Supplementary Materials, Part 3, for the Blueprint of the "Alignment Neighborhood Explorer" (ANEX) (tentatively named),

by Kiyoshi Ezawa

(Finished on April 1st, 2019; TOC edited on August 14th, 2020)

[NOTE ADDED (2019/01/26): The "$B" in this file should be replaced with "$sub_bl" (a variable specifying the "upper-bound" of an index representing a block), which should usually (but not always) be "$B" (a user-selected parameter).]

Table of Contents

[Supplementary Figures are in "figures_sppl3_bp1_ANEX.draft8.odp".]

 # Differences from "draft8":
(1) Replaced @lb_sorted_set & @orders_lb with @blocks_w_spec_lb,
and also replaced @rb_sorted_set & @orders_rb with @blocks_w_spec_rb. (DONE on 2019/01/15)

(2) Introduced TWO ADDITIONAL categories, '<(pa)' and '>(ch)', into the relations stored in @inter_block_relations.
'<(pa)' means that $bl2 includes $bl1, and that $bl2 is effectively the "parent" of $bl1.
'>(ch)' means that $bl2 is included in $bl1, and that $bl2 is effectively a "child" of $bl1.
(DONE on 2019/01/16&17)

(3) Updated the Table of Contents. (DONE on 2019/01/26)

Supplementary Methods

SM-1. Contemplating on fast computation of the effects of simultaneous shifts of *interfering* (i.e., *non-isolated*) gap-blocks on the residue component of MSA probability

In **Appendix A5** of "**blueprint1_ANEX.draft5+.pdf**", we contemplated on how to quickly compute the effects of simultaneous shifts of *non-interfering* (i.e., *isolated*) gap-blocks on the residue component of MSA probability.
Here, let us attempt to extend the results derived there to simultaneous shifts (or shift-like moves) of gap-blocks that *interfere* with each other.

(1) First of all, when *none* of the gap-blocks *vertically overlap* any other,
the method outlined for non-interfering gap-blocks could still apply.
However, care must be taken if the gap-blocks under consideration involve all sequences in the alignment, because null (i.e., gap-only) columns will emerge if all gap-blocks overlap horizontally.
~~In such cases, we must apply the method for a pair of complementary gap-blocks (see item 2) every time when a similar situation arises, but due caution must be exercised. (In Figure SSSSA1, we will examine the consistency of such "swapping"s for 3 blocks among which any combination of the two is complementary to the remaining one.)~~
We decided NOT to swap the gap-pattern blocks EVEN WHEN seemingly complementary blocks become immediately adjacent to each other (because swapping could complicate the issue at hand as exemplified in Figures SSSSA1-SSSSA3).
Instead, we will count **the number of null columns**, as well as **the number of possible equivalent alignments** that will arise in the coordinate space under consideration.

(2) For a pair of vertically complementary gap-blocks (as in Figure 4 b of "figures_simultaneous_moves_of_multiple_blocks_METH.odp"),
as long as we stick to "shifts" (i.e., ignore "(incomplete) merges", etc.),
the columns we have to consider should be as illustrated in **Figure SSSS1**.

[Because the gap-blocks could swap their horizontal positions, it would be more convenient to consider what pairs of half-columns can result from the "shifts", rather than sticking to the positions in the (original) alignment. … Superfluous?]
Then, it becomes clear that the set of columns that must be considered is essentially the same as that for two *non-interfering* gap-blocks (as in Figure 15 of "figures&legends1_ANEX.draft3+.pdf");
more precisely, the set is actually somewhat smaller than that for non-interfering cases, because some columns can be ignored (as they are null (the middle alignment in Figure SSSS1), and because some columns are equivalent to others via the horizontal swapping of the blocks (in these cases, though, half-columns in question are always aligned with gaps) (the two alignments mediated by the double-headed arrow in Figure SSSS1).
This means that the same algorithm as non-interfering cases could be used also in this case for calculating the constituent column-wise probabilities necessary for the computation of (substitution components of) probabilities of alternative alignments(, as long as the computational efficiency does not matter significantly).
Thus, we should only modify the algorithm for calculating the alternative alignment probabilities from the column-wise probabilities (in SM-3).

NOTE added on Oct 28, 2018: as already mentioned in (1), we decided NOT to SWAP mutually complementary gap-blocks even when they become immediately adjacent; instead, we will count the number of null columns and the number of equivalent alignments that will arise in the coordinate space under consideration.

(3) When one gap-block *vertically* includes another gap-block (as in Figure 4 c & d of "figures_simultaneous_moves_of_multiple_blocks_METH.odp"),
the columns we have to consider should be as illustrated in **Figure SSSS2**.

Although the set of columns that must be considered is somewhat different from that for two *non-interfering* gap-blocks (as in Figure 15 of "figures&legends1_ANEX.draft3+.pdf"), the set can be constructed (or enumerated) in essentially the same way as for two *non-interfering* gap-blocks, that is, by considering the cases where each gap-block is on the left of, encompassing, and on the right of, the column in question, and by forming their "direct product". (When the vertically larger block is encompassing the column, however, the position of the vertically smaller block can be ignored.)

(4) When two gap-blocks are identical in vertical range to each other (as in Figure 4 e of "figures_simultaneous_moves_of_multiple_blocks_METH.odp"),
the columns we have to consider should be as illustrated in **Figure SSSS3**.

Although the set of columns that must be considered is somewhat different from that for two *non-interfering* gap-blocks (as in Figure 15 of "figures&legends1_ANEX.draft3+.pdf"), the set can be constructed (or enumerated) in essentially the same way as for two *non-interfering* gap-blocks, that is, by considering the cases where each gap-block is on the left of, encompassing, and on the right of, the column in question, and by forming their "direct product". (When either of the two blocks is encompassing the column, however, the position of the other block can be ignored.)

(5) When two gap-blocks *vertically* overlap each other but neither of them includes the other (as in Figure 4 f & g of "figures_simultaneous_moves_of_multiple_blocks_METH.odp"),
the columns we have to consider should be as illustrated in **Figure SSSS4**.

NOTE Added on Oct 30, 2018: The swapping of vertically overlapping, non-nesting blocks seems to remain consistent with the "shift"s of other gap-blocks that are in various vertical relationships with the former two, as long as the sets of columns are defined as spanning the respective sets of columns. (**Figures SSSSA4-SSSSA7** provide some example cases.) Therefore, **we will perform the swapping in this case**, as already considered in "simultaneous_moves_of_multiple_blocks_METH_xxxx.odp" (Item 10 of section (ii) of Window analysis [2]). Thus, a swapped configuration should NOT result from different "double-shifts" represented as different pairs of coordinates.

This case is very similar to case (2) (in Figure SSSS1), and thus the set of columns to be considered could be enumerated in essentially the same way, that is, by considering the cases where each gap-block is on the left of, encompassing, and on the right of, the column in question, and by forming their "direct product", while ignoring the case where both blocks encompass the column.

[Summary] Having considered 5 different cases, I propose **the following algorithm** to calculate the column-wise probabilities (on residue configurations) of the columns that will constitute the alternative alignments created by shifts (or by shifting-like moves) of the gap-blocks (in the input alignment).
(NOTE: The alignment columns are supposed to be numbered as 0, 1, …, from the leftmost to the right.)

1. Classify the sequences according to what gap-blocks affect them.
For example, let us consider a local alignment with three gap-blocks.
If a sequence is affected by all gap-blocks, classify it as "**TTT**";
if a sequence is not affected by any gap-block, classify it as "**FFF**";
if a sequence is affected only by the 2nd gap-block, classify it as "**FTF**"; etc.

2. Group together the sequences classified identically. The sequences of each class will always move together as the gap-blocks move.
(For example, sequences of class "TTT" will move in response to the move of any of the gap-blocks, but sequences of class "FFF" will never move no matter what gap-block moves.)

3. For each class, assign successive numbers, 0, 1, …, to the "**class-specific columns**", that is, the columns restricted to (or projected onto) the class each of which contains at least one residue. (Let us refer to the numbers as "**class-specific column numbers**".)

4. Examine whether each class is monophyletic, the complement of monophyletic, or neither of them.
=> If monophyletic or complement-monophyletic, identify the separating branch & identify a column in which the class is occupied solely by gaps. (=> This will enable the use of pre-computed partial probabilities.)

5. For each column (of the input alignment), calculate the probabilities of the columns resulting from positioning each of the gap-blocks on its left (L), just on it (O), and on its right (R). Graphically, the set of the considered columns could be represented as a direct product:

{L, O, R}_block1 × {L, O, R}_block2 × {L, O, R}_block3 ….

(a) When a block is just on the column, the affected class-specific columns will be occupied solely by the gaps, as expected.

(b) When ALL blocks are on the right of the column, the column should consists of the class-specific columns with: {class-specific column number} = {column number}. (If any of such class-specific columns are lacking, skip the calculation of their probabilities.)

(c) When no affecting block is just on the column, but some affecting blocks are on the left of the column, the class-specific column should have:
{class-specific column number}
= {column number} - $\sum_{\{\text{affecting gap-blocks on the left}\}}$ {horizontal block-size}. (If any of such class-specific columns are lacking, skip the calculation of their probabilities.)

Putting together the rules (a), (b) and (c) above, **an efficient algorithm** to calculate column-wise blocks should be, for example, as follows.

$CT_CLMS; # #{columns in the local alignment}
$ct_blocks; # #{blocks in the local alignment}
@set_blsizes; # $set_blsizes[$b] is the horizontal size of the $b th block.
@class_labels = ('TTT', 'TTF', 'TFT', 'TFF', …., 'FFF'); # enumerate all non-empty classes. #
$ct_classes = @class_labels; #{classes}

%label2class = ($class_label => \@indices_seqs_in_class, ….); # For ALL conceivable all non-empty **(revised on Nov 24, 2018)** classes. @indices_seqs_in_class = () if the class is empty.

#%label2monophyl = ($class_label => $separating_branch, …); # For monophyletic classes.
#%label2compl_monophyl = ($class_label => $separating_branch, …); # For complementary-monophyletic classes. **… OBSOLETE as of Nov 22, 2018.**
@set_cw_monophyl_roots, where **@{$set_cw_monophyl_roots[$cl]}** lists the roots of the monophyletic groups belonging to the $cl th class (= empty if the $cl th class contains no monophyletic group).
$complement_lower_bound, which is the "lower-bound" (or separating branch) of the complement monophyletic group (= undef if there is no such complement).
$cl_w_cmpl_mp, which is the ID of the class accommodating a complement monophyletic group (= undef if there is no such complement).
… These will be used in APPENDIX B (… ADDED on Nov 22, 2018.)

%label2class_sp_clms = ($class_label => \@set_class_sp_clms, …), where
@set_class_sp_clms[$i] is the (overall) column-number of the $i th class-specific column.
%label2ct_class_sp_clms = ($class_label => #class_sp_clms, …); # For non-empty classes.

@affected_classes = (\@classes_affected_by_block1, \@classes_affected_by_block2, …); # Just for convenience. NOTE that \@classes_affected_by_blocki contains the indices of the relevant classes in @class_labels.

@list_considered_columns = (); # element = \@set_of_class-specific_column_numbers (or '-'s for gaps).
%already; # = ($colon_concatenated_set_of_class-specific_column_numbers => 1, ...)

for (my $c = 0; $c < $CT_CLMS; $c++) { # Outermost for. #

```
    my @csd_clm = ();
    for (1 .. $ct_classes) { push @csd_clm, $c; } # initialize the column to be considered.

    my @minilist_csd_clms = (\@csd_clm); # Initialize the minilist of the columns to be considered.

    for (my $b = 0; $b < $ct_blocks; $b++) { # Outer for. #

        my $blsize = $set_blsizes[$b];
        my $classes_afd = $affected_classes[$b]; # Probably superfluous, but for convenience. #

        my @new_minilist_csd_clms = (); # New minilist. #
        while  (my $csd_clm = shift @minilist_csd_clms) {# Middle while. #

                    # When the block is on the right of the column (R). #
                push @new_minilist_csd_clms, $csd_clm;

                    # When the block is just on the column (O). #
                my @cp1_csd_clm = @{$csd_clm}; # Copy the column to be considered.
                foreach my $indx_cls (@{$classes_afd}) { # Inner for (1). #
                    $cp1_cds_clm[$indx_cls] = '-'; # Replace the class-specific column number with '-' (a gap). #
                } # End of the inner for (1). #
                push @new_minilist_csd_clms, \@cp1_csd_clm;

                    # When the block is on the left of the column (R). #
                my @cp2_csd_clm = @{$csd_clm}; # Copy the column to be considered.
                foreach my $indx_cls (@{$classes_afd}) { # Inner for (2). #
                    $cp2_csd_clm[$indx_cls] -= $blsize; # Subtract the block size. #
                } # End of the inner for (2). #
                push @new_minilist_csd_clms, \@cp2_csd_clm;

        } # End of the middle while.

        @minilist_csd_clms = @new_minilist_csd_clms; # Update the minilist. #

    } # End of the outer for. #

                # Examine the integrity & redundancy of the columns. #

    while (my $csd_clm = shift @minilist_csd_clms) { # Outer while (2). #

        my $fl_safe = 1;
        for (my $cl=0; $cl<$ct_classes; $cl++) {
            my $indx_cl_sp_clm = $csd_clm→[$cl];

            ($indx_cl_sp_clm eq '-') and next;  # If it's a gap, there is no problem. #
```

```
            my $ct_cl_sp_clms = $label2ct_cl_sp_clms{$class_labels[$cl]};
            if ( ($indx_cl_sp_clm < 0) or ($ct_cl_sp_clms <= $indx_cl_sp_clm) ) { # The index is
out of the range. #
                $fl_safe = 0;
                last;
            }
        }
        $fl_safe or next; # The column cannot be made from the ingredient at hand. #

        my $cnct_set_cls_sp_clm_nos = join (':', @{$csd_clm});
        if (defined $already{$cnct_set_cls_sp_clm_nos}) { next; } # The column is already in the
list. #

        push @list_csd_clms, $csd_clm;
        $already{$cnct_set_cls_sp_clm_nos} = 1;

    }   # End of the outer while (2). #

}   # End of the outermost for. #
```

=> Calculate the column-wise probability of each element of @list_csd_clms,
using a modified version of Felsenstein's pruning (or peeling) algorithm. (See **Appendix B**.)
(
Or, if you prefer, the calculation could be performed before the command:
$already{$cnct_set_cls_sp_clm_nos} = 1;,
and replace the command with:
$clm2prob{$cnct_set_cls_sp_clm_nos} = $prob_clm_to_be_examined;
)

In any case, we must construct a hash:
%clm2prob = ($cnct_set_cls_sp_clm_nos => $prob_clm_to_be_examined, …),
as a final product of this algorithm.

SM-2. Actually implementing algorithm to quickly compute the effects of simultaneous shifts of *non-interfering* (i.e., *isolated*) gap-blocks on the residue component of MSA probability

In **Appendix A5** of "**blueprint1_ANEX.draft5+.pdf**", we contemplated on how to quickly compute the effects of simultaneous shifts of *non-interfering* (i.e., *isolated*) gap-blocks on the residue component of MSA probability.
Here, let us attempt to implement the resulting method as an **actual algorithm**.

1. Use some variables used in the algorithm in item 5 of [Summary] of SM-1.

2. Prepare **a B-dimensional array** (where B = #{blocks} (= $ct_blocks))**, @incr_probs**, as a "container" of the calculated probabilities (more precisely, increments of the probability compared to that for the input (or "reference") alignment).
The array element, **$incr_probs[$k$_1$]->[k_2$]...[$k$_B$]** (with k_1$, k_2$, …, k_B$ = 0, 1, 2, …, or $2W_M$ $bds_bl_coords[$bl]→[1] … MODIFIED on Dec 24, 2018), should represent the increment of the probability for the alignment resulting from the "shift"s of the 1st block by k_1$ – W_M $org_bl_coords[0] columns, the 2nd block by k_2$ – W_M $org_bl_coords[1] columns, …, and the B-th block by k_B$ – W_M $org_bl_coords[B-1] columns. (A negative integer indicates the move to the left by its absolute value. A positive integer indicates the move to the right.)
By definition, **$incr_probs[$k$_1$=W_M $org_bl_coords[0]]→[$k$_2$=W_M $org_bl_coords[1]]...[$k$_B$=W_M $org_bl_coords[B-1]] = 0.** **(NOTE ADDED on Dec 24, 2018: The origin was changed from W_M(independent of the block) to $org_bl_coords[$bl] for the $bl th block.)**

3. From the input alignment, create **a set of columns** each of which is represented as a set of class-specific columns, and record **the positions of the blocks** in it (in terms of the (overall) column numbers at their left- and right-boundaries).

@set_columns, where **$set_columns[$c]→[$cls] =** {the ID number (or index) of the class-specific column that occupies the $cls th class at the $c th overall column}. It is initialized by using the input alignment.

@bds_blocks, where **@{$bds_blocks→[$bl]} = ($left_bd, $right_bd)** is for the $bl th block, with $left_bd and $right_bd being the column numbers at the left- and right-boundaries of the block.

4. Compute the probability increments as an accumulation of the increments by single-column-moves of the blocks, while gradually modifying @set_columns and @bds_blocks from their initial values.

