By: Eli Levy Karin, Simulation-Based Long Indel project

SimBa-SAl code documentation

This documentation requires familiarity with the paper "*A long indel model for evolutionary sequence alignment*" by Miklos, Lunter, and Holmes (2004) and "*A simulation-based approach to statistical alignment*" by Levy Karin, Ashkenazy, Hein, and Pupko (2018, to be submitted) and refers to some equations and concepts therein.

The code is composed of two programs. The second program uses the first's output as input:

1. **simba_sal_sim**: this program simulates indel events on a long ancestral sequence along a given branch length (default simulation bases the rates on the long indel model). At the end of the simulation it writes three files: (1) the estimated chop probabilities (2) the results of consistency checks concerning these probabilities, as detailed in Appendix B of MLH04, and (3) a chunk of the simulated alignment (can be used for simulation study purposes). Arguments are given as param=val pairs. Running the program with no arguments details its arguments.
2. **simba_sal_inf**: this program reads a file with a list of files containing tables of estimated chop probabilities. In addition, it reads a file with a pair of sequences (aligned/unaligned, DNA/AA). The sequences should be given in Fasta format. If the input is aligned, the program will compute its conditional probability, unalign it and compute the maximum-likelihood alignment. In case of unaligned sequences, it will compute the maximum-likelihood alignment. The program receives a list of chop tables to consider and has two optimization modes to choose from. The program allows the option to sample an alignment. It can be easily extended to use the P and X tables (see details below) to compute other interesting statistical alignment properties. In both cases, the estimation will be based on the chop probabilities and a JC/JTT process for substitution. Arguments are given as param=val pairs. Running the program with no arguments details its arguments.

Implementation details - simba_sal_sim

1. The sequence is represented as vector<size_t>, where ancestral characters are "1" and descendant characters are "2". The character "0" denotes a gap.

2. Object **rates**:
   a. **General purpose**: This object deals with the indel rate vectors (mus and lambdas) and their functionalities. It computes the vectors of the indel rates based on the type it receives (it does so only for the "N" type of chop; it receives the type in case we choose to extend in the future for some reason) and the basic rates.

   b. **Notes**:
      - The insertion rates do not include the type of character that is inserted.
      - The vectors of mus and lambdas are not infinite (unlike the theoretical model). They reach a max indel length (fixed at 50). The rate of the max indel is in fact the rate of an indel equal-to or greater-than that length.
      - The computation for type "N" is implemented according to the formulae:

$$\mu_k = \mu(1-r)^2 r^{k-1}$$

$$\lambda_k = \gamma^k \times \mu_k$$

$$\gamma^k = \frac{\lambda_k}{\mu_k}$$

These rates reflect the rate an event of a specific size starts at a specific position. At a given position, the total rate of deletion events that start at that position *d* is

$$d = \sum_{k=1}^{\infty} \mu(1-r)^2 r^{k-1} = \mu(1-r)$$

Similarly, *i*, the total rate of insertion events that start at a given position, is

$$i = \sum_{k=1}^{\infty} \mu_k \gamma^k = \sum_{k=1}^{\infty} \gamma\mu(1-r)^2(\gamma r)^{k-1} = \frac{\mu\gamma(1-r)^2}{1-\gamma r}$$

Thus, the total rate of events of any kind to start at a given position is $q = d + i$.

c. **Private methods**:
- set_rate_vectors(): see details above about the computation.
- set_rate_cumu_vecs(): cumulative rates are required for sampling events later.

d. **Public methods**:
- get_mus(): returns the vector of deletion rates mus.
- get_lambdas(): returns the vector of insertion rates lambdas.
- get_total_rate_to_change(): returns the total rate of an indel event of any size and any type (insertion / deletion) per-position. This is the sum of mus and lambdas.
- get_basic_gamma(): returns $\gamma = \frac{\lambda_1}{\mu_1}$, which is a parameter provided in the constructor.
- get_max_indel_size(): returns the maximal indel size (we work with truncated rate vectors). This parameter was provided in the constructor.
- sample_jump_time(size_t num_pos): returns a jump time based on the total rate to change per-position times the sequence length. This is based on exponential waiting time.
- sample_event_type(): returns an event type (insertion / deletion) and a size based on the relative rates.
- sample_insertion_jump_time(): returns a jump time based for an insertion event from the "immortal" link. It is called in case the evolving sequence reaches the length zero during simulation (i.e., NEVER).
- sample_insertion_size(): returns a size for an insertion event based on the relative rates. It is called in case the evolving sequence reaches the length zero during simulation (i.e., NEVER).

