = -10,\
                score_gap_cont = -5, score_match = 3, score_no_match = -2, verbose = False)
print("# Guide Tree:", guide_tree_upgma)
t_upgma, ts_upgma = draw_guide_tree(guide_tree_upgma)
print("# Alignment:")
for i, s in enumerate(align):
    print(">" + sequence_names[indexes[i]])
    print(s)
print()
```

```
print("Score", score(align, MatrixInfo.blosum62, quality_score_gap_ini,
 ↪  quality_score_gap_cont))

print()
guide_tree_nj, align, sequence_names, indexes = do_msa_from_fasta(file,\
                main_alg = "CLUSTAL", method = "NJ", \
                matrix = matrix, matrix_mode = "SUBST",\
                mode = "GLOBAL", score_gap_ini = -10,\
                score_gap_cont = -5, score_match = 3, score_no_match = -2, verbose = False)
print("# Guide Tree:", guide_tree_nj)
t_nj, ts_nj = draw_guide_tree(guide_tree_nj)
print("# Alignment:")
for i, s in enumerate(align):
    print(">" + sequence_names[indexes[i]])
    print(s)
print()
# print("Score", score(align, matrix, quality_score_gap_ini, quality_score_gap_cont))
```

Output

```
['1aboA', '1ycsB', '1pht', '1vie', '1ihvA']
{(1, 0): 24, (2, 0): 19, (2, 1): 23, (3, 0): 19, (3, 1): 13, (3, 2): 25, (4, 0): 16, (4, 1):
 ↪  18, (4, 2): 20, (4, 3): 18}
Guide Tree [(3, 2, 5), (1, 0, 6), (5, 4, 7), (7, 6, 8)] ['1aboA', '1ycsB', '1pht', '1vie',
 ↪  '1ihvA']
# Guide Tree: (((1vie, 1pht), 1ihvA), (1ycsB, 1aboA))

         /-1vie
      /-|
   /-|   \-1pht
  |  |
--|   \-1ihvA
  |
  |   /-1ycsB
   \-|
      \-1aboA
# Alignment:
>1vie
----------DRVRKKSGAAWQGQIVGWYCTNLTPEGYAVESEAHP-------------GSVQIYPVAALERI------N
>1pht
GYQYRALYDYKKEREEDIDLHLGDILTVNKGSLVALGFSDGQEARPEEIGWLNGYNETTGERGDFPGTYVEYIGRKKISP
>1ihvA
-----------NFRVYYRDSRDPVWKGPAKLLWKGEGAVVIQ--------------------DNSDIKVVPRRKAKIIRD
>1ycsB
KGVIYALWDYEPQNDDELPMKEGDCMTIIHREDEDEIEWWWA--------------------RLNDKEGYVPRNLLGLYP
>1aboA
-NLFVALYDFVASGDNTLSITKG--EKLRVLGYNHNGEWCEA--------------------QTKNGQGWVPSNYITPVN

Score -1105

['1aboA', '1ycsB', '1pht', '1vie', '1ihvA']
```

```
Guide Tree [(1, 0, 6), (6, 2, 7), (7, 4, 8), (3, 8, 9)] ['1aboA', '1ycsB', '1pht', '1vie',
↪  '1ihvA']
# Guide Tree: (1vie, (((1ycsB, 1aboA), 1pht), 1ihvA))

   /-1vie
  |
  |         /-1ycsB
--|      /-|
  |   /-|   \-1aboA
  |  |  |
   \-|   \-1pht
     |
      \-1ihvA
# Alignment:
>1vie
DR----------VRKKSGAAWQGQIVGWYCTNLTPEGYAVESEAHPGSVQIYPVAALE-------------------RIN
>1ycsB
KGVIYALWDY--EPQNDDELPMKEGDCMTIIHREDEDEIEWWWARLNDKEGYVPRNLLG-----------------LYP
>1aboA
-NLFVALYDF--VASGDNTLSITKG--EKLRVLGYNHNGEWCEAQTKNGQGWVPSNYIT-----------------PVN
>1pht
GYQYRALYDYKKEREEDIDLHLGDILTVNKGSLVALGFSDGQEARPEEIGWLNGYNETTGERGDFPGTYVEYIGRKKISP
>1ihvA
NFRVYYRDSRDPVWKGPAKLLWKGEGAVVIQDNSDIKVVPRRKAKI------------------------------IRD
```

1.8.3 CLUSTALW

Script 1.8.3 (text)

```
1 %%bash
2 ./clustalw2  -OUTPUTTREE=phylip -NEGATIVE -INFILE=sample.fasta -OUTORDER=ALIGN
  ↪  -STATS=align.log -TREE -ALIGN -CLUSTERING=NJ -OUTFILE=align.fasta -OUTPUT=CLUSTAL
  ↪  -MATRIX=BLOSUM -TYPE=PROTEIN -PWGAPOPEN=10 -PWGAPEXT=5
3 cat align.fasta
4 cat sample.dnd
```

Output

```
 CLUSTAL 2.1 Multiple Sequence Alignments


Sequence type explicitly set to Protein
Sequence format is Pearson
Sequence 1: 1aboA         57 aa
Sequence 2: 1ycsB         60 aa
Sequence 3: 1pht          80 aa
Sequence 4: 1vie          51 aa
Sequence 5: 1ihvA         49 aa
Start of Pairwise alignments
Aligning...
```

```
Sequences (1:2) Aligned. Score:  22
Sequences (1:3) Aligned. Score:  12
Sequences (1:4) Aligned. Score:  5
Sequences (1:5) Aligned. Score:  6
Sequences (2:3) Aligned. Score:  11
Sequences (2:4) Aligned. Score:  9
Sequences (2:5) Aligned. Score:  4
Sequences (3:4) Aligned. Score:  15
Sequences (3:5) Aligned. Score:  12
Sequences (4:5) Aligned. Score:  6
Guide tree file created:   [sample.dnd]

There are 4 groups
Start of Multiple Alignment

Aligning...
Group 1:                    Delayed
Group 2:                    Delayed
Group 3:                    Delayed
Group 4:                    Delayed
Alignment Score -155

CLUSTAL-Alignment file created  [align.fasta]

CLUSTAL 2.1 multiple sequence alignment


1aboA           -NLFVALYDFVASGDNTLSITKGEKLRVLGY---------NHNG-----EWCEAQ--TKN
1ycsB           KGVIYALWDYEPQNDDELPMKEGDCMTII------------HREDEDEIEWWWAR--LND
1pht            GYQYRALYDYKKEREEDIDLHLGDILTVNKGSLVALGFSDGQEARPEEIGWLNGYNETTG
1vie            ----------------------DRVRKKSGAAWQG----------QIVGWYCTN---LT
1ihvA           -------------------------------------NFRVYYRDSRDPVWKGPAKLLWK
                                                              *

1aboA           GQGW-------------VPSNYI--TPVN-------
1ycsB           KEGY-------------VPRNLLGLYP---------
1pht            ERGD-------------FPGTYVEYIGRKKISP---
1vie            PEGYAVESEAHPGSVQIYPVAALERIN---------
1ihvA           GEGAVVIQDNSD----------IKVVPRRKAKIIRD
                 .*                  :
(
(
1aboA:0.38808,
1ycsB:0.38385)
:0.08490,
(
1pht:0.39594,
1vie:0.44720)
:0.00848,
1ihvA:0.47811);
```