{Recursive version}
Let us first consider a recursive algorithm, which is generally simpler.
Then, we will attempt to remove the recursion.

```
our $B = $ct_blocks;
our $Wm = {the maximum width of a block move};
our @incr_probs = initialize_incr_probs ();

my @ord_bl_coords = (The set of "origins" for the block coordinates);
        # $org_bl_coords[$bl] is the "origin" for the coordinate of the $bl th block. (ADDED on Dec 24, 2018.)

my @bds_bl_coords = (The set of boundaries for the block coordinates);
      #  @{$bds_bl_coords[$bl]} = ($lb_coord, $rb_coord)  in the full-closed convention;

# my @init_bl_coords = ($k_1 = $Wm, $k_2 = $Wm, …, $k_B = $Wm);
my @init_bl_coords = @org_bl_coords; # MODIFIED on Dec 24, 2018.
my @init_set_columns = (The set of columns (each represented by a set of constituent class-specific columns) that constitute the initial local alignment);
my @init_bds_blocks = (The set of block boundaries for the initial local alignment);

my $init_ct_null_clms = #{null columns in the initial local alignment}; … ADDED on Nov 13, 2018.

recur_comput_prob_incre_by_bl_mv (@init_set_columns, @init_bds_blocks, @init_bl_coords, $init_ct_null_clms, @bds_bl_coords, 0); # The call of the function in the main(?) program. #

sub recur_compt_prob_incr_by_bl_mv (@@@$@$) { # The DEFINITION. #

        my ($prev_set_columns, $prev_bds_blocks, $prev_bl_coords, $prev_ct_null_clms, $bds_bl_coords, $bl) = @_;
        # $bl is the index of the block that will be "shift"ed in this function call.
        # @{$prev_bl_coords} = ($k_1, $k_2, …, $k_B) specified in the previous call.

                ## First, self-call the function before moving the $bl th block at all.
                ## (This is necessary for exhaustively exploring the coordinate space.) #

        if ($bl < $B-1) { recur_compt_prob_incr_by_bl_mv (@{$prev_set_columns}, @{$prev_bds_blocks}, @{$prev_bl_coords}, $prev_ct_null_clms, @{$bds_bl_coords}, $bl+1);  }

                # Retrieve the boundaries of the block coordinates.
        my ($lbd_bl_coord, $rbd_bl_coord) = @{$bds_bl_coords→[$bl]};
```

```
            # Retrieve the base of the probability increments,
            # which will be necessary for the following computations. #

        my ($k_10, $k_20, …, $k_B0) = @{$prev_bl_coords};
        my $base_prb_incr =  $incr_probs[$k_10]→[$k_20]....[$k_B0];

            ## Second, shift the block to the right. ##
            # Initialize variables. #

        my @curr_set_columns = copy (@{$prev_set_columns});
        my @curr_bds_blocks = copy (@{$prev_bds_blocks});
        my @curr_bl_coords = @{$prev_bl_coords};
        my $prb_incr = $base_prb_incr;
        my $curr_ct_null_clms = $prev_ct_null_clms; … ADDED on Nov 13, 2018.

        for (my $i = $org_bl_coords[$bl] ; $i < $rbd_bl_coord; $i++) { # 1st outer for-loop.
(modified on Dec 24, 2018)
        # for (my $i = $Wm ; $i < $rbd_bl_coord; $i++) { # 1st outer for-loop. # OBSOLETE as of
2018/12/24.

            my ($flag, $incr_incr, $incr_ct_null_clms) = shift_bl_and_compt_prob_incr
(@curr_set_columns, @curr_bds_blocks, @curr_bl_coords, $bl, +1);

            my ($k_1, $k_2, …, $k_B) = @curr_bl_coords;

        if ((defined $prb_incr) and (defined $incr_incr)) {
            $prb_incr += $incr_incr;
            $incr_probs[$k_1]→[$k_2]....[$k_B] = $prb_incr;
            if (defined $curr_ct_null_clms) { $curr_ct_null_clms += $incr_ct_null_clms; }

        } elsif ($flag <= 0) { # Hereafter, actually, we MUST prevent it from exploring
theoretically inaccessible regions. (An IMPORTANT homework)

            $prb_incr = - $prb_laln0;  # Minus the probability of the initial local alignment. #
            $curr_ct_null_clms = 0;
            foreach my $clm (@curr_set_columns) {  # Inner foreach-loop.
                my $cnct_clm = join (':', @{$clm});
                 my $cw_prb = $clm2prob{$cnct_clm};
                 if (defined $cw_prb) {
                    $prb_incr += $cw_prb;
                    if ($cnct_clm eq $cnct_null_clm) { $curr_ct_null_clms++; }
                  } else {
                    $prb_incr = undef;
                    $curr_ct_null_clms = undef;
                    last;
                 }
            }        # End of the inner foreach-loop. #

            $incr_probs[$k_1]→[$k_2]....[$k_B] = $prb_incr;

        } else {
            $incr_probs[$k_1]→[$k_2]....[$k_B] = undef;
            $curr_ct_null_clms = undef;
        }

            if ($bl < $B-1) { recur_compt_prob_incr_by_bl_mv (@curr_set_columns,
@curr_bds_blocks, @curr_bl_coords, $curr_ct_null_clms, @{$bds_bl_coords}, $bl+1);  }
```

```
	}	# End of the 1st outer for-loop.

	## Third, shift the block to the left. ##
	# RE-initialize the variables. #

	@curr_set_columns = copy (@{$prev_set_columns});
	@curr_bds_blocks = copy (@{$prev_bds_blocks});
	@curr_bl_coords = @{$prev_bl_coords};
	$prb_incr =  $base_prb_incr;
	$curr_ct_null_clms = $prev_ct_null_clms; … ADDED on Nov 13, 2018.

	for (my $i = $org_bl_coords[$bl] ; $i > $lbd_bl_coord; $i--) {  # 2nd outer for-loop.
(modified on Dec 24, 2018)
	# for (my $i = $Wm ; $i > $lbd_bl_coord; $i--) {  # 2nd outer for-loop. # OBSOLETE as of
2018/12/24.

		my ($flag, $incr_incr, $incr_ ct_null_clms) = shift_bl_and_compt_prob_incr
(@curr_set_columns, @curr_bds_blocks, @curr_bl_coords, $bl, -1);

		my ($k_1, $k_2, …, $k_B) = @curr_bl_coords;

	    if ((defined $prb_incr) and (defined $incr_incr)) {
		$prb_incr += $incr_incr;
		$incr_probs[$k_1]→[$k_2]....[$k_B] = $prb_incr;
		if (defined $curr_ct_null_clms) { $curr_ct_null_clms += $incr_ct_null_clms; }

	    } elsif ($flag <= 0) { # Hereafter, actually, we MUST prevent it from exploring
theoretically inaccessible regions. (An IMPORTANT homework)

		$prb_incr = - $prb_laln0;  # Minus the probability of the initial local alignment. #
		$curr_ct_null_clms = 0;
		foreach my $clm (@curr_set_columns) {  # Inner foreach-loop.
		    my $cnct_clm = join (':', @{$clm});
		     my $cw_prb = $clm2prob{$cnct_clm};
		     if (defined $cw_prb) {
			$prb_incr += $cw_prb;
			if ($cnct_clm eq $cnct_null_clm) { $curr_ct_null_clms++; }
		     } else {
			$prb_incr = undef;
			$curr_ct_null_clms = undef;
			last;
		     }
		}	# End of the inner foreach-loop. #

		$incr_probs[$k_1]→[$k_2]....[$k_B] = $prb_incr;

	    } else {
		$incr_probs[$k_1]→[$k_2]....[$k_B] = undef;
		$curr_ct_null_clms = undef;
	    }

		if ($bl < $B-1) { recur_compt_prob_incr_by_bl_mv (@curr_set_columns,
@curr_bds_blocks, @curr_bl_coords, $curr_ct_null_clms, @{$bds_bl_coords}, $bl+1);  }

	} # END of the 2nd outer for-loop.

	return 1;
```

} # END of the DEFINITION of "**sub recur_compt_prob_incr_by_bl_mv (@@@$@$)** {...}".

```
	# The following satellite subroutine of "recur_compt_prob_incr_by_bl_mv (@@@$@$)"
	# actually shifts the $bl th block by $sh columns
	# (1 column to the right if $sh = +1, 1 column to the left if $sh = -1).  ($sh must be either +1 or -1).
	 # More precisely, it modifies @curr_set_columns , @curr_bds_blocks and @curr_bl_coords accordingly, while extracting the changes in @curr_set_columns.
	# Then, it computes the probability increment ($incr_incr) according to the changes in @curr_set_columns.
	# It returns ($flag, $incr_incr, $incr_ct_null_clms), with
	# $flag = 0 if the computation succeeded,
	#	= +1 if the probabilities of columns after the move cannot be done,
	#	= -1 if the probabilities of columns before the move cannot be done.
	# $incr_ct_null_clms is the increment of #{null columns in the local alignment}.

sub shift_bl_and_compt_prob_incr (@@@$$) {  # The DEFINITION. #

	my ($curr_set_columns, $curr_bds_blocks, $curr_bl_coords, $bl, $sh) = @_;

	my $prev_bl_coord = $curr_bl_coords->[$bl]; # Keep the previous coordinate.
	my $curr_bl_coord = ($curr_bl_coords->[$bl] += $sh);   # Shift the current coordinate.

	my $relv_bds_block = $curr_bds_blocks→[$bl];
	my ($prev_lbd, $prev_rbd) = @{$relv_bds_block};    # Keep the  relevant block boundaries before the move.

		## (1) Actually shift the $bl th block one column to the right.  ##
		## (In the current case, its very simple, because the blocks do NOT interfere with one another.) => Will be Sophisticated later.

		# Modify @curr_bds_blocks.
	my $curr_lbd = ($relv_bds_block→[0] += $sh);
	my $curr_rbd = ($relv_bds_block→[1] += $sh);

		# Keep the columns before being modified. #
		# (See Figure SSSS5 a.)

	my @rlv_clm_nos = ($sh>0) ? ($prev_lbd, $curr_rbd) : ($curr_lbd, $prev_rbd);
	my ($rlv_clm1, $rlv_clm2) =  my @tmp = @{$curr_set_columns}[@rlv_clm_nos];

	my @cp_rlv_clm1 = @{$rlv_clm1};
	my @cp_rlv_clm2 = @{$rlv_clm2};
	my @columns_bf = (\@cp_rlv_clm1, \@cp_rlv_clm2);

		# Modify @curr_set_columns.
		#  More precisely, swap the portions of the relevant columns affected by the relevant block.

	my $indices_aff_classes = $affected_classes[$bl];

	foreach my $indx_ac (@{$indices_aff_classes}) { # 1st foreach-loop. #
		my $tmp = $rlv_clm1→[$indx_ac];
		$rlv_clm1→[$indx_ac] = $rlv_clm2→[$indx_ac];
		$rlv_clm2→[$indx_ac] = $tmp;
	}	# End of the 1st foreach-loop. #
```

```
		# Extract the columns after being modified. #

	#my @columns_af = @{$curr_set_columns}[@rlv_clm_nos];
	my @columns_af = ($rlv_clm1, $rlv_clm2);

		## (2) Compute the probability increment ($incr_incr) according to the changes in
@curr_set_columns);

	my $incr_incr = 0;
	my $incr_ct_null_clms = 0; ... ADDED on Nov 13, 2018.
	foreach my $clm (@columns_af) { # 2nd foreach-loop.
		my $cnct_clm = join (':', @{$clm});
		my $cw_prob = $clm2prob{$cnct_clm};
		(defined $cw_prob) or (return (+1, undef, undef));
		$incr_incr += $cw_prob;
		if ($cnct_clm eq $cnct_null_clm) { $incr_ct_null_clms++; }
	}		# End of the 2nd foreach-loop.

	foreach my $clm (@columns_bf) {  # 3rd foreach-loop.
		my $cnct_clm = join (':', @{$clm});
		my $cw_prob = $clm2prob{$cnct_clm};
		(defined $cw_prob) or (return (-1, undef, undef));
		$incr_incr -= $cw_prob;
		if ($cnct_clm eq $cnct_null_clm) { $incr_ct_null_clms--; }
	}		# End of the 3rd foreach-loop.

	return (0, $incr_incr, $incr_ct_null_clms);

} # END of The DEFINITION of "sub shift_bl_and_compt_prob_incr (@@@$$) {...}"
```

{Removal of Recursion}

The self-recursion function, "**recur_compt_prob_incr_by_bl_mv (@@@@$)** {...}",
is recursive simply in order to implement the multiple shifts systematically.
Here,
utilizing a stack and *systematic* branching conditions,
we will remove the recursion.

What must be stacked is a **triple quadruple (or 4-tuple)**: **(… REVISED on Nov 13, 2018)**
[\@set_columns, \@bds_blocks, \@bl_coords, $ct_null_clms].

However, its **initial value**:
[\@set_columns0, \@bds_blocks0, \@bl_coords0, $ct_null_clms0],
must remain unchanged throughout the algorithm; so, it may not necessarily be in the stack.

The simplest way to remove the recursion would be
to exhaust the block coordinate values in a serial manner, like:
$k_1 = $k_1_lbd, …, $Wm $org_bl_coords[0] (modified on Dec 24, 2018), …, $k_1_rbd,
then $k_2 = $k_2_lbd, …, $Wm $org_bl_coords[1] (modified on Dec 24, 2018), …, $k_2_rbd,
…, and finally, $k_B = $k_B_lbd, …, $Wm $org_bl_coords[B-1] (modified on Dec 24, 2018), …,
$k_B_rbd,
while stacking all resulting triples quadruples along the way.

However, such an algorithm is **not practical**, because it consumes *extremely* large memory-space.

(Roughly speaking, the number of triples quadruples that must be stored will be at most: ($k_1_rbd - $k_1_lbd +1) * ($k_2_rbd - $k_2_lbd +1) * … * ($k_B_rbd - $k_B_lbd +1).)

Here, we will devise an algorithm that needs to stack at most (B+1) triples quadruples.

To do this, we mimic the **recursive algorithm** in terms of the order of the block shifts.
More precisely, the block shifts will follow the following **rules**:

(i) In addition to the initial triple quadruple, prepare one triple quadruple for each block;
(ii) Start at the "Origin", **@bl_coords0 = ($k_1 = $Wm, $k_2 = $Wm, …, $k_B = $Wm) @org_bl_coords (MODIFIED on Dec 24, 2018)**;

(iii) First, for $bl = B-1, explore $k_B = $Wm+1 $org_bl_coords[B-1]+1, …, $rb_k_B, then, $k_B = $Wm-1 $org_bl_coords[B-1]-1, …, $lb_k_B, while keeping ($k_1, .., $k_{B-1}) = ($Wm, …, $Wm) **@org_bl_coords[0..B-2] (MODIFIED on Dec 24, 2018)**;

(iv) Then, for $bl = B-2, move $k_{B-1} from $Wm+1 $org_bl_coords[B-2]+1 to $rb_k_{B-1}, one column at a time, and (after evaluating the probability for $k_B = $Wm $org_bl_coords[B-1],) explore $k_B as in (iii) for each value of $k_{B-1};
(v) Then, again for $bl = B-2, move $k_{B-1} from $Wm-1 $org_bl_coords[B-2]-1 to $lb_k_{B-1}, one column at a time, and, again, (after evaluating the probability for $k_B = $Wm $org_bl_coords[B-2],) explore $k_B as in (iii) for each value of $k_{B-1};

(vi) Then, for $bl = $B-3, move $k_{B-2} from $Wm+1$org_bl_coords[B-3]+1 to $rb_k_{B-2}, one column at a time, and (after evaluating the probability for $k_{B-1} =$k_B = $Wm ($k_{B-1} , $k_B) = ($org_bl_coords[B-2], $org_bl_coords[B-1]),) explore ($k_{B-1}, $k_B) as in (iii)-(v) for each value of $k_{B-2};
(vii) Then, again for $bl = $B-3, move $k_{B-2} from $Wm-1 $org_bl_coords[B-3]-1 to $lb_k_{B-2}, and do the rest similarly to (vi);

(viii) Repeat the procedures like (iii)-(vii) similarly for $bl = B-4, …, 1, 0.

(ix) When we finish the move to ($k_1, $k_2, …, $kB) = ($lb_k_1, $lb_k_2, …, $lb_k_B) and the computation of the probability at that point, leave the loop.

When attempting to implement this system of block-moves for exploring the coordinate space, the key element is:

How the **"vertical move"** (i.e, the change in $bl, i.e., the block to be shifted) will be determined.

On the other hand, the "horizontal move" of each block should be almost obvious from the value of its coordinate (i.e., $bl).
However, when the block is at the center (i.e., $bl_coords→[$bl] = $Wm $org_bl_coords[$bl]), there could be ambiguity as to whether to move the block to the left or to the right.
So, prepare the following ($B-dimensional) array:
@horizontal_shifts, with
$bl_shifts[$bl] = +1 when the $bl th block must move to the right,
$bl_shifts[$bl] = -1 when the $bl th block must move to the left.
(The initial value is: @bl_shifts = (+1, +1, …, +1).)

Then, the moves in the coordinate space will be determined uniquely, as follows:

(a) First of all, move the $bl th block by one column (to the left or right) if it remains within the range, second, compute the corresponding block boundaries and the set of columns, as well as the probability increment and the increment of the count of null columns, then, **raise the $bl all the**

way to $B -1, while stuffing the stack with ($B – 1 - $bl) copies of the newly computed triple (and updating $bl_shifts[$bl+1] = … = $bl_shifts[$B-1] = +1, and $bl_coords … Superfluous!!);

(b) If the $bl th block moves over the **right-boundary**, discard the working triple quadruple and copy the top triple quadruple remaining in the stack (and make it as the new working triple quadruple), **return to the center ($bl_coords->[$bl] = $Wm $org_bl_coords[$bl] (modified on Dec 24, 2018)) (… actually, automatically done by updating the triple quadruple) and update $bl_shifts[$bl] = -1**, and go next;

(c) If the $bl (>0) th block moves over the **left-boundary**, discard the working triple quadruple and update it to the top triple quadruple remaining in the stack, (**update $bl_shifts[$bl] = +1, and $bl_coords→[$bl] = $Wm … actually, automatically done by updating the triple quadruple**), **go down to the ($bl-1) th block**, and go next;

(d) If the $bl = 0 th block moves over the left-boundary, **leave the loop** (i.e., end the whole computation).

The above moves are illustrated in Figure SSSS6 (in the simplest non-trivial case of B=3).
The **actual (pseudo-)cord** is as follows:

```
our $B = $ct_blocks;
our $Wm = {the maximum width of a block move}; # NOTE added on Dec 24, 2018: In some cases, the block may move more than this value.
our @incr_probs = initialize_incr_probs ();
our @org_bl_coords; # The "origins" of the block coordinates, with $org_bl_coords[$bl] gives the "origin" of the coordinate of the $bl th block. (ADDED on Dec 24, 2018.)
our @bds_bl_coords; # The boundaries of the block coordinates, with @{$bds_bl_coords[$bl]} = ($lbd_bl_coord, $rbd_bl_coord) gives the boundaries (in the full-closed convention) of the coordinate of the $bl th block.

# Assume that @set_columns0 and @bds_blocks0 are given.
#
my @bl_coords0 = my @bl_shifts = ();
for (1 .. $B) { push @bl_coords0, $Wm; push @bl_shifts, +1; }
my @bl_coords0 = @org_bl_coords;  # MODIFIED on Dec 24, 2018.

my $prb_laln0 = prob_laln (@set_columns0) ; # Actually, prob_laln () just sums column-wise log-probabilities for all columns.
my $ct_null_clms0 = #{null columns in the initial local alignment}; … ADDED on Nov 13, 2018.

my @triple0 = (\@set_columns0, \@bds_blocks0, \@bl_coords0, $ct_null_clms0);  # Initial value of the triple quadruple (i.e., 4-tuple).

my @stack = ();
for (0 .. $B) { my @triple = copy (@triple0) ; push @stack, \@triple; } # NOTE: Now, the @triple is actually a quadruple.

my $wrk_triple = pop @stack; # Working triple.
my $bl = $B-1;  # Start with the right-most lowest-ranked block. #
$incr_probs[$coord_b1 = $Wm $org_bl_coords[0]]→[$coord_b2 = $Wm $org_bl_coords[1]]→...→[$coord_bB = $Wm $org_bl_coords[B-1]] = 0;  # Modified on Dec 24, 2018.

while (1) {  # Outer while-loop. #

    my ($set_columns, $bds_blocks, $bl_coords, $ct_null_clms) = @{$wrk_triple};

    my $sh = $bl_shifts[$bl];
```

```
        my $bl_coord = $bl_coords->[$bl];
        my $new_bl_coord = $bl_coord + $sh;
        my ($lbd_bl_coord, $rbd_bl_coord) = @{$bds_bl_coords[$bl]};

        if ($rbd_bl_coord < $new_bl_coord) {

                # Stepping over the right-boundary. => (b)

            my @copy = copy (@{$stack[$#stack]});
            $wrk_triple = \@copy;  # This automatically includes $bl_coords→[$bl] = …
$bl_coords→[$B-1] = $Wm  @bl_coords[$bl .. $B-1] = @org_bl_coords[$bl .. $B-1]  (MODIFIED
on Dec 24, 2018).
            $bl_shifts[$bl] = -1;
            next;

        } elsif ($new_bl_coord < $lbd_bl_coord) {

                # Stepping over the left-boundary.

            ($bl == 0) and last; # (d)

                # (c)
            $wrk_triple = pop @stack;  # This automatically includes $bl_coords→[$bl] = …
$bl_ coords→[$B-1] = $Wm  @bl_coords[$bl .. $B-1] = @org_bl_coords[$bl .. $B-1]
(MODIFIED on Dec 24, 2018).
            $bl_shifts[$bl] = +1; # Maybe superfluous. … currently, not. #
            $bl--;
            next;
        }

            # Within the range. => (a)

        my ($set_columns, $bds_blocks, $bl_coords, $ct_null_clms) = @{$wrk_triple};
        my ($k_10, $k_20, …, $k_B0) = @{$bl_coords};
        my $prb_incr = $incr_probs[$k_10]→[$k_20]→...->[$k_B0];

            # Execute the block shift.
            # Compute (or update) the block coordinates, the block boundaries,
            # the set of columns, and the probability increments.
            #
        my ($flag, $incr_incr, $incr_ct_null_clms) = shift_bl_and_compt_prob_incr
(@{$set_columns}, @{$bds_blocks}, @{$bl_coords}, $bl, $sh);  # Shift the $bl th block by $sh.

        my ($k_1, $k_2, …, $k_B) = @{$bl_coords};

        if ((defined $prb_incr) and (defined $incr_incr)) {

            $prb_incr += $incr_incr;
            $incr_probs[$k_1]→[$k_2]....[$k_B] = $prb_incr;
            if (defined $ct_null_clms) { $ct_null_clms += $incr_ct_null_clms; }

        } elsif ($flag <= 0) { # Hereafter, actually, we MUST prevent it from exploring theoretically
inaccessible regions. (An IMPORTANT homework)

            $prb_incr = - $prb_laln0;  # Minus the probability of the initial local alignment. #
            $ct_null_clms = 0;
            foreach my $clm (@{$set_columns}) {  # Inner foreach-loop.
                my $cnct_clm = join (':', @{$clm});
```

```
                my $cw_prb = $clm2prob{$cnct_clm};
                if (defined $cw_prb) {
                    $prb_incr += $cw_prb;
                    if ($cnct_clm eq $cnct_null_clm) { $ct_null_clms++; }
                  } else {
                    $prb_incr = undef;
                    $ct_null_clms = undef;
                    last;
                 }
            }       # End of the inner foreach-loop. #

            $incr_probs[$k_1]→[$k_2]....[$k_B] = $prb_incr;

        } else {
            $incr_probs[$k_1]→[$k_2]....[$k_B] = undef;
            $ct_null_clms = undef;
        }

        $wrk_triple→[3] = $ct_null_clms; … ADDED on Nov 13, 2018.

            # Raise $bl all the way to $B -1.
            # Update @bl_shifts and @{$bl_coords}. … Superfluous!!

        my $ct_cps = 0;
        while ($bl < $B-1) { # 1st inner while-loop. #

            $bl++;
            # $bl_shifts[$bl] = +1; # Superfluous!
            # $bl_coords→[$bl] = $Wm; # Superfluous!
            # $bl_coords→[$bl] = $org_bl_coords[$bl]; # Superfluous! (MODIFIED on Dec 24, 2018.)
            $ct_cps++;

        }   # End of the 1st inner while-loop. #

            # Stuff the stack with copies of the newly computed triple.
        while ($ct_cps>0) { # 2nd inner while-loop. #
            $ct_cps--;
            my @copy = copy (@{$wrk_triple});
            push @stack, \@copy;
        }       # End of the 2nd inner while-loop. #

}       # End of the outer while-loop. #
```

SM-3. Actually implementing algorithm to quickly compute the effects of simultaneous shifts of *interfering* (i.e., *non-isolated*) gap-blocks on the residue component of MSA probability

Let us now extend the **non-recurrent algorithm** in SM-2 to more general cases, where some of the gap-blocks interfere each other (or one another) (as considered in SM-1).