3. Object **simulation**:

a. **General purpose**: This object manages a single simulation along a branch. To do so, among others, it receives a rate object.

b. **Private methods**:
- reconstruct_alignment(): see details about the procedure below.
- trim_reconstructed_alignment(): see details about the procedure below.
- get_pos_and_adjust_size_del_event(...): in case of a deletion event, sample uniformly the start position and truncate the event if necessary (see details below).
- get_pos_ins_event(...): see details about the procedure below.

c. **Public methods**:
- simulate():
  The process follows the destiny of characters. Each character receives an ID in this process. A global counter keeps track of the IDs. Throughout the process, events occur to the right of a starting position.

  Initialize:
  1. Copy all N ancestral sequence characters to the evolving vector.

2. Match the evolving vector of characters with a vector of ids: 0 to (N-1).
3. Set the global ids counter to N.

Repeat:

1. Obtain the current number of positions M in the evolving sequence.
2. Sample the waiting time for some event by using the parameter (M*q). If the sampled time exceeds the branch – end the simulation.
3. Sample the event type and size according to the relative rates.
4. Sample the left-most starting position of the event uniformly along the evolving sequence.
    a. If the event is a deletion of size k – remove the starting point and (k-1) positions to its right. Remove the matching ids from the ids vector. If the event exceeds the evolving sequence, truncate the event.
    b. If the event is an insertion of size k – insert k characters right to the starting point. Insert ids in the matching positions in the ids vector. With each character insertion, raise the global counter.

At the end of the simulation we have the descendant characters as well as their matching ids. A private method reconstruct_alignment() is called. It works as follows:

Initialize:

```
curr_anc_id = 0
curr_des_index = 0
curr_des_id = desc_ids[curr_des_index]

Repeat while (not empty ancestor vector && not empty descendant vector)
If (curr_des_id >= N) // these are new characters, not in ancestor.
        des_character = des_characters[curr_des_index]
        put "-" against des_character in the alignment
        curr_des_index++
        curr_des_id = desc_ids[curr_des_index]
else
        If (curr_des_id == curr_anc_id) // homology
                anc_character = des_characters[curr_anc_id]
                des_character = des_characters[curr_des_index]
                put anc_character against des_character in the alignment
                curr_anc_id++
                curr_des_index++
                curr_des_id = desc_ids[curr_des_index]
        else // ancestor only, deleted from descendant
                anc_character = des_characters[curr_anc_id]
                put anc_character against "-" in the alignment
                curr_anc_id++
```

Take care of leftover characters if one vector is not empty…

At the end of this method there is a call to a private method trim_reconstructed_alignment() that will remove two flanking regions from each side to avoid edge problems to produce a trimmed_true_alignment that will be used for the chop estimations.

- get_true_alignment(): returns the true alignment.
- get_timmed_true_alignment(): returns the true alignment after trimming the flank regions.
- get_basic_gamma(): returns the basic gamma value (see above). It just wraps the rates object getter.

4. Object **chop_table**:

a. **General purpose**: to compute the N, L, R, and B chop tables based on a simulation object. For a detailed definition of the chop probabilities.

b. **Private methods**:
- fill_N_map(): based on the trimmed true alignment, it scans from the left. Once it finds the first matched position, the recording begins. The counters i and j will go up until the next match is identified. On that match, the recorded i,j combination will be inserted into a hash table and the counters will be reset. An N chop is a segment that ends on a match to the right, conditioned on having seen a match to the left.
- fill_L_R_B_maps(): the same trimmed true alignment is again scanned. This time in chunks. The length of each chunk is either fixed at 200 or chosen according to the equilibrium sequence length distribution (this option is currently chosen in the code). In this method, a distribution of ancestral sequence length is first computed (and trimmed at 200). Then, each time a segment is taken, the ancestral size is drawn according to this distribution. If the entire chunk does not contain a match at all, it is used to record a B chop. Otherwise, the segment from the left to the first match is used to estimate an L chop and the segment after the last match is used to record an R chop (an R chop does not end on a match on the right, it is conditioned on a match to the left).
- compute_cumu_dist_of_ancestral_lengths(): this assists in sampling chunk lengths for the fill_L_R_B_maps procedure.