Actually, the basic framework constructed in SM-2 (i.e., the non-recurrent while-loop performing the main subroutine, "**shift_bl_and_compt_prob_incr** (@@@$$) {...}", orderly at every point in the specified neighbor) is applicable also to the more general cases.

All we have to change is the instructions to modify @curr_bds_blocks, i.e.:

```
my $relv_bds_block = $curr_bds_blocks→[$bl];
...
my $curr_lbd = ($relv_bds_block→[0] += $sh);
my $curr_rbd = ($relv_bds_block→[1] += $sh);
```

in "**shift_bl_and_compt_prob_incr (@@@$$)** {...}".

The basic rules are already discussed in SM-1.
To summarize:
1. **When gap-blocks do NOT vertically overlap each other**,
we *simply* **shift the blocks by one site each time**, even if the blocks are vertically complementary.
2. **When a gap-block vertically includes another gap-block,**
the latter will straddle the former (while the former will remain compact) when they horizontally overlap.

3. **When two gap-blocks are vertically identical,**
the "lower-rank" block will straddle the "higher-rank" one (while the latter will remain compact) when they horizontally overlap.

4. **When two gap-blocks vertically overlap but neither includes the other,**
and after they become immediately adjacent to each other (via a shift of one),
the two blocks must be swapped before a further shift (in the same direction)**.**

NOTE1: How the **block boundary changes** will accompany the "***shift***" of the block, according to the surrounding blocks, are exemplified in **Figures SSSSA8 & SSSSA9**.
Especially, when two gap-blocks get **immediately adjacent** to each other, they will be **horizontally as compact as possible**, that is, neither of them will straddle the other (and the past straddling must be undone), even if they are vertically nested or equivalent. **… ADDED on Nov 6, 2018.**

NOTE2: When performing the block shifts, it should be better also to count null columns and equivalent alignments that will arise in the coordinate space, and maybe also to monitor topological changes.

{ADDED on Nov 7, 2018:
According to **the above rules 1-4**, as well as the examples in **Figures SSSSA8 & SSSSA9**, the positions (i.e., boundaries) of the **"subject block"**, i.e, the gap-block that is being **"shift"**ed, and of the blocks surrounding it, will move as follows:

(i) Before the **"shift"**, keep extending its "front-end" while it is immediately adjacent to another block that is either vertically including or vertically equivalent to and higher-ranked than the subject.
(ii) Again, before the **"shift"**, if the subject block is immediately adjacent to another block that is vertically overlapping but non-nested with it, swap the block with the subject.
(iii) Repeat (i) and (ii) until the subject is no longer immediately adjacent to any such blocks.

(iv) **Perform the "shift"**, by moving the "sub-column", which is one site ahead of the subject and covers the same sequences as the subject, to the "rear-end" of the subject-block before the shift. If the subject is immediately adjacent to a block vertically included in it (before the "shift"), be sure to move the position of the latter block's "rear-end" to where the "read-end" of the subject was before the shift.

(v) After the **"shift"**, if the "rear-end" of the subject reaches the "rear-end" of another block that is either vertically including or vertically equivalent to and higher-ranked than the subject, move the position of the subject's "rear-end" to immediately ahead of the "front-end" of the latter. (Repeat the process until we no longer encounter any such situation.)

(vi) Again, after the **"shift"**, if the "front-end" of the subject reaches the "front-end" of another block that is either vertically included in or vertically equivalent to and lower-ranked than the subject, move the position of the latter's "front-end" to immediately behind the subject's "rear-end".

(While (i), (ii), (iv), (v) and (vi) are performed, the blocks will be **re-ordered** accordingly.)

} … END of "ADDED on Nov 7, 2018".

Here, we will consider **an algorithm that can realize all these rules**.

First, it should be convenient to prepare an array of hashes:

@inter_block_relations, with

#%{$inter_block_relations[$bl]} = ($relation => \@set_of_relevant_block_sets, …)
#for the $bl th block, with
#$relation specifies the category of their vertical relationship,
#and each element of @set_of_relevant_block_sets is the reference to an array, which contains
#blocks that *collectively* have the $relation relationship with the $bl th block. … WITHHELD on Nov 4, 2018.

$inter_block_relations[$bl1]→[$bl2] = $relation,
for the relation between the $bl1 th and $bl2 th blocks.
($inter_block_relations[$bl]→[$bl] may be undef.) … REVISED on Nov 4, 2018.

Here, $relation can be:

'NIF' (for 'non-interfering'),
'S' # (for 'sibling/parent/child' of the $bl th block … OBSOLETE as of Jan 16, 2019)
(for "effective sibling" of the $bl th block) … MODIFIED on Jan 16, 2019,
'P' (for 'parent' of the $bl th block),
'Ch' (for 'child' of the $bl th block),
'Cp' (for 'complementary'),
'ONN' (for 'overlapping yet non-nesting' NOR 'complementary-sibling/parent/child'),
'ONCS' (for 'overlapping yet non-nesting' but 'complementary-sibling/parent/child' of the $bl th block),
'ONCP' (for 'overlapping yet non-nesting' but 'complementary-parent' of the $bl th block),
'ONCC' (for 'overlapping yet non-nesting' but 'complementary-child' of the $bl th block),
'>' (for the $bl2 th block being 'vertically included' in the $bl1 th block),
'>(ch)' (for the $bl2 th block being 'vertically included' in, and an "effective child" of, the $bl1 th block), # ADDED on Jan 16, 2019.
'<' (for the $bl2 th block 'vertically including' the $bl1 th block),
'<(pa)' (for the $bl2 th block 'vertically including', and being an "effective parent" of, the $bl1 th block), # ADDED on Jan 16, 2019.
'=' (for 'vertically identical').

… See **Appendix F** for how to construct this 2D-array. (Added on Dec 11, 2018.)

NOTE1: Actually, the 'sibling' and 'complementary-sibling' here include the 'parent-child' and 'complementary-parent-child' relationships, respectively.

NOTE2: Although the categories need not be so finely classified for the purpose of this algorithm *alone*, they will be defined this way so that they will be adequately useful for other purposes, such as monitoring topological changes.

Second, we also prepare:

@collectively_complementary_blocks, with
{element} = [a set of 3 **2** or more blocks that are collectively complementary to one another,
where vertically equivalent blocks are bundled together (into an anonymous array)],

#each of whose elements is a set of 3 or more blocks that are collectively complementary to one another,
as well as:

@block2coll_comple, where
@{$block2coll_comple[$bl]} lists (the indices in @collectively_complementary_blocks of) the sets of collectively complementary blocks that the $bl th block belongs to; it is empty if there is no such set.

Third, we extend the triple quadruple: **\@triple = [\@set_columns, \@bds_blocks, \@bl_coords, $ct_null_clms],**
into **an septet octet (i.e., an seven eight-tuple): (… REVISED on Nov 13, 2018.)**

 # REVISED on Jan 13, 2019. #

\@octet = [@triple, \@blocks_w_spec_lb, \@blocks_w_spec_rb].
(NOTE on 2019/01/22: Now, the "@octet" is actually a sextet (i.e., a six-tuple).)

Here,

@{$blocks_w_spec_lb[$lb]} lists the blocks whose left-bound is $lb; and
@{$blocks_w_spec_rb[$rb]} lists the blocks whose right-bound is $rb.

[OBSOLETE as of Jan 13, 2019 (1):
 # \@sepoctet = [@triple, \@lb_sorted_set, \@orders_lb, \@rb_sorted_set, \@orders_rb].
#
Here,
#
 # $lb/rb_sorted_set[$k] is the index (e.g., in @bl_coords) (or "rank") of the $k th block in ascending order of their left-bounds/right-bounds; and
 # $orders_lb/rb[$bl] is the order (from 0 to $B -1) of the $bl th block in @lb/rb_sorted_set.
] END of "OBSOLETE as of Jan 13, 2019 (1)"

The actual generalization is ***implemented*** as follows:

```
my $relv_bds_block = $curr_bds_blocks→[$bl];
my ($prev_lbd, $prev_rbd) = @{$relv_bds_block};   # Keep the relevant block boundaries before the move. … It may be updated in the processes (i) or (ii).

my ($prev_lbd0, $prev_rbd0) = ($prev_lbd, $prev_rbd); # Keep the boundaries before any moves, which will be necessary when checking topological changes. (ADDED on Jan 24, 2019.)

...
```

{ADDED from Nov 4, 2018:

```
my @blk_rels_to_rlv = @{$inter_block_relations→[$bl]};
#       my $order_lb_rlv = $orders_lb[$bl];
#       my $order_rb_rlv = $orders_rb[$bl]; # OBSOLETE as of Jan 13, 2019.

# {The COPY of "ADDED on Nov 7, 2018" (see above):
# According to the above rules 1-4, as well as the examples in Figures SSSSA8 & SSSSA9,
# the positions (i.e., boundaries) of the "subject block",
```

```
        # i.e, the gap-block that is being  "shift"ed, and of the blocks surrounding it,
        # will move as follows:

  if ($sh > 0) { # The "shift" is to the right.

                # REVISED on Jan 13, 2019. #

  while ($prev_rb < $right_end_laln) { # Outer-outer while-loop (until $bl has NO immediately
adjacent block on its right).

    my $right_neighbors = $blocks_w_spec_lb[$prev_rb+1];
    unless (@{$right_neighbors}>0) { last; }

    my $if_relv = 0; # ADDED on Jan 14, 2019.
    foreach my $bl2 (@{$right_neighbors}) { # Outer foreach-loop (over blocks ($bl2) that are
immediate right-neighbors of $bl).

#     my $order_lb2 = $order_lb_rlv+1;
#     for (; $order_lb2 < $B; $order_lb2++) { # Examine the left-bounds of the blocks on the
right. #
#       my $bl2 = $lb_sorted_set[$order_lb2];  # OBSOLETE as of Jan 13, 2019 (2)

        my $bds_bl2 = $bds_blocks[$bl2];
        my ($lbd2, $rbd2) = @{$bds_bl2};

        # (iii) Repeat (i) and (ii) until the subject is
        # no longer immediately adjacent to any such blocks.
#       if ($lb2 > $prev_rbd+1) { last; }
#       if ($lb2 < $prev_rbd+1) { next; } # Skip if the block already overlaps the subject
horizontally.  # OBSOLETE as of Jan 13, 2019 (2)


        my $rel_to_rlv = $blk_rels_to_rlv[$bl2];

        if ( ($rel_to_rlv eq '<') or ($rel_to_rlv eq '<(pa)') or (($rel_to_rlv eq '=') and ($bl > $bl2)) )
{ # Added '<(pa)' on Jan 17, 2019.

        # (i) Before the "shift", keep extending its "front-end"
        # while it is immediately adjacent to another block
        # that is either vertically including or
        # vertically equivalent to and higher-ranked than the subject.

        => {Remove $bl from @{$blocks_w_spec_rb[$prev_rbd]}.}; # ADDED on Jan 13, 2019. #

          $relv_bds_block→[1] = $prev_rbd = $rbd2; # Extend the right-boundary.

        => {Add $bl to @{$blocks_w_spec_rb[$prev_rbd]}.}; # ADDED on Jan 13, 2019. #

        $if_relv = 1; # ADDED on Jan 14, 2019.

        $last; # Leave the foreach-loop. # ADDED on Jan 14, 2019.

                # OBSOLETE as of Jan 13, 2019 (5). #
                # Re-order the right-boundaries of the blocks. #
                # (More precisely, move the current block "to the right" of the immediately adjacent
one.)
#
#           my $order_rb2 = $orders_rb[$bl2];
```

```
#
#           for (my $k=$order_rb_rlv; $k < $order_rb2; $k++) {
#                 my $bl3 = $rb_sorted_set[$k+1];
#                 $orders_rb[$bl3]--;
#                 $rb_sorted_set[$k] = $bl3;
#           }
#           $rb_sorted_set[$order_rb2] = $bl;
#           $orders_rb[$bl] = $order_rb2;
                # END of "OBSOLETE as of Jan 13, 2019 (5)." #

                # This is just a coordinate change (of the right-boundary) and NOT a topological
change. #

        } elsif (($rel_to_rlv eq 'ONN') or ($rel_to_rlv eq 'ONCS')) {

        # (ii) Again, before the "shift",
        # if the subject block is immediately adjacent to another block
        # that is vertically overlapping but non-nested with it,
        # swap the block with the subject.

          my $size_bl1 = $prev_rbd - $prev_lbd + 1;
          my $size_bl2 = $rbd2 - $lbd2 + 1;

                # First, swap the columns. #

          my @reservoir_clms = @set_columns[$prev_lbd .. $prev_rbd];
          for (my $c = $lbd2; $c <= $rbd2; $c++) { $set_columns[$c-$size_bl1] =
$set_columns[$c]; }
          for (my $c=0; $c<$size_bl1; $c++) { $set_columns[$rbd2-$c] =
$reservoir_clms[$size_bl1 -1- $c ]; }

                # Second, swap the left boundaries. #
                # & Third, swap the right boundaries. #

        if ($ub_bl < $prev_rbd - $prev_lbd + $rbd2 - $lbd2 + 2 ) { # ADDED on Jan 17, 2019.

                # ADDED on Jan 17, 2019.
          for (my $bl3 = 0; $bl3 < $ub_bl; $bl3++) {
                my $bds_bl3 = $bds_blocks[$bl3];
                foreach my $indx (0, 1) {
                    my $bd = $bds_bl3→[$indx];
                    if (($prev_lbd <= $bd) and ($bd <= $prev_rbd)) {
                        $bds_bl3→[$indx] += $size_bl2;
                    } elsif (($lbd2 <= $bd) and ($bd <= $rbd2)) {
                        $bds_bl3→[$indx] -= $size_bl1;
                    }
                } # End of the foreach-loop (over $indx).
          }   # End of the for-loop (over $bl3).

        } else { # ADDED on Jan 17, 2019.

                # ADDED on Jan 14, 2019. (1) #

          for (my $c = $prev_lbd; $c <= $prev_rbd; $c++) { # Process the boundaries in the 1st set.
#
                my $blocks_w_lb_on_c = $blocks_w_spec_lb[$c];
                foreach my $bl3 (@{$blocks_w_lb_on_c}) { $bds_blocks[$bl3]→[0] +=
$size_bl2; }
```

```
                my $blocks_w_rb_on_c = $blocks_w_spec_rb[$c];
                foreach my $bl3 (@{$blocks_w_rb_on_c}) { $bds_blocks[$bl3]→[1] +=
$size_bl2; }

            } # End of "Process the boundaries in the 1st set". #

            for (my $c = $lbd2; $c <= $rbd2; $c++) { # Process the boundaries in the 2nd set. #

                my $blocks_w_lb_on_c = $blocks_w_spec_lb[$c];
                foreach my $bl3 (@{$blocks_w_lb_on_c}) { $bds_blocks[$bl3]→[0] -= $size_bl1; }

                my $blocks_w_rb_on_c = $blocks_w_spec_rb[$c];
                foreach my $bl3 (@{$blocks_w_rb_on_c}) { $bds_blocks[$bl3]→[1] -=
$size_bl1; }

            }  # End of "Process the boundaries in the 2nd set". #

        } # ADDED on Jan 17, 2019.

            # Penultimately, swap the 1st and 2nd sets in @blocks_w_spec_lb and
@blocks_w_spec_rb. #

            my @reservoir_w_spec_lb = @blocks_w_spec_lb[$prev_lbd .. $prev_rbd];
            my @reservoir_w_spec_rb = @blocks_w_spec_rb[$prev_lbd .. $prev_rbd];

            for (my $c = $lbd2; $c <= $rbd2; $c++) {
                $blocks_w_spec_lb[$c-$size_bl1] = $blocks_w_spec_lb[$c];
                $blocks_w_spec_rb[$c-$size_bl1] = $blocks_w_spec_rb[$c];
            }
            for (my $c=0; $c<$size_bl1; $c++) {
                $blocks_w_spec_lb[$rbd2-$c] = $reservoir_w_spec_lb[$size_bl1 -1- $c ];
                $blocks_w_spec_rb[$rbd2-$c] = $reservoir_w_spec_rb[$size_bl1 -1- $c ];
            }

            # END of "ADDED on Jan 14, 2019. (1)" #

            # OBSOLETE as of Jan 13, 2019 (6). #
#           my ($lb1_order_lb, $rb1_order_lb) = ($order_lb_rlv, $order_lb2-1);
#           my ($lb2_order_lb, $rb2_order_lb) = ($order_lb2, $order_lb2);
#
#           while ($lb1_order_lb > 0) { # Examine the current $lb1_order_lb. #
#               my $bl3 = $lb_sorted_set[$lb1_order_lb-1];
#               if ($bds_blocks[$bl3]→[0] < $prev_lbd) { last; }
#               $lb1_order_lb--;
#           }
#           while ($rb1_order_lb > $lb1_order_lb) { # Examine the current $rb1_order_lb. #
#               my $bl3 = $lb_sorted_set[$lb1_order_lb];
#               if ($bds_blocks[$bl3]→[0] < $lbd2) { last; }
#               $rb1_order_lb--;
#               $lb2_order_lb--;
#           }
#           while ($rb2_order_lb < $B-1) { # Examine the current $rb2_order_lb. #
#               my $bl3 = $lb_sorted_set[$rb2_order_lb+1];
#               if ($bds_blocks[$bl3]→[0] > $rbd2) { last; }
#               $rb2_order_lb++;
#           }
```

```
#           my $ct_in_bl1 = $rb1_order_lb - $lb1_order_lb +1;
#           my $ct_in_bl2 = $rb2_order_lb - $lb2_order_lb +1;
#
#           my @reservoir_lb_std = @lb_sorted_set[$lb1_order_lb .. $rb1_order_lb];
#
#           for (my $k=$lb2_order_lb; $k <= $rb2_order_lb; $k++) { # Process the left-boundaries in
the 2nd set.
#               my $bl3 = $lb_sorted_set[$k];
#               $lb_sorted_set[$k-$ct_in_bl1] = $bl3;
#               $orders_lb[$bl3] -= $ct_in_bl1;
#               $bds_blocks[$bl3]→[0] -= $size_bl1;
#           }
#
#           for (my $k=$lb1_order_lb; $k <= $rb1_order_lb; $k++) { # Process the left-boundaries in
the 1st set.
#               my $bl3 = $reservoir_lb_std[$k-$lb1_order_lb];
#               $lb_sorted_set[$k+$ct_in_bl2] = $bl3;
#               $orders_lb[$bl3] += $ct_in_bl2;
#               $bds_blocks[$bl3]→[0] += $size_bl2;
#           }
                # END of "OBSOLETE as of Jan 13, 2019 (6)." #

                # OBSOLETE as of Jan 13, 2019 (7). #

                # Third, swap the right boundaries. #

#            my $order_rb2 = $orders_rb[$bl2];
#
#           my ($lb1_order_rb, $rb1_order_rb) = ($order_rb_rlv, $order_rb_rlv);
#           my ($lb2_order_rb, $rb2_order_rb) = ($order_rb_rlv+1, $order_rb2);
#
#           while ($lb1_order_rb > 0) { # Examine the current $lb1_order_rb. #
#               my $bl3 = $rb_sorted_set[$lb1_order_rb-1];
#               if ($bds_blocks[$bl3]→[1] < $prev_lbd) { last; }
#               $lb1_order_rb--;
#           }
#           while ($lb2_order_rb < $rb2_order_rb) { # Examine the current $lb2_order_rb. #
#               my $bl3 = $rb_sorted_set[$lb2_order_rb];
#               if ($bds_blocks[$bl3]→[1] > $prev_rbd) { last; }
#               $rb1_order_rb++;
#               $lb2_order_rb++;
#           }
#           while ($rb2_order_rb < $B-1) { # Examine the current $rb2_order_rb. #
#               my $bl3 = $rb_sorted_set[$rb2_order_rb+1];
#               if ($bds_blocks[$bl3]→[1] > $rbd2) { last; }
#               $rb2_order_rb++;
#           }
#
#           $ct_in_bl1 = $rb1_order_rb - $lb1_order_rb +1;
#           $ct_in_bl2 = $rb2_order_rb - $lb2_order_rb +1;
#
#           my @reservoir_rb_std = @rb_sorted_set[$lb1_order_rb .. $rb1_order_rb];
#
#           for (my $k=$lb2_order_rb; $k <= $rb2_order_rb; $k++) { # Process the right-boundaries
in the 2nd set.
#               my $bl3 = $rb_sorted_set[$k];
#               $rb_sorted_set[$k-$ct_in_bl1] = $bl3;
#               $orders_rb[$bl3] -= $ct_in_bl1;
```

```
#                   $bds_blocks[$bl3]→[1] -= $size_bl1;
#               }
#
#               for (my $k=$lb1_order_rb; $k <= $rb1_order_rb; $k++) { # Process the right-boundaries in the 1st set.
#                   my $bl3 = $reservoir_rb_std[$k-$lb1_order_rb];
#                   $lb_sorted_set[$k+$ct_in_bl2] = $bl3;
#                   $orders_rb[$bl3] += $ct_in_bl2;
#                   $bds_blocks[$bl3]→[1] += $size_bl2;
#               }
                    # END of "OBSOLETE as of Jan 13, 2019 (7)." #


                    # Finally, update some important variables. #

               ($prev_lbd, $prev_rbd) = @{$relv_bds_block};
#              $order_lb_rlv = $orders_lb[$bl];
#              $order_rb_rlv = $orders_rb[$bl]; # OBSOLETE as of Jan 13, 2019 (8).


                    # This is just a swapping of blocks and NOT a topological change. #

                $if_relv = 1; # ADDED on Jan 14, 2019.

                last; # Leave the foreach-loop. # ADDED on Jan 14, 2019.

            } else {

                next; # Otherwise, do nothing.
                    # Later, we may include something here to mark the imminent topological change.
            }

#       } # END of the for-loop (over $order_lb2) to Examine the left-bounds of the blocks on the right. # OBSOLETE as of Jan 13, 2019.

        } # End of the Outer foreach-loop (over blocks ($bl2) that are  immediate right-neighbors of $bl).
            unless ($if_relv) { last; } # ADDED on Jan 14, 2019.

    } # END of the Outer-outer while-loop (until $bl has NO immediately adjacent block on its right).


            if ($rbd_lcl_align <= $prev_rbd) { # This is crutial for preventing meaningless computations. #

                    return  {Something to indicate that the $bl th block will go beyond the right-boundary of the local alignment under consideration};
                        # (=> STOP "shift"ing the $bl th block further to the right (and start "shift"ing it to the left (from the "origin").)

            }

            # Here, either ($order_lb2 = $B) or ($bl2 = $lb_sorted_set[$order_lb2];
            # $bds_blocks[$bl2]→[0] > $prev_rbd + 1) MUST hold!!


            # (iv) Perform the "shift", by moving the "sub-column",
```

```
        # which is one site ahead of the subject and covers the same sequences as the subject,
        # to the "rear-end" of the subject-block before the shift.

        # If the subject is immediately adjacent to a block vertically included in it (before the
"shift"),
        # be sure to move the position of the latter block's "rear-end" to
        # where the "read-end" of the subject was before the shift.

                # MODIFIED on Jan 14, 2019. (1) #

    my $right_neighbors = $blocks_w_spec_lb[$prev_rbd+1];
    if (@{$right_neighbors}>0) {

        foreach my $bl3 (@{$right_neighbors}) { # Foreach-loop (over $bl3 that are the
immediate right-neighbors of $bl). #

        # OBSOLETE as of Jan 14, 2019. (1) #
#       for (my $order_lb3 = $order_lb2-1; $order_lb3 > $order_lb_rlv; $order_lb3--) {
#
#           my $bl3 = $lb_sorted_set[$order_lb3];
#           my $lbd3 = $bds_blocks[$bl3]→[0];
#           if ($lbd3 <= $prev_rbd) { last; }
#           if ($lbd3 > $prev_rbd+1) { next; }
        # END of "OBSOLETE as of Jan 14, 2019. (1)" #


           my $rel_to_rlv = $blk_rels_to_rlv[$bl3];

           if (($rel_to_rlv eq '>') or ($rel_to_rlv eq '>(ch)') or (($rel_to_rlv eq '=') and ($bl3 >
$bl)) ) { # Added '>(ch)' on Jan 17, 2019.

                my $lbd3 = $bds_blocks[$bl3]→[0]; # ADDED on Jan 14, 2019.

                => {Remove $bl3 from @{$blocks_w_spec_lb[$lbd3]}. }; # ADDED on Jan 14,
2019.

                $bds_blocks[$bl3]→[0] = $prev_lbd;  # Update the left-boundary of the immediately
adjacent block. #

                => {Add $bl3 to @{$blocks_w_spec_lb[$prev_lbd]}. }; # ADDED on Jan 14,
2019.

                $last; # Leave the foreach-loop. # ADDED on Jan 14, 2019.

                        # OBSOLETE as of Jan 14, 2019. (2) #
                        # RE-order the left boundaries. #
#               for (my $k = $order_lb_rlv; $k < $order_lb3; $k++) {
#                   my $bl5 = $lb_sorted_set[$k];
#                   $lb_sorted_set[$k+1] = $bl5;
#                   $orders_lb[$bl5]++;
#               }
#               $orders_lb[$bl3] = $order_lb_rlv;
#               $lb_sorted_set[$order_lb_rlv] = $bl3;
#               $order_lb_rlv = $orders_lb[$bl];
                        # END of "OBSOLETE as of Jan 14, 2019. (2)" #

           }
```

```
#         }   # END of the for-loop (over $order_lb3) to examine blocks immediately adjacent to the right-boundary of the subject. # Paired with "OBSOLETE as of Jan 14, 2019.  (1)"

        } # END of the foreach-loop (over $bl3 that are the immediate right-neighbors of $bl).
#
    } # END of the "if (@{$right_neighbors}>0) {...}"
                # END of "MODIFIED on Jan 14, 2019. (1)" #


        # my $curr_lbd = ($relv_bds_block→[0] += $sh);
        # my $curr_rbd = ($relv_bds_block→[1] += $sh);
        # … The simplest implementation of the "shift" of the positions (in SM-2). … CAN be used as it is!!
        # But, $sh = +1 here, so modify then slightly.


        {Remove $bl from @{$blocks_w_spec_lb[$prev_lbd]}. }; # ADDED on Jan 14, 2019.
        {Remove $bl from @{$blocks_w_spec_rb[$prev_rbd]}. }; # ADDED on Jan 14, 2019.

        my $curr_lbd = ({$relv_bds_block→[0]}++);
        my $curr_rbd = ({$relv_bds_block→[1]} ++);

        => {Add $bl to @{$blocks_w_spec_lb[$curr_lbd]}. }; # ADDED on Jan 14, 2019.
        => {Add $bl to @{$blocks_w_spec_rb[$curr_rbd]}. }; # ADDED on Jan 14, 2019.

                # OBSOLETE as of Jan 14, 2019. (3) #

        #  … And, BE SURE to RE-ORDER the left boundaries and the right boundaries!!
#
#       for (my $order_lb3 = $order_lb_rlv+1; $order_lb3 < $B; $order_lb3++) { # Examine the left-boundaries. #
#           my $bl3 = $lb_sorted_set[$order_lb3];
#           if ($curr_lbd <= $bds_blocks[$bl3]→[0]) { last; }
#
#               # Swap the orders of $bl and $bl3. #
#           $orders_lb[$bl3] = $order_lb_rlv;
#           $lb_sorted_set[$order_lb_rlv] = $bl3;
#           $order_lb_rlv = $orders_lb[$bl] = $order_lb3;
#           $lb_sorted_set[$order_lb3] = $bl;
#       }
#
#       for (my $order_rb3 = $order_rb_rlv+1; $order_rv3 < $B; $order_rb3++) { # Examine the right-boundaries. #
#           my $bl3 = $rb_sorted_set[$order_rb3];
#           if ($curr_rbd <= $bds_blocks[$bl3]→[1]) { last; }
#
#               # Swap the orders of $bl and $bl3. #
#           $orders_rb[$bl3] = $order_rb_rlv;
#           $rb_sorted_set[$order_rb_rlv] = $bl3;
#           $order_rb_rlv = $orders_rb[$bl] = $order_rb3;
#           $rb_sorted_set[$order_rb3] = $bl;
#       }
#
                # END of "OBSOLETE as of Jan 14, 2019. (3)" #


                ### Here, do the main computations for the subroutine. ###
```

```
        # (v) After the "shift",
        # if the "rear-end" of the subject reaches the "rear-end" of another block
        # that is either vertically including or vertically equivalent to and higher-ranked than the subject,
        # move the position of the subject's "rear-end" to immediately ahead of the "front-end" of the latter.
        # (Repeat the process until we no longer encounter any such situation.)


                # MODIFIED on Jan 14, 2019. (2) #

    my $blocks_w_curr_lbd = $blocks_w_spec_lb[$curr_lbd];
    whille (@{$blocks_w_curr_lbd}>0 1) { # Modified the condition on Jan 17, 2019.

        my $if_relv = 0; # ADDED on Jan 14, 2019.
        foreach my $bl3 (@{$blocks_w_curr_lbd}) { # Foreach over $bl3 that share the left-boundary with $bl. #
            if ($bl3 == $bl) { next; } # ADDED on Jan 17, 2019.
            my $rbd3 = $bds_blocks[$bl3]→[1];

                # OBSOLETE as of Jan 14, 2019. (5) #
#       my $order_lb3 = $order_lb_rlv+1;
#       for (; $order_lb3 < $B; $order_lb3++) {  # Here, "rear-end" = "left boundary".
#           my $bl3 = $lb_sorted_set[$order_lb3];
#           my ($lbd3, $rbd3) = @{$bds_blocks[$bl3]};
#           if ($curr_lbd < $lbd3) { last; }
#           if ($lbd3 < $curr_lbd) { next; }
                # END of "OBSOLETE as of Jan 14, 2019. (5)" #

            my $rel_to_rlv = $blk_rels_to_rlv[$bl3];
            if ( ($rel_to_rlv eq '<')  or ($rel_to_rlv eq '<(pa)') or (($rel_to_rlv eq '=') and ($bl3 < $bl)) ) { # Added '<(pa)' on Jan 17, 2019.

                {Remove $bl from @{$blocks_w_spec_lb[$curr_lbd]}. }; # ADDED on Jan 14, 2019.

                $curr_lbd = $bds_blocks[$bl]→[0] = $rbd3+1;

                => {Add $bl to @{$blocks_w_spec_lb[$curr_lbd]}. }; # ADDED on Jan 14, 2019.

                $if_relv = 1;# ADDED on Jan 14, 2019.

                last; # Leave the foreach-loop.  # ADDED on Jan 14, 2019.
            }
#       }  # End of for over $bl3. # Paired with "OBSOLETE as of Jan 14, 2019. (5)"

        }  # End of foreach over $bl3 that share the left-boundary with $bl. #

                # Update @{$blocks_w_curr_lbd}. #
        $blocks_w_curr_lbd  = ($if_relv and ($curr_lbd +$size_bl -1 < $curr_rbd)) ?
                              $blocks_w_spec_lb[$curr_lbd] : []; # ADDED on Jan 14, 2019.

    } # End of "while (@{$blocks_w_curr_lbd}>0 1) {...}"

                # END of "MODIFIED on Jan 14, 2019. (2)" #

                # OBSOLETE as of Jan 14, 2019. (6) #
```

```
                # Re-order the left boundaries accordingly. #
#
#       for ($order_lb3--; $order_lb3 > $order_lb_rlv; $order_lb3--) { # Move the $order_lb3 back to the rightmost block among those whose left-ends are on the left of the current left-end of the subject.
#           my $bl3 = $lb_sorted_set[$order_lb3];
#           my ($lbd3, $rbd3) = @{$bds_blocks[$bl3]};
#           if ($bds_blocks[$bl3]→[0] < $curr_lbd) { last; }
#       }
#
#       if ($order_lb_rlv < $order_lb3) {
#           for (my $k = $order_lb_rlv; $k < $order_lb3; $k++) {
#               my $bl5 = $lb_sorted_set[$k+1];
#               $lb_sorted_set[$k] = $bl5;
#               $orders_lb[$bl5]--;
#           }
#           $lb_sorted_set[$order_lb3] = $bl;
#           $orders_lb[$bl] = $order_lb3;
#       }
#
                # END of "OBSOLETE as of Jan 14, 2019. (6)" #


        # (vi) Again, after the "shift",
        # if the "front-end" of the subject reaches the "front-end" of another block
        # that is either vertically included in or vertically equivalent to and lower-ranked than the subject,
        # move the position of the latter's "front-end" to immediately behind the subject's "rear-end".


                # MODIFIED on Jan 14, 2019. (3) #

    my $blocks_w_curr_rbd = $blocks_w_spec_rb[$curr_rbd];
    if (@{$blocks_w_curr_rbd}> 1) { # Modified the condition on Jan 17, 2019.
#    if (@{$blocks_w_curr_rbd}>0) {

        foreach my $bl3 (@{$blocks_w_curr_rbd}) { # Outer foreach-loop (over $bl3 whose right-bounds are $curr_rbd). #
            if ($bl3 == $bl) { next; }

                # OBSOLETE as of Jan 14, 2019. (7) #
#
#       my $order_rb3 = $order_rb_rlv+1;
#       for (   ; $order_rb3 < $B; $order_rb3++) {  # Here, "front-end" = "right boundary".
#
#           my $bl3 = $rb_sorted_set[$order_rb3];
#           my ($lbd3, $rbd3) = @{$bds_blocks[$bl3]};
#           if ($curr_rbd < $rbd3) { last; }
#
                # END of "OBSOLETE as of Jan 14, 2019. (7)" #


            my $rel_to_rlv = $blk_rels_to_rlv[$bl3];
            unless ( ($rel_to_rlv eq '>') or ($rel_to_rlv eq '>(ch)') or (($rel_to_rlv eq '=') and ($bl < $bl3)) ) { next; } # Added '>(ch)' on Jan 17, 2019.

            my ($lbd3, $rbd3) = @{$bds_blocks[$bl3]}; # ADDED on Jan 14, 2019.
```

```
            => {Remove $bl3 from @{$blocks_w_spec_rb[$rbd3]}. };  # ADDED on Jan 14, 2019.

            $rbd3 = $bds_blocks[$bl3]→[1] = $curr_lbd -1;

            => {Add $bl3 to @{$blocks_w_spec_rb[$rbd3]}. };  # ADDED on Jan 14, 2019.


                # ADDED on Jan 14, 2019. (2) #

                # Further examine whether some blocks share the right-bounds with $bl3, and,
                #  if so, further retract the right-bound, $rbd3, of $bl3.

            my $rels_w_bl3 = $inter_block_relations[$bl3];

            while ($lbd3 + $size3 -1 < $rbd3) {

                my $blocks_w_rbd3 = $blocks_w_spec_rb[$rbd3];
                if (@{$blocks_w_rbd3} < 2) { last; } # Modified the condition on Jan 17, 2019.
#               if (@{$blocks_w_rbd3} == 0) { last; }

                my $if_relv = 0;
                foreach my $bl5 (@{$blocks_w_rbd3}) { # foreach-loop (over $bl5 with $rbd3). #

                    if ($bl5 == $bl3) { next; }

                    my $rel = $rels_w_bl3→[$bl5];
                    unless (($rel eq '<') or ($rel eq '<(pa)') or (($rel eq '=') and ($bl5 < $bl3)) )
{ next; } # Added '<(pa)' on Jan 17, 2019.

                    my $lbd5 = $bds_blocks[$bl5]→[0];

                    => {Remove $bl3 from @{$blocks_w_spec_rb[$rbd3]};

                   $rbd3 = $bds_blocks[$bl3]→[1] = $lbd5 -1;

                   => {Add $bl3 to @{$blocks_w_spec_rb[$rbd3]}. };

                    $if_relv = 1;
                    last; # Leave the foreach-loop.

                } # End of foreach-loop (over $bl5 with $rbd3). #

                unless ($if_relv) { last; }

            } # End of "while ($lbd3 + $size3 -1 < $rbd3) {...}"

                # END of "ADDED on Jan 14, 2019. (2)" #

                # OBSOLETE as of Jan 14, 2019. (8) #

                # Re-order the right-boundaries accordingly. #
#
#               my $order_rb5 = $order_rb3-1;
#               for (; $order_rb5>= 0; $order_rb5--) {
#                   my $bl5 = $rb_sorted_set[$order_rb5];
#                   if ($bds_blocks[$bl5]→[1] <= $rbd3) {
#                       last;
```

```
#                 }
#             }
#             $order_rb5++;
#             if ($order_rb5 < $order_rb3) {
#                 for (my $k= $order_rb5; $k < $order_rb3; $k++) {
#                     my $bl6 = $rb_sorted_set[$k];
#                     $rb_sorted_set[$k+1] = $bl6;
#                      $orders_rb[$bl6]++;
#                 }
#                 $rb_sorted_set[$order_rb5] = $bl3;
#                 $orders_rb[$bl3] = $order_rb5;
#             }
#
                    # END of "OBSOLETE as of Jan 14, 2019. (8)" #


#         } # End of "for (   ; $order_rb3 < $B; $order_rb3++) {...}" # Paired with "OBSOLETE as of Jan 14, 2019. (7)" #

        } # End of outer foreach-loop (over $bl3 whose right-bounds are $curr_rbd). #

    }  # END of "if (@{$blocks_w_curr_rbd}>0) {...}"

                    # END of "MODIFIED on Jan 14, 2019. (3)" #


        # (While (i), (ii), (iv), (v) and (vi) are performed, the blocks will be re-ordered accordingly. … (probably) DONE already!!)
        #
        # }  … END of "The COPY of "ADDED on Nov 7, 2018" (see above)".

    } else {   # The "shift" is to the left.

        ## Do almost exactly the same as when the "shift" is to the right,
        ## but in the opposite direction (i.e., from the right to the left).

    }

} … END of "ADDED from Nov 4, 2018"


        # Counting the null columns can be easily incorporated by
        # (i) initially counting the null columns in the input local alignment,
        # and then
        # (ii) counting the null columns created and annihilated
        # in "shift_bl_and_compt_prob_incr (@@@$$) {...}". … ADDED on Nov 10, 2018.
        (=> MODIFIED the above algorithms accordingly!! … on Nov 13, 2018.)


                    # RESTARTED on Jan 15, 2019. (1) #


        # Before the main processes, it would be better to examine the degeneracy of each alignment due to the swapping of vertically equivalent blocks with the same size.

        # => REPLACE this process with
```

```
        # checking the positional arrangement of such swappable blocks, i.e., whether the positional order of such blocks conforms to the order of their ranks, which will be performed AFTER the counting of null columns. (REVISED on Dec 10. 2018.)

        # For this purpose, we will construct:
        # %branch2up_down2equiv_blocks = ($branch => {$up_or_down => \@equiv_blocks, …}, ..),
        # where $up_or_down = 'U'/'L' depending on whether the gap-block is on the "upper"/"lower"-side of the branch,
        # and @equiv_blocks lists the indices of vertically equivalent blocks (in ascending order).
        # And @indices_blocks_w_equiv, which list the indices of blocks with vertically equivalent ones,
        # as well as @degeneracies0, which is $BE = #{blocks w/ vertical equivalents} dimensional,
        # and $degeneracies0[$y_1]→[$y_2]→...→[$y_{$BE-1}]
        # gives the baseline degeneracy of the alignment in which
        # the 1st, 2nd, …, {$BE-1} th blocks w/ vertical equivalents (in @indices_blocks_w_equiv)
        # have the coordinates $y_1, $y_2, …, $y_{$BE-1}, respectively.

(1) Enumerate the sets of vertically equivalent blocks. (As before, this will be performed before the main process. … ADDED on Dec 10, 2018.)

my @indices_blocks_w_equiv = (); # Initialize the list of indices. #
my @sets_blocks_w_equiv = ();

foreach my $br (sort {$a <=> $b} keys %branch2up_or_down2equiv_blocks) { # Preliminary outer foreach-loop.
    my $up_or_down2equiv_blocks = $branch2up_or_down2equiv_blocks{$br};

    foreach my $ud (sort keys %{$up_or_down2equiv_blocks}) { # Preliminary middle foreach-loop.
        my $equiv_blocks = $up_or_down2equiv_blocks{$ud};

        my %size2equiv_blocks;
        foreach my $bl (@{$equiv_blocks}) { # 1st preliminary inner foreach-loop.
            my ($lbd, $rbd) = @{$bds_blocks[$bl]};

                # MODIFIED on Jan 15, 2019. #

            my $size = $block_sizes[$bl]; # See Appendix F-sppl G.

                # OBSOLETE as of Jan 15, 2019.  (1) #
#           my $size = $rbd - $lbd + 1;
#
#               # Subtract the sizes of (vertically) 'including' or 'equivalent' blocks, if at all.
#
#               # (ADDED on Nov 29, 2018.)
#           my $relations_w_bl = $inter_block_relations[$bl];
#           for (my $bl2 = 0; $bl2 < $B; $bl2++) {
#               if ($bl2 == $bl) { next; }
#               my $rel = $relations_w_bl→[$bl2];
#               unless (($rel eq '<') or (($rel eq '=') and ($bl2 < $bl)) ) { next; } # $bl2 is NEITHER vertically including NOR vertically equivalent to (and higher-ranked than) $bl.
#               my ($lbd2, $rbd2) = @{$bds_blocks[$bl2]};
#               unless (($lbd <= $lbd2) and ($rbd2 <= $rbd)) { next; } # $bl2  is NOT included in $bl.
#               $size2 = $rbd2 - $lbd2 + 1;
#               $size -= $size2;
```

```
#                }
#                        # (END of "ADDED on Nov 29, 2018".)
#
                         # OBSOLETE as of Jan 15, 2019.  (2) #

                         # END of "MODIFIED on Jan 15, 2019." #

                my $ebs = $size2equiv_blocks{$size};
                unless (defined $ebs) { $ebs = $size2equiv_blocks{$size} = []; }
                push @{$ebs}, $bl;
        } # END of the 1st preliminary inner foreach-loop.

        foreach my $size (sort {$a <=> $b} keys %size2equiv_blocks) { # 2nd preliminary inner
foreach-loop.
            my $ebs = $size2equiv_blocks{$size};
            if (@{$ebs} < 2) { next; } # Skip if containing only one block.

            push @indices_blocks_w_equiv, @{$ebs};
            push @sets_blocks_w_equiv, $ebs;

        }  # END of the 2nd preliminary inner foreach-loop.

    } # END of the preliminary middle foreach-loop.
} # END of the  preliminary outer foreach-loop.

                # END of "RESTARTED on Jan 15, 2019. (1)" #
```