c. **Public methods**: these allow getting the results of the computed tables. The actual computation occurs by calls to private methods (see details above) in the constructor.
- get_N_chops_map(): just a getter, returns a map (hash table) for N chops where the i,j pair is a key and the number of times this kind of chop was recorded is the value.
- get_L_chops_map(): just a getter
- get_R_chops_map(): just a getter
- get_B_chops_map(): just a getter
- get_total_number_of_N_chops(): just a getter
- get_total_number_of_L_chops(): just a getter
- get_total_number_of_R_chops(): just a getter
- get_total_number_of_B_chops(): just a getter
- write_tables(...): writes the tables to a file in a simple format by enumerating over all hash tables.
- write_true_trimmed_alignment(…): writes the true trimmed alignment to a file. This is useful because we can take this later and add the substitution content to that skeleton alignment to produce true alignments for testing.

5. Code **simba_sal_sim**:
   a. **General purpose**: this is not an object but rather, the code that contains the main function and manages the simulation using the above-mentioned objects. This code performs the consistency validations based on equations 30, 31, 32, 33, 34, and 35 in Appendix B of MLH04 with a modification of equations 31 and 32 (see Levy Karin et al 2018). If run without parameters, it gives information about the parameters it requires.

Implementation details - simba_sal_inf

1. Object **read_chop_tables**:
   a. **General purpose:** reads the chop tables tab delimited file and constructs 4 lookup (hash = map) tables based on that file.

   b. **Private methods**:

- read_tables_from_file()

c. **Public methods**:
- get_chop_prob(): this method looks up in the right table (according to the chop type) and returns the value for the provided i,j. It returns the conditional chop probability (this refers only to the "indel skeleton" of the chop, not to the substitution process. If the i and j combination was not find in the table, it returns a very small number. Let C be the smallest probability in one of the tables divided by 100. If i and j were not found in the table, the procedure will return C / exp(i + j + X), where X = 0 if the chop type is N, X = 5 if the chop type is L or R and X = 10 if the chop type is B.

2. Object **read_input_seqs**:
   a. **General purpose:** reads the input (aligned or unaligned), performs coding and validations of alphabet, if aligned – length validations, unalignment, and computation of the initial Gotoh-PWA for parameter optimization.

   b. **Private methods**:
   - read_seqs_from_fasta_file()
   - get_original_unaligned_seqs()
   - code_seqs(): codes characters to size_t. '0' denotes a gap, 1 - 'A', 2 - 'C', 3 - 'C', 4 - 'T' (for DNA). For AA: ARNDCQEGHILKMFPSTWYV (1 – 'A', 2 – 'R', …)
   - get_coded_unaligned_seqs()
   - transpose_to_get_alignment()
   - computeAffineNWForPair(): the Gotoh affine gap algorithm with fixed parameters: gap_open = -5 and gap_extend = -1.
   - initNucMap(): for Gotoh initial PWA
   - initAAMap(): for Gotoh initial PWA, Blosum62 is used
   - get_pair_score(): for Gotoh initial PWA, based on the map
   - estimate_branch_MP(): maximum parsimony estimation based on the Gotoh initial PWA

   c. **Public methods**:
   - get_str_orig_seq()
   - get_coded_alignment()
   - get_length_of_alignment()
   - get_str_from_coded_alignment()
   - get_length_of_anc_unaligned()
   - get_length_of_des_unaligned()
   - get_coded_anc_unaligned()
   - get_coded_des_unaligned()
   - compute_gotoh_alignment()
   - get_gotoh_coded_alignment()
   - get_estimated_branch_length_MP()

3. Object **quick_jtt**:
   a. **General purpose:** reads a file with Chebyshev coefficients to allow an approximation of the transition matrix based on JTT. The input file has 20 X 20 X 60 + 20 values. The last 20 values are the stationary probabilities of the amino acids.

   b. **Private methods**:
   - read_file_with_cheby_jtt_coef()

   c. **Public methods**:

- get_pij_t()
- freq()

4. Object **chop_prob**:
   a. **General purpose**: handles the probability computations for a specific chop zone. These include both the indel skeleton as well as the substitution process. This object is constructed based on the type of chop, i, j, the matched characters in the ancestor and in the descendant (relevant in case of an N or an L chop) and a vector of the inserted characters in the descendent (not empty if j > 0).