(2) Initialize @degeneracies0, so that every relevant element is 1 (unity). **… OBSOLETE as of Dec 10, 2018.**

```
        # Use a single while-loop, which is in a sense similar to, but much simpler than, the while-
loop (guided by a stack and an index specifying the block under consideration) for the main
computational process. … OBSOLETE as of Dec 10, 2018.
```

(3) Compute the degeneracies, **Check the positional orders among swappable blocks**, which should be performed after counting null columns **(REVISED on Dec 10, 2018)**:

```
        # NOTE (added on Dec 10, 2018): This has the same effect as assigning the right
degeneracies to all degenerate alignments.
        # Besides, it is much simpler than computing degeneracies, because it can avoid possible
complications due to unequal ranges of the swappable blocks, as well as complications due to
changeable ranges of some blocks.
        # Remember that what we actually need is to avoid double-counting of the same alignment,
but not to know exactly how many times each alignment occur (in a given coordinate space).

# my $rank_init_coord = 0; # This should be the rank (, i.e., the position in
@indices_blocks_w_equiv) of the 1st coordinate in @{$ebs}. … Probably, unnecessary.
(Deliberated on Nov 28, 2018)

        # MODIFIED on Nov 28, 2018. #
my $ct_sets_ebs = scalar (@sets_blocks_w_equiv);

my @sets_sub_degeneracies = (); # (the $i th element) = {the degeneracies assigned to the points of
the sub-coordinate space for the set, @{$sets_blocks_w_equiv[$i]}, resulting from the relations
among its constituent blocks.  … OBSOLETE as of Dec 10, 2018.

my $if_right_order = 1; # = 0 if the positional order of the swappable blocks disagree with the order
of their ranks. (ADDED on Dec 10, 2018.)
```

```
for (my $i = 0; $i < $ct_sets_ebs; $i++) {
        my $ebs = $sets_blocks_w_equiv[$i];
#foreach my $ebs (@sets_blocks_w_equiv) {

        my $ct_ebs = @{$ebs};

        my $bl1 = $ebs->[0];
        my $lbd_bl1 = $curr_bds_blocks→[$bl1][0];

        for (my $i=1; $i <$ct_ebs; $i++) { # Inner for-loop (over swappable blocks). #
            my $bl2 = $ebs->[$i];
            my $lbd_bl2 = $curr_bds_blocks→[$bl2][0];

            if ($lbd_bl2 <= $lbd_bl1) {
                # The positional order clashes with the order of ranks. #
                $if_right_order = 0;
                last;
            } else {
                $lbd_bl1 = $lbd_bl2;
            }
        }  # END of the inner for-loop (over swappable blocks). #

        if ($if_right_order == 0) { last; }
```

[[OBSOLETE as of Dec 10, 2018:

```
          # Using @bds_bl_coords and the initial positions of the blocks,
          # determine the coordinate ranges in which only particular blocks can be swapped with one another. (See Appendix E.)

        # => For each coordinate range, determine the degeneracy according to the groups of swappable blocks.
        # … The point must be to constrain the possibilities by considering the "isolated" blocks first, then, the blocks "isolated" as a result of the blocks "isolated" before…. (See Appendix E.)

        # => Assign the determined degeneracy to each range (via multiplication).

                # ADDED on Nov 28, 2018. #
#       my $sub_degeneracies = sub_degeneracies_in_sub_coordinate_space (@{$ebs}, @init_bds_blocks, @init_bl_coords, @bds_bl_coords, @inter_block_relations);
                # This subroutine embodies the algorithm in Appendix E.
                # $sub_degeneracies→[$x_0]...[$x_{$ct_ebs}] is the sub-degeneracy
                # assigned to the coordinates ($x_0, …, $x_{$ct_ebs-1}) of the $ebs→[0], …, $ebs→[$ct_ebs-1] th blocks, respectively.
                # (Actually, the ",…," includes vertically equivalent but non-swappable blocks as well, in order to consider ALL blocks whose ranks are between $ebs→[0] and $ebs→[$ct_ebs-1].)

#       $sets_sub_degeneracies[$i] = $sub_degeneracies;

                # END of "ADDED on Nov 28, 2018." #

        # (NOTE1: The ranges with degeneracy = 1 (unity) can be skipped.)
        # (NOTE2: the coordinates of the other blocks can be anything.)

        # Use double while-loops, the outer one for the coordinates of the blocks before @{$ebs},
        # and the inner one for the coordinates of the blocks after @{$ebs},
```

```
        # with each while-loop driven by a stack of coordinate vectors and an index specifying the coordinate to be moved.
        # (In a way, each while-loop is similar to, but much simpler than, the while-loop for the main computational process.)
#               … Probably, this double while-loops can be omitted, and an enormous memory space can be saved, by storing only the degeneracies assigned to the sub-coordinate spaces of relevant sets  of blocks.   (Deliberated on Nov 28, 2018.)



# my @bds_bl_coords = (The set of boundaries for the block coordinates);
#  @{$bds_bl_coords[$bl]} = ($lb_coord, $rb_coord)  in the full-closed convention;

#               $rank_init_coord += $ct_ebs;  … Probably unnecessary.
]]   # END of "OBSOLETE as of Dec 10, 2018"

}

if ($if_right_order == 0) {  # ADDED on Dec 10, 2018. #

        {Set the degeneracy of the alignment to be 0 (zero)}; # IN this algorithm, degeneracy = 0 means that the alignment should NOT contribute the probability (at least within this coordinate space).
        {and SKIP the subsequent processes};
}

        # If ( #{null columns} > 0 ),
        # set the degeneracy of the alignment to be 0 (zero), # IN this algorithm, degeneracy = 0 means that the alignment should NOT contribute the probability (at least within this coordinate space).
        # and SKIP the following three processes (+ checking the positional order of the swappable blocks … ADDED on Dec 10, 2018)!!


        # BEFORE DOING the next two processes,
        # examine whether any vertically equivalent blocks overlap or not:

my $if_overlap = 0;

foreach my $br (sort {$a <=> $b} keys %branch2up_or_down2equiv_blocks) { # Outer foreach-loop.
    my $up_or_down2equiv_blocks = $branch2up_or_down2equiv_blocks{$br};

    foreach my $ud (sort keys %{$up_or_down2equiv_blocks}) { # Middle foreach-loop.

        my @std_equiv_blocks = sort {$bds_blocks[$a]→[0] <=> $bds_blocks[$b]→[0]} @{$up_or_down2equiv_blocks{$ud}};

        my $ct_ebs = @std_equiv_blocks;
        for (my $k = 1; $k < $ct_ebs; $k++) { # Inner foreach-loop.

            my ($bl1, $bl2) = @std_equiv_blocks[$k-1, $k];

            if ($bds_blocks[$bl1]→[1]  + 1 >= $bds_blocks[$bl2]→[0]) { # The right-boundary of the block on the left touches or goes beyond the left-boundary of the block on the right.
                $if_overlap = 1;
                last;
            }
```

```
        }   # END of the inner foreach-loop.

        if ($if_overlap) { last; }

    }  # END of the middle foreach-loop.

    if ($if_overlap) { last; }

}  # END of the outer foreach-loop.

if ($if_overlap) {  # This means that the alignment changed its topology in an unacceptable way.

        {Make the degeneracy of this alignment zero.}  # IN this algorithm, degeneracy = 0 means that the alignment should NOT contribute the probability (at least within this coordinate space).
        &
        {Skip the following two processes.}
}

        # It's probably the best to do immediately before process (iv) (i.e., the actual "shift") & immediately after process (iv) (i.e., after all position/order changes associated with the "shift"),
        # check the number of equivalent alignments in the coordinate space as quickly as possible.
        #
        # For this purpose, we will use:
        #
        # @collectively_complementary_blocks, with
        #  {element} =
        # [a set of 3 2 or more blocks that are collectively complementary to one another,
        #  where vertically equivalent blocks are bundled together (into an anonymous array … OBSOLETE as of Nov 12, 2018)],
        # as well as:
        #
        # @block2coll_comple, where
        # @{$block2coll_comple[$bl]} lists
        # (the indices in @collectively_complementary_blocks of)
        # the sets of collectively complementary blocks that
        # the $bl th block belongs to; it is empty if there is no such set. … This array may actually be superfluous...

(NOTE: We also need to devise an algorithm to construct these two arrays. => See Appendix C D.)

Here is an algorithm:


        my @degenerate_sets_ccbs = (); # element = [$lbd_left, $rbd_left, $lbd_right, $rbd_right, \@left, \@right, \@including], where @left/right is the set of blocks on the left/right, @including is the set of blocks (horizontally) including both.
#       my $if_degenerate = 0; … OBSOLETE as of Nov 12, 2018.
#       my ($lbd_rlv, $rbd_rlv) = @{$bds_blocks[$bl]};
#
#    foreach my $indx_cc (@{$block2coll_comple[$bl]}) { # Outer foreach-loop (over the sets of collectively complementary blocks that $bl belongs to). #
#
#       my $set_cc_blks = $collectively_complementary_blocks[$indx_cc];
```

```
    foreach my $set_cc_blks (@collectively_complementary_blocks) {# Outer foreach-loop (over the sets of collectively complementary blocks. #
#    foreach my $src_set_cc_blks (@collectively_complementary_blocks) {# Outer foreach-loop (over the sets of collectively complementary blocks. #

#        my @sets_cc_blks = ([]); # element = [\@indices_connected_equiv_blks, $lbd, $rbd]
#
#        foreach my $equiv_blks (@{$src_set_cc_blks}) { # 1st middle foreach-loop (over the sets of vertically equivalent blocks). #
#
#            my $sets_conn_blks = bundle_connected_blocks (@{$equiv_blks}, @bds_blocks);  # Will be defined later...
#            my @new_sets_cc_blks = ();
#            while (my $set_ccbs = shift @sets_cc_blks) {
#                foreach my $set_cbs (@{$sets_conn_blks}) {
#                    my $cp_set_ccbs = copy (@{$set_ccbs});
#                    push @{$cp_set_ccbs}, $set_cbs;
#                    push @new_sets_cc_blks, $cp_set_ccbs;
#                }
#            }
#
#            @sets_cc_blks = @new_sets_cc_blks;
#        } # END of the "1st middle foreach-loop (over the sets of vertically equivalent blocks)". #
#
#        foreach my $set_cc_blks (@sets_cc_blks) {  # 2nd middle foreach-loop (over the sets of collectively complementary blocks).  … OBSOLETE as of Nov 12, 2018.

                # Sort the blocks in ascending order of their sizes. #
            my @std_set_cc_blks = sort {$bds_blocks[$a]→[1] - $bds_blocks[$a]→[0] <=> $bds_blocks[$b]→[1] - $bds_blocks[$b]→[0]} @{$set_cc_blks};

#            my $bl1 = $set_cc_blks→[0];
            my $bl1 = $std_set_cc_blks[0];
            my ($lbd1, $rbd1) = @{$bds_blocks[$bl1]};  # … RESURRECTED as of Nov 12, 2018.

#            my ($indices1_ceb, $lbd1, $rbd1) = @{$set_cc_blks→[0]}; # The 1st bundle of connected equivalent blocks. #

            my $ct_ccbs = @{$set_cc_blks};

            my @identical = my @including =  #my @included
            = my @left_flanking = my @right_flanking = my @overlapping = my @separated = ();

#            foreach my $bl2 (@{$set_cc_blks}) { # 1st inner foreach-loop (over constituent blocks).#
#                if ($bl2 == $bl1) { next; } # Skip the subject itself. #
#                my ($lbd2, $rbd2) = @{$bds_blocks[$bl2]};

            for (my $k=1; $k <$ct_ccbs; $k++) { # 1st inner foreach-loop (over bundles of connected equivalent blocks). #

#                my $ceb2 = $set_cc_blks→[$k];
#                my ($indices2_ceb, $lbd2, $rbd2) = @{$ceb2};

                my $bl2 = $std_set_cc_blks[$k];
                my ($lbd2, $rbd2) = @{$bds_blocks[$bl2]};

#                if (($rbd2 < $lbd_rlv-1) or ($rbd_rlv+1 < $lbd2)) {
```

```
                if (($rbd2 +1 < $lbd1) or ($rbd1 + 1 < $lbd2)) {

                    push @separated, $bl2;
#                   push @separated, $ceb2;
                    last;

                } elsif ($rbd1 +1 == $lbd2) { push @right_flanking, $bl2;
                } elsif ($rbd2 + 1 == $lbd1) { push @left_flanking, $bl2;

                } elsif ($lbd1 == $lbd2) {
                    if ($rbd1 < $rbd2) {
                        push @including, $bl2;

                    } else { # $rbd1 == $rbd2
                        push @identical, $bl2;
                    }

                } elsif ($lbd1 < $lbd2) {
                    push @overlapping, $bl2;
                    last;

                } else { # $lbd2 < $lbd1

                    if ($rbd2 < $rbd1) {
                        push @overlapping, $bl2;
                        last;

                    } else { # $rbd1 <= $rbd2
                        push @including, $bl2;
                    }
                }
            } # END of the "1st inner foreach-loop (over bundles of connected equivalent blocks)". #
#       }       # End of the 1st inner foreach-loop (over constituent blocks). #

            if (@separated > 0) { next; # If one or more blocks are separated from the smallest block, this set should NOT be degenerated.
            } elsif (@overlapping > 0) { next; # If one or more blocks overlap the smallest block (in a non-nested manner), this set should NOT be degenerated.

            } elsif ((@left_flanking == 0) and (@right_flanking == 0)) { next; # If no blocks flank the smallest block, this set should NOT be degenerated.
            } elsif ((@left_flanking > 0) and (@right_flanking >0)) { next; # If the smallest block is flanked on both sides, this set should NOT be degenerated.
            }

            my @flanking = (@left_flanking > 0) ? @left_flanking : @right_flanking;
            my ($lbd_fl, $rbd_fl) = @{$bds_blocks[$flanking[0]]};
            my $ct_flanking = @flanking;
            my $if_staggered = 0;
            for (my $i=1; $i< $ct_flanking; $i++) { # 2nd inner for-loop.
                my ($lbd3, $rbd3) = @{$bds_blocks[$flanking[$i]]};
                if (($lbd3 != $lbd_fl) or ($rbd3 != $rbd_fl)) {
                    $if_staggered = 1;
                    last;
                }
            } # END of the 2nd inner for-loop.
```

```
            if ($if_staggered) { next; } # For the set to be degenerated, the flanking blocks MUST have identical boundaries.

            my $lbd_joint = ($lbd1 < $lbd_fl) ? $lbd1 : $lbd_fl;
            my $rbd_joint = ($rbd1 < $rbd_fl) ? $rbd_fl : $rbd1;

            if (@including>0) {
                my $if_non_accommodating = 0;
                foreach my $bl3 (@including) { # The 3rd inner foreach-loop.
                    my ($lbd3, $rbd3) = @{$bds_blocks[$bl3]};
                    if (($lbd_joint < $lbd3) or ($rbd3 < $rbd_joint)) {
                        $if_non_accommodating = 1;
                        last;
                    }
                }               # End of the 3rd inner foreach-loop.

                if ($if_non_accommodating) { next; } # For the set to be degenerated, all blocks including the subject MUST accommodate the flanking blocks as well.
            }

            my @std_identical = sort {$a <=> $b} ($bl1, @identical);
            my @std_flanking = sort {$a <=> $b} @flanking;
            my @std_including = sort {$a <=> $b} @including;

            my ($lbd_left, $rbd_left, $lbd_right, $rbd_right, $std_left, $std_right, $std_including) =
                                                                    # ADDED on Nov 28, 2018.
            my @degenerate_set = (@left_flanking>0) ?
                ($lbd_fl, $rbd_fl, $lbd1, $rbd1, \@std_flanking, \@std_identical, \@std_including] :
                ($lbd1, $rbd1, $lbd_fl, $rbd_fl, \@std_identical, \@std_flanking, \@std_including] ;


                        # Finally check whether the collective blocks are really swappable or not. #
                        # ADDED on Nov 28, 2018.
            my $if_movable = 1;
            foreach my $bl5 (@{$std_left}) {
                # Check whether the block is movable to the position of the right collective block.
                my $bl_coord5 = $(curr_)bl_coords[$bl5];
                my ($lbd_coord5, $rbd_coord5) = @{$bds_bl_coord[$bl5]};
                if ($rbd_coord5 - $bl_coord5 < $rbd_right - $rbd_left) {
                    $if_movable = 0;
                    last;
                }
            }
            if ($if_movable) {
                foreach my $bl5 (@{$std_right}) {
                    # Check whether the block is movable to the position of the left collective block.
                    my $bl_coord5 = $(curr_)bl_coords[$bl5];
                    my ($lbd_coord5, $rbd_coord5) = @{$bds_bl_coord[$bl5]};
                    if ($lbd_coord5 - $bl_coord5  < $lbd_right - $lbd_left) {
                        $if_movable = 0;
                        last;
                    }
                }
            }
            if ($if_movable == 0) { next; }
                        # END of "ADDED on Nov 28, 2018."
```

```
            push @degenerate_sets_ccbs, \@degenerate_set;

#         } # End of the "2nd middle foreach-loop (over the sets of collectively complementary blocks)". ... OBSOLETE as of Nov 12, 2018.

    }  # END of "Outer foreach-loop (over the sets of collectively complementary blocks  #that $bl belongs to#)". #

            # NOTE ADDED on Nov 13, 2018: Come to think of it,
            # we could further restrict the sets of collectively complementary blocks
            # by imposing additional conditions on the horizontal sizes,
            #  that is, a set of ccbs MUST consist of
            # one or two sets, each of (horizontally) identically sized blocks, and the remainder (if at all),
            # which consists of blocks whose horizontal sizes are equal to or longer than the sum of those for the former two sets,
            # (or twice the horizontal size for the only one former set).


                    # Then, examine whether some degenerate sets can be chained or not. #

my @std_deg_sets = sort {$a->[0] <=> $b->[0]} @degenerate_sets_ccbs;
@degenerate_sets_ccbs = @std_deg_sets;
#my @degenerate_sets_ccbs = @std_deg_sets;
my @chains_deg_sets = ();
my $ct_cmplx_chains = 0; # The number of "complex" chains, each of which consists of two or more degenerate sets.

while (@degenerate_sets_ccbs>0) { # Outer while-loop. #

            my $chain_tail = shift @std_deg_sets;
            my @chain = ($chain_tail);
            my @remainder = ();
            while (my $subject = shift @degenerate_sets_ccbs) { # Inner while-loop. #

                    if ($chain_tail->[2] < $subject->[0]) {
                        @remainder = (@remainder, $subject, @degenerate_sets_ccbs);
                         last;
                    } elsif ($subject->[0] < $chain_tail->[2]) {
                        push @remainder, $subject;
                        next;
                    }

                if ($subject->[1] == $chain_tail->[3]) { # Now, the left blockset of the subject is positioned identically with the right blockset of the tail of the chain.
                    if (0 == compare_arrays (@{$chain_tail->[5]}, @{$subject->[4]}) { # The two blocksets are identical.
                        if (0 == compare_arrays (@{$chain_tail->[6]}, @{$subject->[6]}) { # The "including" blocksets in the two sets are also identical.
                            push @chain, $subject;
                            $chain_tail = $subject;
                            # next;
                            last;
                        }
                    }
                }

                    push @remainder, $subject;
```

```
        # my @degenerate_sets_ccbs = (); # element = [$lbd_left, $rbd_left, $lbd_right,
$rbd_right, \@left, \@right, \@including], where @left/right is the set of blocks on the left/right,
@including is the set of blocks (horizontally) including both.

        }  # END of the Inner while-loop. #

        push @chains_deg_sets, \@chain;

        if (@chain>1) {
            $ct_cmplx_chains++;
            last;
        }

        @degenerate_sets_ccbs = @remainder;

}   # END of the outer while-loop. #

my $degeneracy;
if ($ct_cmplx_chains>0) { # This case should actually be considered in a different coordinate
space(, in which the @left in the left ccb and the @right in the right ccb are merged).
        Set the degeneracy of this local alignment to be 0 (zero),
        & SKIP the next process!!

} else {
        $degeneracy = {the basic degeneracy at the point in the relevant coordinate subspace};
        for (1 .. scalar (@chains_deg_sets) ) { $degeneracy *= 2; } # Each degenerate set doubles
the degeneracy.
}



        # NEXT,  check whether the alignment topology changed or not, as quickly as possible.

        # Because the topological change MUST result from the "shift" of the current ($bl th)
block,
        # we should be able to focus on this block and its surroundings.


        # NOTE: Before checking the topological change,
        # you should check whether the indel probability is already recorded for either
        # @bl_coords + (1, 1,…, 1)   or  @bl_coords – (1, 1, .., 1).
        # If either probability is already recorded, just use it.
        # (But care must be exercised if the local alignment reaches either end of the whole
alignment.)
        # Otherwise, check the topological change, as follows.
        #
        # (NOTE: Merger and re-split between vertically identical blocks need not be checked here,
        #   because they should already have been examined in a preceding process.)

        # For this purpose, it would be convenient to construct:
```

@interfering_blocks, where

@{$interfering_blocks[$bl]} = (\@blocks, \@relations) stores information on blocks that can interfere with the $bl th block (or empty if it has no such blocks), where
$blocks[$k] is the index (or rank) of the $k th interfering block,
$relations[$k] is the relation of the $k th interfering block with the $bl th block.

Here, we will only consider the following relations:
'S' (for 'sibling/parent/child' of the $bl th block),
'Cp' (for 'complementary'),
MODIFIED on Jan 17, 2019. (In this case, though, we are concerned only with parsimonious indel histories)
'>(ch)' (for the $bl2 th block being 'vertically included' in, and an "effective child" of, the $bl1 th block),
'<(pa)' (for the $bl2 th block 'vertically including', and being the "effective parent" of, the $bl1 th block),
'>' (for the $bl2 th block being 'vertically included' in the $bl1 th block),
'<' (for the $bl2 th block 'vertically including' the $bl1 th block),
'=' (for 'vertically identical').

NOTES:
(1) Blocks with '**ONN**' relations cause NO topological changes, as far as we employ the current coordinate system;
(2) Also, blocks with **'ONCS'** relations cause NO topological changes, *as far as* we restrict ourselves to parsimonious indel histories (more precisely, parsimonious ancestral gap states);
(3) Also, *as far as* we restrict ourselves to parsimonious indel histories (more precisely, parsimonious ancestral gap states), the separating branches of blocks with the **'>'** or **'<'** relations with the $bl th block *must be* either the **child or parent** (**or a sibling**
#if they are children of a trivalent root
) of the separating branch of the $bl th block. # (As of Jan 17, 2019, this condition is trivially satisfied.)

In addition, it would also be convenient to have **@interfering_blocksets**, where **@{$interfering_blocksets[$bl]} = (\@blocksets, \@relations)** are nearly the same as above, except that **@{$blocksets[$k]}** is now a set of indices (or ranks) of the $k th interfering block set. (NOTE: Each @{$blockset[$k]} contains only blocks with the same horizontal size (this condition may be loosened **… examine later!!** **… DONE until (and including) 2019/01/25!!**) that form a (complementary) monophyletic group.)

For the moment, we will only consider the following relations:

'S' (for 'sibling/parent/child' of the $bl th block),
MODIFIED on Jan 17, 2019.
'>(ch)' (for the blockset being 'vertically included' in, and an "effective child" of, the $bl1 th block).
'>' (for the blockset being 'vertically included' in the $bl1 th block).

NOTES:
(5) As in note (3), the separating branch of the blockset with the **'>'** relation must be either the **child or parent** **or sibling** of the separating branch of the $bl th block; # As of Jan 17, 2019, this condition is automatically satisfied.
(6) Each blockset with the '**Cp**' relation likely involves a single block with the 'S' relation;
(7) Likewise, **'='** likely involves a single block with the 'S' relation;
(8) And **'<'** likely involves a single block with the '=', '>', and/or the 'S' relations.

[ADDED on Nov 20, 2018, after re-considering Figures SSSSSA10&11 in "figures_sppl3_bp1_ANEX.draft8.odp"]
Furthermore, we will also prepare **the "complement"** of @interfering_blocksets, denoted as **@cmpl_interfering_blocksets**, for which the subject is each constituent of each block-set recorded in @interfering_blocksets.
More precisely,
@{$cmpl_interfering_blocksets[$bl]} = (\@cmpl_blocksets, \@relations)

records block-blockset pairs in @interfering_blocksets in which **the $bl th block** is a constituent of @blockset = @{$interfering_blocksets[$bl2]->[0][$k2]} for some $bl2 and $k2.
(It is empty if the $bl th block is not a constituent of any block-sets.)
We will use the convention:
@{$cmpl_blocksets[$k]} = ($bl2, @blockset with $bl removed)
and
$relations[$k] = the **"complement"** of **$interfering_blocksets[$bl2]→[1][$k2]**.

Thus, the relations should be:
'S' if the corresponding relation in @interfering_blocksets is 'S',
MODIFIED on Jan 17, 2019.
'<(pa)' if the corresponding relation in @interfering_blocksets is '>(ch)'.
'<' if the corresponding relation in @interfering_blocksets is '>'.

The following are the **algorithms** to quickly check a topological change.

Let's assume that we have at hand:

```
($curr_lbd, $curr_rbd) = @{$relv_bds_block}; # $relv_bds_block = $bds_blocks[$bl];
($curr_lbd0, $curr_rbd0); # The boundaries of $bl BEFORE the "shift"-like move. # Added on 2019/01/25.
my $relv_interfering_blks = $interfering_blocks[$bl];
my $relv_interfering_blksets = $interfering_blocksets[$bl];
my $relv_cmpl_interfering_blksets = $cmpl_interfering_blocksets[$bl]; ... ADDED on Nov 20, 2018.

my $if_topological_change = 0; # Initialize the output variable (flag).
```

(1) Using @interfering_blocks,

```
if (@{$relv_interfering_blks}==0) { return; }

my ($inf_blks, $relations) = @{$relv_interfering_blks};
my $ct_inf_blks = @{$inf_blks};

for (my $k=0; $k<$ct_inf_blks; $k++) { # Outer for-loop (1).
my $bl2 = $inf_blks→[$k];
my ($lbd2, $rbd2) = @{$bds_blocks[$bl2]};
my $rel = $relations→[$k];

if ($rel eq 'S' ) { # 'sibling/parent/child' of the $bl th block

    if ($sh > 0) { # The block moved to the right.

                # ADDED on Jan 24, 2019. (1) #
            if (($lbd2 < $curr_lbd) and ($rbd2 < $curr_rbd)) { # The current block is now non-nested with the 2nd block.

                if (($lbd2 <= $curr_lbd0) and ($curr_rbd0  <= $rbd2)) { # The current block was horizontally included in the 2nd one.
                    $if_topological_change = 1;
                } elsif (($curr_lbd0 <= $lbd2) and ($rbd2 <= $curr_rbd0)) { # The current block horizontally included the 2nd one.
                    $if_topological_change = 1;
                }
                    # END of "ADDED on Jan 24, 2019. (1)" #
```

```
[[ OBSOLETE as of 2019/01/24. (1)
#
#                   if (($lbd2 + $sh == $curr_lbd) and ($rbd2 < $curr_rbd)) { # The current block, which horizontally included the 2nd block, now overlaps but is non-nested with the latter.
#                       $if_topological_change = 1;
#
#                   } elsif (($lbd2 < $curr_lbd) and ($rbd2 + $sh == $curr_rbd)) {  # The current block, which was horizontally included in the 2nd block, now overlaps but is non-nested with the latter.
#                       $if_topological_change = 1;
]] END of "OBSOLETE as of 2019/01/24. (1)"

                    } elsif (($curr_lbd == $lbd2) and ($curr_rbd <= $rbd2)) { # The current block, which overlapped but was non-nested with the 2nd block, now is horizontally included in the latter.
                        $if_topological_change = 1;

                    } elsif (($curr_lbd < $lbd2) and ($curr_rbd == $rbd2)) {# The current block, which overlapped but was non-nested with the 2nd block, now horizontally includes the latter.
                        $if_topological_change = 1;

[[ OBSOLETE as of 2019/01/24. (2a)
#
#                       # Added on 2019/01/23. #
#
#                   } elsif (($lbd2 + $sh < $curr_lbd) and ($rbd2 < $curr_rbd)) { # The current block, which may have horizontally included the 2nd block,  now overlaps but is non-nested with the latter.
#
#                       my %skipped = ($bl => 1, $bl2 => 1);
#                       my $ct_shifts = ct_necessary_shifts ($curr_lbd, $lbd2, $bl, @bds_blocks, @inter_block_relations, %skipped);
#
#                       if ($ct_shifts == 1) { $if_topological_change = 1; }
#
]] END of "OBSOLETE as of 2019/01/24. (2a)"

sub ct_necessary_shifts ($$$\@\@\%) {
        # Counts the number of "shift"s of $bl_sbj necessary
        # for moving the boundary, $bd_curr, to $bd_goal.
        # %{$skipped} = ($block_to_be_skipped => 1, ...)

        my ($bd_curr, $bd_goal, $bl_sbj, $bds_blocks, $inter_block_relations, $skipped) = @_;

        if ($bd_curr == $bd_goal) { return 0; }

        my $rels_w_sbj = $inter_block_relations->[$bl_sbj];
        my $ct_blks = @{$rels_w_sbj};
        my ($lb_mv, $rb_mv) = ($bd_curr < $bd_goal) ? ($bd_curr+1, $bd_goal) : ($bd_goal, $bd_curr-1);

        my %clm2set_rlv;
        for (my $bl3 = 0; $bl3 < $ct_blks; $bl3++) { # Outer for-loop (over $bl3).
            if (defined $skipped{$bl3}) { next; }
            my $rel = $rels_w_sbj->[$bl3];
            unless (($rel eq '<') or ($rel eq '<(pa)') or ($rel eq 'ONN') or ($rel eq 'ONCS') or (($rel eq '=') and ($bl3 < $bl_sbj)) ... Added on 2019/03/20. ) { next; }
            my ($lb3, $rb3) = @{$bds_blocks->[$bl3]};
            if (($rb_mv < $lb3) or ($rb3 < $lb_mv)) { next; }

            my $lb_intrsct = ($lb3 > $lb_mv) ? $lb3 : $lb_mv; # Added on 2019/01/24.
```

```
            my $rb_intrsct = ($rb3 < $rb_mv) ? $rb3 : $rb_mv; # Added on 2019/01/24.

            for (my $c = $lb_intrsct; $c <= $rb_intrsct; $c++) { # Inner for-loop (over columns, $c).
# MODIFIED on 2019/01/24.
#           for (my $c = $lb3; $c <= $rb3; $c++) { # Inner for-loop (over columns, $c).
                my $set_rlv = $clm2set_rlv{$c};
                unless (defined $set_rlv) { $set_rlv = $clm2set_rlv{$c} = []; }
                push @{$set_rlv}, $bl3;
            } # End of the inner for-loop (over columns, $c).
        }  # End of the outer for-loop (over $bl3).

        my $ct_shifts = 0;
        for (my $c = $lb_mv; $c <= $rb_mv; $c++) { # Outer for-loop (over columns, $c).
            unless (defined $clm2set_rlv{$c}) { $ct_shifts++; }
        } # End of the outer for-loop (over columns, $c).

        return $ct_shifts;

} # End of "sub ct_necessary_shifts ($$\@\@\%) {...}"


[[ OBSOLETE as of 2019/01/24. (2b)
#
#               } elsif (($lbd2 < $curr_lbd) and ($rbd2 + $sh < $curr_rbd)) {  # The current block,
which may have been horizontally included in the 2nd block, now overlaps but is non-nested with
the latter.
#
#                   my %skipped = ($bl => 1, $bl2 => 1);
#                   my $ct_shifts = ct_necessary_shifts ($curr_rbd, $rbd2, $bl, @bds_blocks,
@inter_block_relations, %skipped);
#
#                   if ($ct_shifts == 1) { $if_topological_change = 1; }
#
#
#                       # Also consider the possibility that
#                       # a block (or a set of blocks) vertically including $bl has either the left-bound
at $lbd2 + $sh + 1
#                       # or the right-bound at $rbd2 + $sh -1,
#                       # and "mediates" between $bl and the relevant boundary of $bl2. … Maybe
Later!!  (POSED on Jan 17, 2019) … DONE!!
#
#                       # End of "Added on 2019/01/23." #
]] END of "OBSOLETE as of 2019/01/24. (2b)"

                }

            } else { # The block shifted to the left.
                # The same as above, but with the left- and right-ends swapped.
            }

        } elsif ( ($rel eq 'Cp')  #  'complementary'
            or  ($rel eq '>(ch)')    # 'vertically included' in the $bl th block # Added '(ch)' on Jan
17, 2019.
            or  ($rel eq '<(pa)') # 'vertically including' the $bl th block # Added '(pa)' on Jan 17,
2019.
#           or  ($rel eq '=')  # 'vertically identical' --- Actually, it may be superfluous…
(OBSOLETE as of 2019/01/22)
            ) {
```

```
            if ($sh > 0) {# The block shifted to the right.

                if ($curr_rbd + 1 == $lbd2) { # The current block, which was separated from the 2nd block by one column, is now immediately adjacent to the latter.
                    $if_topological_change = 1;

                } elsif ($rbd2 + 1 == $curr_lbd0) { # The current block, which was immediately adjacent to the 2nd block, is now separated from the latter by one column. # MODIFIED on 2019/01/24.
#               } elsif ($rbd2 + 2 == $curr_lbd) { # The current block, which was immediately adjacent to the 2nd block, is now separated from the latter by one column.
                    $if_topological_change = 1;

                        # ADDED on Jan 24, 2019. (2) #
                } elsif ($curr_rbd + 1 < $lbd2) { # The current block now appears separated from the 2nd one, which is on the right.

                    my %skipped = ($bl => 1, $bl2 => 1);
                    my ($bl_sbj, $start, $goal) = ($rel eq '>(ch)') ? ($bl2,  $lbd2, $curr_rbd+1) : ($bl, $curr_rbd, $lbd2-1);
                    my $ct_shifts = ct_necessary_shifts ($start, $goal, $bl_sbj, @bds_blocks, @inter_block_relations, %skipped);

                    if ($ct_shifts == 0) { $if_topological_change = 1; } # The blocks are actually NOT separated!!

                } elsif ($rbd2 + 1 < $curr_lbd0) { # The current block appeared separated from the 2nd one, which is on the left.

                    my %skipped = ($bl => 1, $bl2 => 1);
                    my ($bl_sbj, $start, $goal) = ($rel eq '>(ch)') ? ($bl2,  $rbd2, $curr_lbd0 -1) : ($bl, $curr_lbd0, $rbd2+1);
                    my $ct_shifts = ct_necessary_shifts ($start, $goal, $bl_sbj, @bds_blocks, @inter_block_relations, %skipped);

                    if ($ct_shifts == 0) { $if_topological_change = 1; } # The blocks were actually NOT separated!!

                        # Also consider the possibility that
                        # a block (or a set of blocks) either vertically including or overlapping yet non-nested with $bl (or $bl2)  flanks $bl2.
                        # . … Maybe Later!!  (POSED on Jan 17, 2019) … DONE on 2019/01/24!!

                        # END of "ADDED on Jan 24, 2019. (2)" #
                }

            } else {# The block shifted to the left.
                # The same as above, but with the left- and right-ends swapped.
            }

        }

        if ($if_topological_change) { last; }

    } # END of the outer for-loop (1).

    if ($if_topological_change) { return $if_topological_change; }
```

```
# $inter_block_relations[$bl1]→[$bl2] = $relation,
# for the relation between the $bl1 th and $bl2 th blocks.
#
# Here, $relation can be:
#
#       'NIF' (for 'non-interfering'),
#       'S' (for 'sibling/parent/child' of the $bl th block),
#       'Cp' (for 'complementary'),
#       'ONN' (for 'overlapping yet non-nesting' NOR 'complementary-sibling/parent/child'),
#       'ONCS' (for 'overlapping yet non-nesting' but 'complementary-sibling/parent/child' of the
$bl th block),
#       '>' (for the $bl2 th block being 'vertically included' in the $bl1 th block),
#       '<' (for the $bl2 th block 'vertically including' the $bl1 th block),
#       '=' (for 'vertically identical').

(2) Using @interfering_blocksets,

        if (@{$relv_interfering_blksets} == 0) { return; }

        my ($inf_blksets, $relations2) = @{$relv_interfering_blocksets};
        my $ct_inf_bss = @{$inf_blksets};

    for (my $k=0; $k < $ct_inf_bss; $k++) { # Outer for-loop (2).
        my $blset = $inf_blksets→[$k];

        my ($lbd2, $rbd2) = @{$bds_blocks[$blset->[0]]};  # The ($lbd2, $rbd2) will be the
boundaries of the "horizontal" intersection of the blocks in the set.  (See Figures SSSSSA10&11 in
"figures_sppl3_bp1_ANEX.draft8.odp".)
        my $rel = $relations2→[$k];

        my $size_blset = @{$blset};


        if ($rel eq 'S') {# MOVED from below the inner for-loop (2),  (MODIFIED as of
2019/01/24. (1a)) #

            for (my $k=1; $k<$size_blset; $k++) { # Inner for-loop (2).
                my ($lbd3, $rbd3) = @{$bds_blocks[$blset→[$k]]};

                        # It may be better to 'extend' the boundaries if the block ,$blset→[$k] , is
flanked by some blocks vertically including it. .. Maybe later!! (POSED on Jan 17, 2019). …
Actually, taken into account below (in "MODIFIED as of 2019/01/24. (1b)").

                if ($lbd2 < $lbd3) { $lbd2 = $lbd3; }
                if ($rbd3 < $rbd2) { $rbd2 = $rbd3; }
            }   # END of the inner for-loop (2). … ADDED on Nov 20, 2018.

                    # MODIFIED as of 2019/01/24. (1b) #

            unless ($lbd2 <= $rbd2) { next; }

            if ((($lbd2 <= $curr_lbd) and ($curr_rbd <= $rbd2))
                or (($lbd2 <= $curr_lbd0) and ($curr_rbd0 <= $rbd2)) ) {

                # Identical to the above for ($rel eq 'S') in (1). #

            } else {
```

```
                my $if_min_blks = 0;
                for (my $k=1; $k<$size_blset; $k++) { # Inner for-loop (3).
                    my $bl3 = $blset→[$k];
                    my ($lbd3, $rbd3) = @{$bds_blocks[$bl3]};
                    if (($lbd2 == $lbd3) and (($rbd2 == $rbd3)) {
                        $if_min_blks = 1;
                        last;
                    }
                }  # END of the inner for-loop (3).

                if ($if_min_blks) {
                     # Identical to the above for ($rel eq 'S') in (1). #
                }
            }

        } elsif ($rel eq '>(ch)') {

            my $ct_topo_changes = my $ct_including = 0;
            if ($sh > 0) {
                for (my $k=0; $k <$size_blset; $k++) { # Inner for-loop (5) (over blocks in the set).
                    my $bl3 = $blset→[$k];
                    my ($lb3, $rb3) = @{$bds_blocks[$bl3]};
                    if (($lb3 <= $curr_rbd0 +1) and ($curr_lbd <= $rbd3 + 1)) {
                        $ct_including++;


                    } elsif ($curr_rbd +1 == $lb3) { # The current block, which was separated from
the 3rd block by one column, is now immediately adjacent to the latter.
                        $ct_topo_changes++;

                    } elsif ($rb3 +1 == $curr_lbd0) { # The current block, which was immediately
adjacent to the 3rd  block, is now separated from the latter by one column.
                        $ct_topo_changes++;

                    } elsif ($curr_rbd +1 < $lb3) { # The current block now appears separated from
the 3rd  one, which is on the right.
                        my %skipped = ($bl => 1, $bl3 => 1);
                        my $ct_shifts = ct_necessary_shifts ($lb3, $curr_rbd+1, $bl3, @bds_blocks,
@inter_block_relations, %skipped);

                        if ($ct_shifts > 0) {
                            last; # The blocks are indeed separated.
                        } else {
                            $ct_topo_changes++; # The blocks are actually NOT separated!!
                        }

                    } elsif ($rb3 +1 < $curr_lbd0) { # The current block appeared separated from the
3rd  one, which is on the left.
                        my %skipped = ($bl => 1, $bl3 => 1);
                        my $ct_shifts = ct_necessary_shifts ($rb3, $curr_lbd0-1, $bl3,
@bds_blocks, @inter_block_relations, %skipped);

                        if ($ct_shifts > 0) {
                             last; # The blocks were indeed separated.
                        } else {
                             $ct_topo_changes++; # The blocks were actually NOT separated!!
                        }
```

```
                    } else {
                        last; # Because we are sure that there should be NO topological change between this block-set and $bl, leave the loop here.
                    }
                }   # END of the inner for-loop (5) (over blocks in the set).

            } else { # if ($sh <0)
                    # Identical to the "if ($sh>0) {...}" block above, with the left and right swapped. #
            }

            if (($ct_including + $ct_topo_changes == $size_blset) and ($ct_topo_changes>0)) {
                $if_topological_change = 1;
                last;
            }

        } END of "if ($rel eq 'S') {…} elsif ($rel eq '>(ch)') {...}".


[[  OBSOLETE as of 2019/01/24. (3)
#       if ($lbd2 <= $rbd2) { … Condition ADDED on Nov 20, 2018.
#
#               # If the "horizontal" intersection is not empty. #
#
#           if ($rel eq 'S') { # 'sibling/parent/child' of the $bl th block
#
#               # Identical to the above for ($rel eq 'S') in (1). #
#
#           } elsif ($rel eq '>(ch)') { # 'vertically included' in the $bl th block # Added '(ch)' on Jan 17, 2019.
#
#               # Identical to the above for ($rel eq '>(ch)') in (1). # # Added '(ch)' on Jan 17, 2019.
#           }
#
#       } … END of the "Condition ADDED on Nov 20, 2018."
]]  END of "OBSOLETE as of 2019/01/24. (3)"

                    # END of "MODIFIED as of 2019/01/24. (1a,b)" #

#       if ($if_topological_change) {  # Examine whether the constituent blocks align perfectly.
#           my $size_blset = @{$blset};
#           for (my $k=1; $k<$size_blset; $k++) { # Inner for-loop (2).
#               my ($lbd3, $rbd3) = @{$bds_blocks[$blset→[$k]]};
#               if (($lbd2 == $lbd3) and ($rbd2 == $rbd3)) { next; }
#               $if_topological_change = 0;
#               last;
#           }   # END of the inner for-loop (2).
#       } # END of "if ($if_topological_change) {...}" … OBSOLETE as of Nov 20, 2018.

        if ($if_topological_change) { last; }

    }  # END of the outer for-loop (2).

    if ($if_topological_change) { return $if_topological_change; }


[ BEGINNING of "ADDED on Nov 20, 2018, after re-considering Figures SSSSSA10&11 in "figures_sppl3_bp1_ANEX.draft8.odp"" ]
```