      Substitution process:
      Two substitution processes are supported: JC (for DNA) and JTT (for proteins). The stationary and transition probabilities for JC are straight forward. The JTT process relies on the Chebyshev approximation using the quick_jtt object.

   b. **Private methods**:
      - compute_JC_substitution_prob(): for DNA. works in log space. Computes the probability for the j inserted descendant characters and in case of an N or an L chop – the probability of a match/mismatch.
      - compute_JTT_substitution_prob(): for AA.
      - compute_chop_prob(): works in log space. Returns the probability of the chop-associated segment: indel skeleton + substitution process.

   c. **Public methods**:
      - get_chop_log_prob(): returns the total log chop zone probability. This probability is computed based on the probability of the chop indel skeleton, the probability for the match at the right end of the chop (in case of an N or an L chop) and the probability of the new descendant characters.
      - print_chop_info(): prints the details concerning the chop object and the computed probabilities. This is for debugging and such.
      - get_chop_type(): just a getter.
      - get_i(): just a getter of i, the number of deleted ancestral characters.
      - get_j(): just a getter of j, the number of inserted descendant characters.

5. Object **alignment**:
   a. **General purpose**: It analyzes a coded alignment derived from a read_input_seqs object. It will then compute the probability of this alignment given the ancestral sequence. It will do so by identifying the chops along the alignment and then using the above-mentioned objects to compute the probability of those chop. The conditional alignment (log) probability is the product (sum) of all chop-associated segment probabilities (log probabilities).

   b. **Private methods**:
      - break_alignment_to_chops(): identifying the start and end of each kind of chop is similar to what was described above for ChopTable. The identified chops are turned into chop_prob objects saved in a private vector.

   c. **Public methods**:
      - get_alignment_log_probability_cond_on_anc(): return the total alignment log probability.

6. Object **compute_alignment_dp**:
   a. **General purpose**: this object manages all interesting dynamic programming procedures. There are several purposes for dynamic programming:

- **Task I**: Compute the conditional **log** probability of the descendant sequence D having observed the ancestral sequence A. This considers all possible alignments between A and D and is described in equations in equations 14 and 15 of MLH04.
- **Task II**: Compute the most probable (maximum) alignment between A and D conditioned on having observed A. This can be achieved with a minor modification of equations 14 and 15 of MLH04 (equations 6 and 7 of Levy Karin et al 2018) so that they take the maximum contributing factor instead of summing all factors.
- **Task III**: Compute a sampled alignment between A and D according to the alignments probability conditioned on having observed A. This can be achieved with a minor modification of equations 14 and 15 of MLH04 so that they sample a factor instead of summing all factors (see Table 1 of Levy Karin et al 2018).
- **Task IV**: Compute a specific chop's "posterior" **log** probability. This means the probability the chop is included in the alignment. For that we need to compute the cumulative probabilities of all alignments including this chop and divide it by the probability computed in Task I (see Table 1 of Levy Karin et al 2018)..

The methods implemented in this object are related to the above tasks. Below are the exact procedure details. In all cases, the equations work with zero indexing (unlike MLH04).

Forward component definition:
The forward component P_j^i is the sum of all partial alignments of the first (i + 1) residues of the ancestral sequence and the first (j + 1) residues of the descendant sequence that end on a match (A_i is matched to D_j). We work in log-space so P_j^i is the **log** of the sum defined above.

Backward component definition:
The forward component X_j^i is the sum of all partial alignments of the ancestral residues (i + 1) to the end and the descendant residues (j + 1) to the end, conditioned on a match between position i and position j (A_i is matched to D_j). We work in log-space so X_j^i is the **log** of the sum defined above.

Chop "posterior" probability:
For some N chop, let us denote by i_0 the index of the ancestral position that participated in the match before the start of the chop (similarly j_0 for the descendant). Let us denote by i_1 the index of the ancestral position that participated in the match at the end of the chop (similarly j_1 for the descendant). We can then compute the chop's posterior probability as:

$$chop\ posterior\ \boldsymbol{log}\ probability = P_{j_0}^{i_0} + X_{j_1}^{i_1} + Z - log\left(Pr_t(D|A)\right)$$