(3) Using @cmpl_interfering_blocksets, # Also consider the cases where vertically 'large' blocks interfere. .. Maybe later!! (POSED on Jan 17, 2018). … DONE in the case ($rel eq '<(pa)') (on 2019/01/25)!!

(In the case ($rel eq 'S'), however, it is difficult to incorporate such 'large' blocks. So, we leave it as a future task to actually consider this case. (My hunch is that the results will not change significantly even if we incorporate the effects of such 'large' blocks.))

```
        if (@{$relv_cmpl_interfering_blksets} == 0) { return; }

        my ($inf_cblksets, $relations3) = @{$relv_cmpl_interfering_blocksets};
        my $ct_inf_cbss = @{$inf_cblksets};

    for (my $k=0; $k < $ct_inf_cbss; $k++) { # Outer for-loop (3).
        my $cblset = $inf_cblksets→[$k];
        my $size_cblset = @{$cblset};

        my ($lbd2, $rbd2) = @{$bds_blocks[$cblset→[0]]}; # The block paired with the block-set. ($bl2 = $cblset→[0]).
        my $rel = $relations3→[$k];

        if ($rel eq 'S') { # 'sibling/parent/child' of the block-set containing the $bl th block

                        # MODIFIED on 2019/01/25. (1) #

            my ($lbd3, $rbd3) = @{$bds_blocks[$cblset→[1]]};
            for (my $k=2; $k <$size_cblset; $k++) { # Inner for-loop (3) (over the remaining blocks in the c-block-set).
                my ($lbd5, $rbd5) = @{$bds_blocks[$cblset→[$k]]};
                if ($lbd3 < $lbd5) { $lbd3 = $lbd5; }
                if ($rbd5 < $rbd3) { $rbd3 = $rbd5; }
                if ($rbd3 < $lbd3) { last; }
            } # End of the inner for-loop (3) (over the remaining blocks in the c-block-set).

            if ($rbd3 < $lbd3) { next; } # Topology should NOT change!!

            my ($max_curr_lbd, $min_curr_lbd) = ($curr_lbd < $curr_lbd0) ?
                                ($curr_lbd, $curr_lbd0)  : ($curr_lbd0, $curr_lbd) ;

            my ($max_curr_rbd, $min_curr_rbd) = ($curr_rbd > $curr_rbd0) ?
                                ($curr_rbd, $curr_rbd0) : ($curr_rbd0, $curr_rbd) ;

            if (($lbd3 <= $lbd2) and ($rbd2 <= $rbd3)) { # $bl2 is horizontally included in the c-block-set (excluding $bl). => Focus on the relationship between $bl2 and $bl. (We do NOT know yet, which of $bl and $bl2 horizontally includes the other.)

                # Identical to the above for ($rel eq 'S') in (1). #

            } elsif (($lbd3 <= $max_curr_lbd) and ($max_curr_rbd <= $rbd3)) { # Both before and after the change, $bl is horizontally included in the c-block-set (excluding $bl). => Focus on the relationship between $bl2 and $bl. (In this case, $bl will NOT include $bl2 horizontally.)

                if ($sh >0) { # The block shifted to the right.
                    if (($lbd2 < $curr_lbd) and ($rbd2 < $curr_rbd)) { # $bl is now non-nested with $bl2.

                                if (($lbd2 <= $curr_lbd0) and ($curr_rbd0  <= $rbd2)) { # The $bl was horizontally included in $bl2.
```

```
                    $if_topological_change = 1;
                }

            } elsif (($curr_lbd == $lbd2) and ($curr_rbd <= $rbd2)) { # $bl, which overlapped but was non-nested with $bl2, now is horizontally included in the latter.
                $if_topological_change = 1;

            }

        } else { # if ($sh <0), that is, The block shifted to the left.
                # The same as above, but with the left- and right-ends swapped.
        }


    } elsif (($lbd3 <= $curr_lbd) and ($curr_rbd <= $rbd3)) { # After the change, $bl is horizontally included in the c-block-set (excluding $bl), but it was not before the change. => Check if $bl2 also horizontally includes $bl (after the change).

        if (($lbd2 <= $curr_lbd) and ($curr_rbd <= $rbd2)) { $if_topological_change = 1; }

    } elsif (($lbd3 <= $curr_lbd0) and ($curr_rbd0 <= $rbd3)) { # Before the change, $bl was horizontally included in the c-block-set (excluding $bl), but it is not after the change. => Examine whether $bl2 also horizontally included $bl (before the change).

        if (($lbd2 <= $curr_lbd0) and ($curr_rbd0 <= $rbd2)) { $if_topological_change = 1; }

    } elsif (($min_curr_lbd <= $lbd3) and ($rbd3 <= $min_curr_rbd)) { # Both before and after the change, $bl horizontally includes the c-block-set (excluding $bl). => Topology will NEVER change, no matter what.
        next;

    } elsif ( (($curr_lbd <= $lbd3) and ($rbd3 <= $curr_rbd)) # After the change, $bl does horizontally includes the c-block-set (excluding $bl), but it did not before the change.
        or
         (($curr_lbd0 <= $lbd3) and ($rbd3 <= $curr_rbd0)) # Before the change, $bl did horizontally include the c-block-set (excluding $bl), but it does not after the change.
        ) {

        unless (($lbd2 <= $lbd3) and ($rbd3 <= $lbd2)) { next; } # $bl2 does NOT include the c-block-set. => Topology will NOT change!!

            # Examine whether some blocks in the c-block-set have the "minimum" boundaries, [$lbd3, $rbd3]. #
        my $if_min_blks = 0;
        for (my $k=1; $k <$size_cblset; $k++) { # Inner for-loop (5) (over the blocks in the c-block-set).
            my ($lbd5, $rbd5) = @{$bds_blocks[$cblset→[$k]]};
            if (($lbd3 == $lbd5) and ($rbd3 == $rbd5)) {
                $if_min_blks = 1;
                last;
            }
        } # End of the inner for-loop (5) (over the blocks in the c-block-set).

        if ($in_min_blks) { $if_topological_change = 1; }

    } # END of "if (($lbd3 <= $lbd2) and ($rbd2 <= $rbd3)) {…} elsif …. {}".
```

```
                # OBSOLETE as of 2019/01/25. (1) #
#               # Identical to the above for ($rel eq 'S') in (1). #
#
#           if ($if_topological_change) {
#
#               # Check if all the remaining blocks "horizontally" include the 2nd block. #
#
#               for (my $k=1; $k<$size_cblset; $k++) { # Inner for-loop (3) (over the remaining
blocks in the block-set).
#                   my ($lbd3, $rbd3) = @{$bds_blocks[$cblset->[$k]]};
#                   if (($lbd2 < $lbd3) or ($rbd3 < $rbd2)) { # Unless the 3rd block "horizontally"
includes the 2nd block. #
#                       $if_topological_change = 0;
#                       last;
#                   }
#
#               } # END of the "Inner for-loop (3) (over the remaining blocks in the block-set)".
#           }
                # END of "OBSOLETE as of 2019/01/25. (1)" #

                    # END of "MODIFIED on 2019/01/25. (1)" #


        } elsif ($rel eq '<(pa)') { # 'vertically including' the block-set containing the $bl th block #
Added '(pa)' on Jan 17, 2019.

                # Identical to the above for ($rel eq '<(pa)') in (1). #

            if ($if_topological_change) {

                # Check if all the remaining blocks are "horizontally" connected with the 2nd block. #

                for (my $k=1; $k<$size_cblset; $k++) { # Inner for-loop (6) (over the remaining
blocks in the block-set).
                    # MODIFIED on 2019/01/25. (2) #

                    my $bl3 = $cblset->[$k];
                    my ($lbd3, $rbd3) = @{$bds_blocks[$bl3]};
                    # my ($lbd3, $rbd3) = @{$bds_blocks[$cblset->[$k]]}; # OBSOLETE as of
2019/01/25.


                    if ($rbd3 < $lbd2 -1) { # $bl3 is separated from, and on the left of, $bl2.

                        my %skipped = ($bl => 1, $bl3 => 1);
                        my $ct_shifts = ct_necessary_shifts ($rbd3, $lbd2 -1, $bl3, @bds_blocks,
@inter_block_relations, %skipped);
                        if ($ct_shifts >0) {
                            $if_topological_change = 0;
                            last;
                        }

                    } elsif ($rbd2 + 1 < $lbd3) { # $bl3 is separated from, and on the right of, $bl2.

                        my %skipped = ($bl => 1, $bl3 => 1);
                        my $ct_shifts = ct_necessary_shifts ($lbd3, $rbd2 +1, $bl3, @bds_blocks,
@inter_block_relations, %skipped);
                        if ($ct_shifts >0) {
```

```
                        $if_topological_change = 0;
                        last;
                    }
                }

            # OBSOLETE as of 2019/01/25. (2) #
#               if (($rbd3 +1 < $lbd2) or ($rbd2 +1 < $lbd3)) { #  If the 3rd block is "horizontally
separated" from the 2nd block. #
#                   $if_topological_change = 0;
#                   last;
#               }
            # END of "OBSOLETE as of 2019/01/25. (2)" #

                # END of "MODIFIED on 2019/01/25. (2)" #

            } # END of the "Inner for-loop (6) (over the remaining blocks in the block-set)".
        }
    }

    if ($if_topological_change) { last; }

} # END of the outer for-loop (3).

    return $if_topological_change;

# @{$cmpl_interfering_blocksets[$bl]} = (\@cmpl_blocksets, \@relations)
# records block-blockset pairs in @interfering_blocksets
# in which the $bl th block is a constituent of
# @blockset = @{$interfering_blocksets[$bl2]->[0][$k2]} for some $bl2 and $k2.
# (It is empty if the $bl th block is not a constituent of any block-sets.)
# We will use the convention:
# @{$cmpl_blocksets[$k]} = ($bl2, @blockset with $bl removed)
# and
# $relations[$k] = the "complement" of $interfering_blocksets[$bl2]→[1][$k2].
#
# Thus, the relations should be:
#       'S' if the corresponding relation in @interfering_blocksets is 'S',
#       '<' if the corresponding relation in @interfering_blocksets is '>'.

[ END of  "ADDED on Nov 20, 2018, after re-considering Figures SSSSSA10&11 in
"figures_sppl3_bp1_ANEX.draft8.odp"" ]



# Extend the quadruple, \@triple = [\@set_columns, \@bds_blocks, \@bl_coords,
$ct_null_columns],
# into an septet octet (i.e., an seven eight-tuple):
#
# \@sepoctet = [@triple, \@lb_sorted_set, \@orders_lb, \@rb_sorted_set, \@orders_rb].


            # ADDED on Jan 17, 2019. (1) #

if ($if_topological_change) {

    {Examine whether either the alignment with all coordinates less than the current ones by one, or
that with all coordinates more than the current ones by one, has the boundary status identical to the
```

current one and is already assigned **the indel component of its log-probability**.}; # ADDED on Jan 18, 2019. (1)

```
    if (already assigned, and identical boundary status) { # ADDED on Jan 18, 2019. (1)

            { Use the assigned value };
    } else {

            my $code_topology = encode_alignment_topology (@bds_blocks,
@inter_block_relations, {other necessary things});  # See Appendix F-sppl2 H.  (Create it later!!)

            my $ln_prob_indels = $ctopo2ln_prob_indels{$code_topology};

            if (defined $ln_prob_indels) {
                    {Store $ln_prob_indels into $set_ln_probs_indels[$coord_bl0]→[$coord_bl1]...
[$coord_bl{B-1}]. };
            } else {
                    {Compute $ln_prob_indels. };
                    $ctopo2ln_prob_indels{$code_topology} = $ln_prob_indels;
                    => {Store $ln_prob_indels into $set_ln_probs_indels[$coord_bl0]→[$coord_bl1]...
[$coord_bl{B-1}]. };
            }

    } # END of "ADDED on Jan 18, 2019. (1)"
}
                    # END of "ADDED on Jan 17, 2019. (1)" #


#{ WITHHELD on Nov 4, 2018:
#       my %rlv_bl_rels = %{$inter_block_relations[$bl]}
#
#       my $onn = $rlv_bl_rels{'ONN'};
#       my $oncs = $rlv_bl_rels{'ONCS'};
#
#       if ((defined $onn) or (defined $oncs)) {
#
#               # If necessary, swap the relevant block with the overlapping block(s),
#               # before shifting the coordinate(s). #
#       }
#
#       my $vincluded = $rlv_bl_rels{'>'}; # Included in the relevant block.
#       my $vincluding = $rlv_bl_rels{'<'}; # Including the relevant block.
#       my $vequiv = $rlv_bl_relv{'='};
#
#       if (defined $vincluding) {
#               # If necessary, adjust the coordinate of the relevant block. #
#       }
#       if (defined $vequiv) {
#               # If necessary, adjust the coordinate of the relevant block. #
#               # In addition, if necessary, also adjust the coordinate of the overlapping block. #
#       }
#
#       if (defined $vincluded) {
#               # If necessary, adjust the coordinate of the relevant block
#               # and also the coordinate of the overlapping block. #
#       }
#} … END of "WITHHELD on Nov 4, 2018"
```

APPENDIX A: An algorithm to cluster a set of sequences (or external nodes) into a minimum number of monophyletic groups (and possibly the complement of a monophyletic group).

This should be implemented via a round-trip traversal of the tree.
In short,
(1) Via a bottom-up traversal, cluster neighboring external nodes into each monophyletic group they belong to; and
(2) If possible, via a top-down traversal, merge neighboring monophyletic groups into the complement of yet another monophyletic group.

The **actual implementation** is as follows.

(0) Preliminary:

```
my %monophyl_roots; # = ($node => 1 for the root of a current monophyletic group,
                    #                0 otherwise.)

foreach my $node (set of external nodes) { $monophyl_roots{$node} = 0: }
foreach my $node (set of subject external nodes) { $monophyl_roots{$node} = 1; }

# It may be better to skip (1) and (2)
if (a) all external nodes are the subjects (=> $top_node is the monophyletic root),
if (b) all external nodes but one are the subjects (=> the non-subject node is the lower-bound of the complementary monophyletic group),
or if (c) only one external node is the subject (=> the subject node is the root of the only one monophyletic group).
```

(1) The bottom-up part:

```
foreach my $node (all internal nodes in descending order of the depth) {

        my $children = $node2ch→{$node};
        ((defined $children) and (@{$children}>0)) or next; # Skip external nodes.
        my $counter = 0;
        foreach my $ch (@{$children}) {  ($monophyl_roots{$ch} == 1) and ($counter++); }
        if ($counter == @{$children}) {
                $monophyl_roots{$node} = 1;
                foreach my $ch (@{$children}) { $monphyl_roots{$ch} == 0; }  # It may be better to delete the hash element.
        } else {
                $monophyl_roots{$node} = 0; # Unnecessary if the above deletion is done.
        }

}
```

(2) The top-down part:

```
my $complement_lower_bound;
my $curr_node = $top_node; # Start with the root of the entire phylogenetic tree.

while (1) {  # This while-loop will be performed ONLY IF the top node has two or more children, and UNLESS $monophyl_roots{$top_node} == 1. #

        my $children = $node2ch→{$curr_node};
        ((defined $children) and (@{$children}>0)) or last; # End the loop if the current node is external.
```

```
        my @non_mp_roots = ();
        my @mp_roots = ();
        foreach my $ch (@{$children}) {
                if ($monophyl_roots{$ch} == 1) {
                   push @mp_roots, $ch;
                } else {
                   push @non_mp_roots, $ch;
                }
        }

        if (1 == @non_mp_roots) { # All except one of the children are the roots of monophyletic groups.
                $complement_lower_bound = $curr_node = $non_mp_roots[0]; # Update the lower-bound of the complement monophyletic group (and the current node).
                foreach my $ch (@mp_roots) { $monophyl_roots{$ch} = 0; } # It may be better to delete the hash element.
        } else {
                last;
        }
}
```

(3) Finish:

```
my @monophyl_roots = ();
foreach my $node (all nodes) { ($monophyl_roots{$node} == 1) and (push @monophyl_roots, $node); }
```

=> @monophyl_roots contains the roots of monophyletic groups (that are not included in the complement monophyletic group), and $complement_lower_bound (if defined) is the lower-bound (i.e., the separating branch) of the largest possible complement monophyletic group.