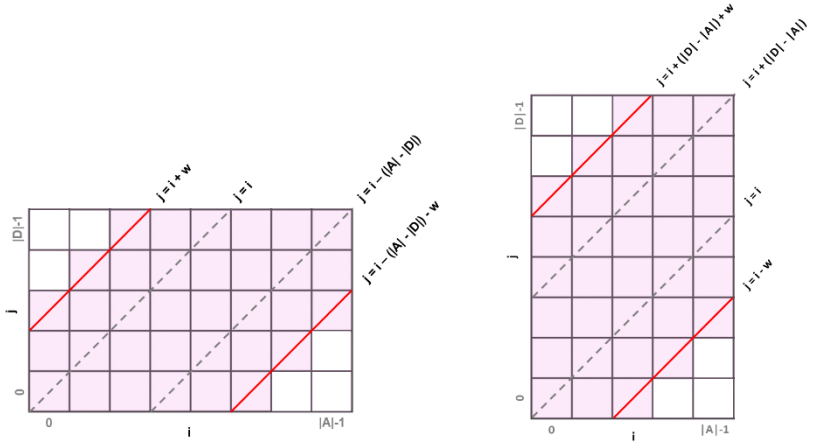
where Z is the log of the probability of the segment associated with the N chop:

$$N_{(i_1-i_0+1),(j_1-j_0+1)} \times Pr_t\left(A_{i_1} \to D_{j_1}\right) \times \prod_{k=j_0+1}^{j_1-j_0+1} q(D_k)$$

Accelerations:
The naïve complexity of the P, X and S, and Z matrices (see Table 1 in Levy Karin et al. 2018) computation is O(n^4), where *n* is an unaligned sequence length. This is because each pair (i,j) relies on computations of all previous pairs. This becomes limiting pretty fast. We allow two kinds of accelerations to save time: **chop acceleration** and **band acceleration**. In the chop acceleration, when considering the pair (i,j), we iterate over the table of N chops and compute previous entries only for the items found in the table. This way, for example, if we consider (5,6) and chop N3,3 is not found in the table, there will not be a computation of a sub-path that needs N3,3 to complete to a match of 5 and 6. This

reduces the complexity to O(V*n^2), where V is the size of the N table. This is pretty accurate – the only pitfall – if there is a chop in "nature" that was never seen in simulations – it gets probability 0. A user can take our chop tables and "thicken" them with unobserved chops with low probability. This will increase running times but will reduce the chance of missing unobserved chops.

The second type of acceleration is band-based. This limits the pairs of (i,j) to be considered (and for each such pair – only a subset of its previous pairs are within the band so another factor of saving time). The complexity depends on the width of the band. Notably, when using a very narrow band, excellent alignments can be missed and with a wide band the complexity is not that great… The band is described schematically below and relies on an equal-width band (w) around the diagonals:



The schemes depict the two possible scenarios: (1) the ancestral sequence is longer than the descendant and (2) the descendant is longer. Only pink cells are considered in the dynamic programming.

For example, for the left case, for each $0 \leq i \leq (|A| - 1)$, we consider:
$max\{0, (i - |A| + |D| - w)\} \leq j \leq min\{(|D| - 1), (i + w)\}$

If the band width is set to -1, then no corner cutting will be performed (i.e., corner cutting is optional). An approximated number of operations is computed under the band-width acceleration (even if the parameter is equal to -1) and under the chop-based acceleration. The method that yields less operations is preferred (usually, chop-based).

<u>Summing probabilities in log-space:</u>
For the P and X matrices, there is a need to sum probabilities. In order to perform this while working in log-space, the **log-sum-exp** trick was implemented in each iteration:
1. collect into a vector all log-probs (denote each ai)
2. find the max log-prob (denote aM)
3. initiate exp_sum = 0.0
4. for each ai compute:
   exp_sum += exp(ai - aM)
5. value for current table element is: aM + log(exp_sum)

<u>Sampling probabilities in log-space:</u>
The Gumbel-max trick is implemented to sample according to the probabilities. This supports sampling an alignment while still working in log-space. References for the method can be found here:

https://stats.stackexchange.com/questions/64081/how-do-i-sample-from-a-discrete-categorical-distribution-in-log-space

https://en.wikipedia.org/wiki/Categorical_distribution#Sampling_via_the_Gumbel_distribution

b. **Private methods**:

- // related to Task II and to Task III // fill_S_table_corner_cutting(bool should_take_max, int band_width): this method implements a modification of equation 6 of Levy Karin et al. 2018 to allow for path sampling (tables S and Z). If should_take_max = true, the matrix will contain information needed to compute the maximal path (maximal alignment).

  ancestral index: $0 \leq i \leq (|A| - 1)$
  descendant index: $0 \leq j \leq (|D| - 1)$

$$S_j^i = max \left\{ \begin{array}{c} L_{i,j} \times P_t(A_i \to D_j) \times \prod_{k=0}^{j-1} q(D_k), \\ max_{n,m} \left\{ S_{j-m-1}^{i-n-1} \times N_{n,m} \times P_t(A_i \to D_j) \times \prod_{k=j-m}^{j-1} q(D_k) \right\} \end{array} \right\}$$

  If j=0 then the yellow part is 1; similarly, if m=0 then, the blue part is 1.

  Replacing “max” with “sample” according to the relative contributions to the probability yields the Z matrix.

  Of note, all computations are carried out in log space using the Gumble-max trick. Furthermore, in each entry in the S matrix, we keep the chop that contributed to the path and the path probability until point (i,j). This will make it possible to traceback.

  Accelerations:
  Chop-based and band-based.

  If band_width = -1 the computation will be according to the equation above. Otherwise, only specific pairs of indices will be considered (see details below). Corner cutting relies on a data structure created by get_data_structure_of_combs_and_prev_combs (see details below).

- // related to Task I and to Task IV // fill_P_table_corner_cutting(int band_width): this method implements the **forward** components computation.

  ancestral index: $0 \leq i \leq (|A| - 1)$
  descendant index: $0 \leq j \leq (|D| - 1)$

$$P_j^i = L_{i,j} \times Pr_t(A_i \to D_j) \times \prod_{k=0}^{j-1} q(D_k) + \sum_{n=0}^{i-1} \sum_{m=0}^{j-1} P_{j-m-1}^{i-n-1} \times N_{n,m} \times Pr_t(A_i \to D_j) \times \prod_{k=j-m}^{j-1} q(D_k)$$

  If j=0 then the yellow part is 1; similarly, if m=0 then, the blue part is 1.

  Of note, all computations are carried out in log space using the **log-sum-exp** trick.

  Accelerations:

Same as S table computation.

- // related to Task I and to Task IV // fill_X_table_corner_cutting(int band_width): this method is based on the equation below to compute the **backward** components. X_j^i is the log of the total probability of the alignments of the ancestral characters $(i+1)$ to $(|A|-1)$ with the descendant characters $(j+1)$ to $(|D|-1)$ *conditional on* character A_i being matched with character D_j.

  Here, again we work with sequences indices that start at 0:
  ancestral index: $0 \le i \le (|A|-1)$
  descendant index: $0 \le j \le (|D|-1)$

  The computation begins with $i = (|A|-1)$ and $j = (|D|-1)$ and the indices decrease:

$$X_j^i = R_{|A|-i-1,|D|-j-1} \times \prod_{k=(j+1)}^{|D|-1} q(D_k) + \sum_{n=(i+1)}^{|A|-1} \sum_{m=(j+1)}^{|D|-1} X_m^n \times N_{n-i+1,m-j+1} \times Pr_t(A_n \to D_m) \times \prod_{k=j+1}^{m-1} q(D_k)$$

  If $(|B|-j-1)$=0 then the yellow part is 1; similarly, if $(m-j+1)$=0 then, the blue part is 1.

  Of note, all computations are carried out in log space using the **log-sum-exp** trick.

  Accelerations:
  Same as S table computation. Corner cutting relies on a data structure created by get_data_structure_of_combs_and_prev_backward_combs (see details below).

- // related to Task II and to Task III // get_sampled_path_last_chop(bool should_take_max): this method implements path sampling. When it is called, the S table has already been filled.

$$P(ML-PWA) = max \left\{ \begin{array}{l} B_{|A|,|D|} \times \prod_{k=0}^{|D|-1} q(D_k), \\ max_{n,m} \left\{ S_{|D|-m-1}^{|A|-n-1} \times R_{n,m} \times \prod_{k=|D|-m}^{|D|-1} q(D_k) \right\} \end{array} \right\}$$

  And again, yellow and blue – only if the descendant sequence is long enough.

  Replacing “max” with “sample” according to the relative contributions to the probability yields a sampled path instead of the ML-PWA path.

  Notably, since this function is called from fill_S_table_corner_cutting, some elements of the S table might be omitted from consideration. It skips these elements by checking their log-likelihood value (if it is positive it indicates an uninitialized element that should be skipped).

- // related to Task III // sample_index_gumbel_max(…):a helper function that receives a vector of log probabilities and samples

- // related to Task II and Task III // get_combs(…): this is a helper function to allow corner cutting (creates a general data structure of pairs to consider). It creates a data structure of the form vector<vector<size_t>>>. Each pair i and j of positions that need to be considered in the dynamic programming procedure is kept. fill_S_table_corner_cutting,

for example, will go over this vector from start to end. When considering a specific pair, it will call get_prev_combs_for_pair with the pair to consider the preceding pairs (see details above). The order of the data structure assures that when a pair is reached, all its preceding pairs were computed.

- // related to Task II and Task III // get_prev_combs_for_pair(…): this is a helper function to allow corner cutting (creates a general data structure of pairs to consider). It creates a data structure of the form vector<vector<size_t>>>. For a given i and j positions (passed as arguments), it will fill a vector of its preceding pairs (see details above).

- // related to Task II and Task III // get_backward_combs(…): this function is needed for the X table, which goes over pairs in reverse order. It first calls get_combs and then just "translates" the indices to the backward ones: backward_ind = length_of_seq – orig_ind - 1.

- // related to Task II and Task III // get_backward_prev_combs_for_pair(…): this function is needed for the X table, which goes over pairs in reverse order.

c. **Public methods**:
The object has two constructors. If it receives a file name, it will read unaligned sequences from the file. If it receives sequence vectors, it will work directly with them.

- // related to Task I // compute_conditional_alignment_total_log_prob(): this method returns $Pr_t(D|A)$ by calling fill_P_table_corner_cutting and then implementing the computation:

$$Pr_t(D|A) = B_{|A|,|D|} \times \prod_{k=0}^{|B|-1} q(D_k) + \sum_{n=0}^{|A|-1} \sum_{m=0}^{|D|-1} P_{|D|-m-1}^{|A|-n-1} \times R_{n,m} \times \prod_{k=|D|-m}^{|D|-1} q(D_k)$$

And again, yellow and blue – only if the descendant sequence is long enough.

Notably, since this function is called from fill_P_table_corner_cutting, some elements of the P table might be omitted from consideration. It skips these elements by checking their log-likelihood value (if it is positive it indicates an uninitialized element that should be skipped). It returns the **log** of the total conditional probability.

- // related to Task II and Task III // get_sampled_alignment(..., bool should_take_max): this is a traceback procedure. It calls fill_S_table_corner_cutting(should_take_max,band_width) and get_sampled_path_last_chop(should_take_max) (see details above) and then traces back the path.

The traceback procedure:
If the last chop (rightmost chop) is of type 'B' → return the "non-homologous" alignment.
Else (the rightmost chop is $R_{n,m}$):
Initialize descendant index: $j = |D| - 1$
Initialize ancestral index: $i = |A| - 1$
For (k = 0; k < m; k++)
Push to the alignment the position: ("-",D(j - k))
For (k = 0; k < n; k++)
Push to the alignment the position: (A(i - k),"-")
Update descendant index: $j = |D| - m - 1$
Update ancestral index: $i = |A| - n - 1$

While ($i \geq 0$ and $j \geq 0$)
Take the chop $C_{x,y}$ from element S_j^i in the sample table. $C_{x,y}$ can be either an N chop or an L chop.
Push to the alignment a match position (A(i),D(j))
Update descendant index: $j = j - 1$
Update ancestral index: $i = i - 1$
For (k = 0; k < y; k++)
Push to the alignment the position: ("-",D(j))
Update descendant index: $j = j - 1$
For (k = 0; k < x; k++)
Push to the alignment the position: (A(i),"-")
Update ancestral index: $i = i - 1$
If there is something left over (for example, $i \geq 0$), push what's left against a gap.
Reverse the order of the alignment and return.

- // related to Task IV // get_X_table(int band_width): If X is not computed – computes the X table and returns it. Otherwise, it just returns it.

- // related to Task IV // get_P_table(int band_width): If P is not computed – computes the P table and returns it. Otherwise, it just returns it.

7. Code **simba_sal_inf**:
   a. **General purpose**: this is not an object but rather, the code that contains the main function and manages the computation using the above-mentioned objects. If run without parameters, it gives information about the parameters it requires.