APPENDIX B: Computing column-wise probabilities taking advantage of the (complementary) monophyletic groups constructed as in APPENDIX A

This can be done by applying Felsenstein's pruning algorithm (in a bottom-up manner)
along **a truncated (or "pruned") tree**,
whose ***root*** is the complementary monophyletic group (or the original root if there is no such group),
and whose ***external nodes (i.e., leaves)*** are the roots of the monophyletic groups.

(We assume that the conditional probabilities concerning each monophyletic groups, and the joint probabilities concerning the complementary monophyletic group, if at all, are already available.)

Probably, the easiest solution would be to *explicitly* construct **the "pruned" tree** first (this has to be done *only once* given a set of gap-blocks (in an input alignment)),
and to compute the column-wise probabilities along the truncated tree.

(1) Construct the "pruned" tree:

```
        # Specify the root of the "pruned" tree. #
my $root_pruned = (defined $complement_lower_bound) ? $complement_lower_bound : $topnode_id ;  # If there is no complement monophyletic group, use the top-node (i.e., root) of the original tree.

my %leaves_pruned;
foreach my $cw_mp_roots (@set_cw_monophyl_roots) {
```

```
        foreach my $mp_root (@{$cw_mp_roots}) { $leaves_pruned{$mp_root} = 1; }
}

my %node2ch_prnd;
my %node2pa_prnd;
my %node2depth_prnd;

my @subjcts = ($root_pruned);
my $depth = 0;
while (@subjects > 0) {
        my $depth++;
        my @new_sbjcts = ();
        foreach my $node (@subjects) {
            $node2depth_prnd{$node} = $depth;
            if (defined $leaves_pruned{$node}) { next; }
            my $children = $node2ch{$node};
            unless ((defined $children) and (@{$children}>0)) { next; }
            push @new_sbjcts, @{$children};

            $node2ch_prnd{$node} = $children;
            foreach my $ch (@{$children}) {  $node2pa{$ch} = $node; }
        }
        @subjects = @new_sbjcts; # Update the set of the subject nodes.
}

my %br2attr_prnd;
foreach my $br (keys %node2pa_prnd) { $br2attr_prnd{$br} = $br2attr; }
```

(2) Compute the column-wise probability:

```
        # Specify the nucleotide frequencies at the root of the "pruned" tree. #
my @ntfreqs_root_pruned = (defined $complement_lower_bound) ?
  ( ($ss eq '-') ? @{$node2ntfreqs{$complement_lower_bound}} :
@{$br2set_sw_extjnt_probs{{$complement_lower_bound}→[$s]} )
        : @ntfreqs_root ;   # Here, $s is the index of the relevant site (or class-specific column).
        # Actually, $s = $label2class_sp_clms{$labels[$cl_w_cmpl_mp]}→{$ss},
        # with $ss = $csd_clms[$cl_w_cmpl_mp].

if (0 == scalar keys %node2ch_prnd) {  # In this case, there should be either a single class
($root_pruned = $topnode_id) or two complementary classes ($root_pruned != $topnode_id).
        my $cl_w_mp;
        my $root_mp;
        for (my $cl=0; $cl < $CT_CLS; $cl++) {
            my $cw_mp_roots = $set_cw_monophyl_roots[$cl];
            if (@{$cw_mp_roots}== 0) { next; }
            if (@{$cw_mp_roots} > 1) {
                return {signal indicating a falure};
            }
            if (defined $root_mp) {
                return {signal indicating a falure};
            }
             $cl_w_mp = $cl;
             $root_mp = $cw_mp_roots->[0];
        }
        if ($complement_lower_bound != $root_mp) {
                return {signal indicating a falure};
        }
```

```
        my $ss2 = $csd_clms[$cl_w_mp];
        my $s2 = ($ss2 eq '-') ? '-' : $label2class_sp_clms{$labels[$cl_w_mp]}->{$ss2};
        my @sw_cond_probs = ($ss2 eq '-') ? (1, 1, 1, 1) :
            @{$node2set_sw_cond_probs{$root_mp}→[$s2]};
        my $cw_prob = 0;
        for (my $i=0; $i< $CT_NTS; $i++) { $cw_prob += $ntfreqs_root_pruned[$i] *
$sw_cond_probs }

        return $cw_prob;
}

        # INITIALIZE the ingredients. #

my %node2sw_extcond_probs_prnd;

for (my $cl=0; $cl <$CT_CLS; $cl++) {
        my $ss3 = $csd_clms[$cl];
        my $s3 = ($ss3 eq '-') ? '-' : $label2class_sp_clms{$labels[$cl_w_mp]}->{$ss3};
        my $cw_monophyl_roots = $set_cw_monophyl_roots[$cl];
        foreach my $root_mp (@{$cw_monophyl_roots}) {
            my @sw_xcond_probs = ($ss3 eq '-') ? (1, 1, 1, 1) :
                @{$node2set_sw_extcond_probs{$root_mp}→[$s3]};
            $node2sw_extcond_probs_prnd{$root_mp} = \@sw_xcond_probs;
        }
}

        # Compute the extended conditional probabilities in a bottom-up manner. #

foreach my $node (sort {$node2depth_prnd{$b} <=> $node2depth_prnd{$a}} keys
%node2depth_prnd) {
        if ($node != $root_pruned) { next; }
        my $children = $node2ch_prnd{$node};
        unless (defined $children) { next; }

            # Compute the conditional probabilities. #
        my @cond_probs = (1, 1, 1, 1);
        foreach my $ch (@{$children}) {
            my $sw_xcond_probs_ch = $node2sw_extcond_probs_prnd{$ch};
            for (my $i=0; $i < $CT_NTS; $i++) { $cond_probs[$i] *=
$sw_xcond_probs_ch→[$i]; }
        }

            # Compute the extended conditional probabilities. #
        my $tr_mtrx = $br2tr_mtrx→{$node};
        my @xcond_probs = ();
        for (my $i=0; $i <$CT_NTS; $i++) {
            my $row_tr_probs = $tr_mtrx→[$i];
            my $xcond_prob = 0;
            for (my $j=0; $j < $CT_NTS; $j++) { $xcond_prob += $row_tr_probs→[$j] *
$cond_probs[$j]; }
            $xcond_probs[$i] = $xcond_prob;
        }
        $node2sw_extcond_probs_prnd{$node} = \@xcond_probs;
}

        # Compute the conditional probabilities of the column. #
my @cw_cond_probs = @ntfreqs_root_pruned;
```

```
foreach my $ch (@{$node2ch{$root_pruned}}) {
        my $sw_xcond_probs = $node2sw_extcond_probs_prnd{$ch};
        for (my $i=0; $i <$CT_NTS; $i++) { $cw_cond_probs *= $sw_xcond_probs→[$i]; }
}
        # Compute the total probability of the column. #
my $cw_prob = 0;
for (my $i=0; $i <$CT_NTS; $i++) { $cw_prob += $cw_cond_probs[$i]; }

return $cw_prob;

# %label2class_sp_clms
# @set_cw_monophyl_roots, where @{$set_cw_monophyl_roots[$cl]} lists the roots of
# the monophyletic groups belonging to the $cl th class (= empty if the $cl th class contains
# no monophyletic group).
# $complement_lower_bound, which is the "lower-bound" (or separating branch) of the
#  complement monophyletic group (= undef if there is no such complement).
# $cl_w_cmpl_mp, which is the ID of the class accommodating a complement monophyletic group
# (= undef if there is no such complement).
```

APPENDIX C: Classifying sequences according to what gap-blocks affect them

Here, we will construct @class_labels (& %label2class) & @affected_classes.
The former is necessary for defining classes of sequences according to the gap-blocks affecting them.
The latter is necessary for clarifying the effects of each gap-block in a compact manner.

(1) Constructing @class_labels (& %label2class):

Input: @info_blocks, with
%{$info_blocks[$bl]} = (
'branch' => $branch_ID, # (for separating branch)
'side' => 'U' or 'L', # when the gap-blocks is on the 'upper'/'lower'-side of the branch.
),
or something similar.
& %id2name = ($node_ID => $seq_name, …) for external nodes, if necessary,
& @seqnames, which stores the names (IDs) of all sequences.
& %node_id2seq_indx = ($node_ID => $indx_seq (in @seqnames), …) for external nodes.

```
my @cmplx_block_effects = (); # An auxiliary output. #
for (my $i=0; $i < $CT_SEQS; $i++) { push @cmplx_block_effects, ''; } # Initialize
@cmplx_block_effects. #

for (my $bl = 0l $bl < $B; $bl++) {

        my $info_block = $info_blocks[$bl];
                # The following two commands will change according to the structure of
@info_block.
        my $br = $info_block→{'branch'};
        my $side = $info_block→{'side'};

        my $ext_offsprings = fetch_ext_offsprings_nr ($br, %node2ch);

        my ($stat_offsprings, $stat_others) = ($side eq 'U') ? ('F', 'T') : ('T', 'F');
                # If ($side eq 'U'), the gap-block is on the 'upper-side', that is, the offsprings are
NOT affected.
                # Otherwise, the gap-block is on the 'lower-side', that is, the offsprings ARE
affected.
```

```
        my @block_effects = ();
        foreach my $nodeid (@{$ext_offsprings}) {
                my $seq_indx = $node_id2seq_indx{$nodeid};
                $block_effects[$seq_indx] = $stat_offsprings;
        }
        for (my $i=0; $i< $CT_SEQS; $i++) {
                my $effect = $block_effects[$i];
                $cmplx_block_effects[$i] .= (defined $effect) ? $effect : $stat_others;
        }

}

my %label2class;
for (my $i=0; $i < $CT_SEQS; $i++) {
        my $label = $cmplx_block_effects[$i];
        my $indices_seqs_in_class = $label2class{$label};
        unless (defined $indices_seqs_in_class) {
           $indices_seqs_in_class = $label2class{$label} = [];
        }
        push @{$indices_seqs_in_class}, $i;
}

my @class_labels = sort keys %label2class;
my $CT_CLS = scalar (@class_labels);
```

(2) Constructing @affected classes:

```
#my @decomp_labels = (); # An auxiliary array. #
#foreach my $label (@class_labels) { my @dc_label = split //, $label; push @decomp_labels, \
@dc_label; } … Turned out to be UNNECESSARY...

my @affected_classes = ();
for (my $bl = 0; $bl< $B; $bl++) { push @affected_classes, []; } # Initialize @affected_classes. #

for (my $cl=0; $cl < $CT_CLS; $cl++) {
        my @dc_label = split //, $class_labels[$cl];
        for (my $bl=0; $bl< $B; $bl++) {
            if ($dc_label[$bl] eq 'T') { push @{$affected_classes[$bl]}, $cl; }
        }
}

# @class_labels = ('TTT', 'TTF', 'TFT', 'TFF', …., 'FFF'); # enumerate all non-empty classes.
#     #  $ct_classes = @class_labels; #{classes}
# %label2class = ($class_label => \@indices_seqs_in_class, ….); # For all non-empty classes.
@indices_seqs_in_class = () if the class is empty.
#
# @affected_classes = (\@classes_affected_by_block1, \@classes_affected_by_block2, …);
#      # Just for convenience.
# NOTE that \@classes_affected_by_blocki contains the indices of the relevant classes in
@class_labels.
```

APPENDIX C D: Exhaustively listing the sets of complementary blocks

Here we will invent an **algorithm** to construct the following two arrays:

@collectively_complementary_blocks, each of whose elements is
a set of 2 or more blocks that are collectively complementary to one another;

@block2coll_comple, where
@{$block2coll_comple[$bl]} lists (the indices in @collectively_complementary_blocks of)
the sets of collectively complementary blocks that the $bl th block belongs to;
it is empty if there is no such set.

(1) Constructing @collectively_complementary_blocks:

This can be done via a bottom-up traversal, plus an examination at the root.

Assume that the following **input hash** is available:

my **%br2U_or_L2blocks** = ($branch_ID => {'U' => \@blocks_upper_side, .'L' => \
@blocks_lower_side}, ….); # @blocks_upper/lower_side stores the IDs (ranks) of gap-blocks that are separated by $branch_ID and on the 'upper'/'lower'-side of the branch.

my @collectively_complementary_blocks = (); # Initialization. #

(i) Bottom-up traversal:
my %br2cmp_blocks_lower; #An auxiliary hash, = ($branch_ID => \@cmp_blocks_lower, …) where @cmp_blocks_lower stores the composite gap-blocks that are on the 'lower'-side of $branch_ID.

```
foreach my $br (sort { $br2depth{$b} <=> $br2depth{$a} } keys %br2depth) {

        my @cmp_blocks_lower = ();
        my $U_or_L2blocks = $br2U_or_L2blocks{$br};
        if ((defined $U_or_L2blocks) and (defined $U_or_L2blocks->{'L'})){
              foreach my $bl (@{$U_or_L2blocks->{'L'}}) { push @cmp_blocks_lower, [$bl]; }
        }

        my $children = $node2ch{$br};
        unless ((defined $children) and (@{$children}>0)) { # The branch is external.
            if (@cmp_blocks_lower>0) {
                $br2cmp_blocks_lower{$br} = \@cmp_blocks_lower;
            }
            next;
        }

        my @new_cblocks = ([]);
        my $if_absent = 0;
        foreach my $ch (@{$children}) {
            my $cblocks_lower_ch = $br2cmp_blocks_lower{$ch};
            unless ((defined $cblocks_lower_ch) and (@{$cblocks_lower_ch}>0)) {
                $if_absent = 1;
                last;
            }
            my @newnew_cblocks = ();
            foreach my $new_cblock (@new_cblocks) {
                foreach my $cblock_lch (@{$cblocks_lower_ch}) {
                    my @cp_new_cblock = @{$new_cblock};
                    push @cp_new_cblock, @{$cblock_lch};
                    push @newnew_cblocks, \@cp_new_cblock;
                }
            }
            @new_cblocks = @newnew_cblocks; # Update @new_cblocks. #
```

```
        }

        if ($if_absent == 0) { # Every child branch possesses composite blocks on its “lower-side”.
            push @cmp_blocks_lower, @new_cblocks;
        }

        if (@cmp_blocks_lower == 0) { next; } # No composite blocks are on the “lower-side” of $br. #

        $br2cmp_blocks_lower{$br} = \@cmp_blocks_lower;

        unless ((defined $U_or_L2blocks) and (defined $U_or_L2blocks->{‘U’}))){ next; } # No gap-blocks are on the “upper-side” of $br. #
        my $blocks_upper = $U_or_L2blocks->{‘U’};

                # Construct collectively complementary blocks containing blocks on the “upper-side” of $br. #
        foreach my $cblocks_lower (@cmp_blocks_lower) {
            foreach my $block_upper (@{$blocks_upper}) {
                my @ccblock = @{$cblocks_lower};
                push @ccblock, $block_upper;

                        # Filter the @ccblock according to the block sizes.
                        # (Assume the existence of @block_sizes, where $block_sizes[$bl] gives the horizontal size of the $bl th block.)
                        # (ADDED on Dec 16, 2018).
                my @std_ccblock = sort {$block_sizes[$a] <=> $block_sizes[$b] } @ccblock;
                my $size1st = $block_sizes[$std_ccblock[0]];
                my ($size2nd, $size3rd);
                foreach my $bl (@std_ccblock) {
                        my $size = $block_sizes[$bl] ;
                        ($size == $size1st) and next;
                        unless (defined $size2nd) {
                            $size2nd = $size;
                            next;
                        }
                        ($size == $size2nd) and next;
                        $size3rd = $size;
                        last;
                }
                if ((defined $size3rd) and ($size3rd < $size1st + $size2nd)) { next; } # This set of “complex-complementary blocks” can NEVER yield degenerate configurations.
                        # (END of “ADDED on Dec 16, 2018”).

                push @collectively_complementary_blocks, \@ccblock;
            }
        }

}
```

(ii) Final examination at the root:

```
my $children_root = $node2ch{$topnode_id};

my $if_absent_r = 0;
my @ccblocks_r = ([]);
foreach my $ch (@{$children_root}) {
```

```
        my $cblocks_lch = $br2cmp_blocks_lower{$br};
        unless ((defined $cblocks_lch) and (@{$cblocks_lch}>0)) {
            $if_absent_r = 1;
            last;
        }
        my @new_ccblocks_r = ();
        foreach my $ccb_r (@ccblocks_r) {
             foreach my $cblk_lch (@{$cblocks_lch}) {
                 my @new_ccb_r = @{$ccb_r};
                 push @new_ccb_r, @{$cblk_lch};
                 push @new_ccblocks_r, \@new_ccb_r;
             }
        }
        @ccblocks_r = @new_ccblocks_r;  # Update @ccblocks_r. #
}

if (($if_absent_r == 0) and (@ccblocks_r>0)) { # Every child branch of the root has composite blocks on its "lower"-side. #
                        # (REPLACED on Dec 16, 2018). #

#       push @collectively_complementary_blocks, @ccblocks_r ;

                        # Filter the ccblocks in @ccblocks_r according to the block sizes.
                        # (Assume the existence of @block_sizes, where $block_sizes[$bl] gives the horizontal size of the $bl th block.)
        foreach my $ccblock (@ccblocks_r) { # Outer foreach-loop (over ccblocks on @ccblocks_r).
                my @std_ccblock = sort {$block_sizes[$a] <=> $block_sizes[$b] } @{$ccblock};
                my $size1st = $block_sizes[$std_ccblock[0]];
                my ($size2nd, $size3rd);
                foreach my $bl (@std_ccblock) {
                        my $size = $block_sizes[$bl] ;
                        ($size == $size1st) and next;
                        unless (defined $size2nd) {
                           $size2nd = $size;
                           next;
                        }
                        ($size == $size2nd) and next;
                        $size3rd = $size;
                        last;
                }
                if ((defined $size3rd) and ($size3rd < $size1st + $size2nd)) { next; } # This set of "complex-complementary blocks" can NEVER yield degenerate configurations.

                push @collectively_complementary_blocks, $ccblock ;

        } # END of the outer foreach-loop (over ccblocks on @ccblocks_r).

                        # (END of "REPLACED on Dec 16, 2018").

}
```

(2) Constructing @block2coll_comple:

```
my $CT_CCBS = scalar (@collectively_complementary_blocks);

my @block2coll_comple = ();
```

```
for (my $bl=0; $bl<$B; $bl++) { $block2coll_comple[$bl] = []; }  # Initialization. #

for (my $i = 0; $i < $CT_CCBS; $i++) {

        my $ccb = $collectively_complementary_blocks[$i];
        foreach my $bl (@{$ccb}) { push @{$block2coll_comple[$bl]}, $i; }
}

# @{$block2coll_comple[$bl]} lists (the indices in @collectively_complementary_blocks of)
# the sets of collectively complementary blocks that  the $bl th block belongs to;
# it is empty if there is no such set.
```

APPENDIX E: Dividing coordinate space according to alignment degeneracies

NOTE!!! (added on Dec 10, 2018): The algorithms below will be REPLACED with ## a simpler and faster measure, which was devised around Dec 5, 2018 (and recorded on Dec 10, 2018).

ADDED on Nov 28, 2018.

Here, (1) we determine the coordinate ranges in which only particular blocks can be swapped with one another.

```
        # Using @bds_bl_coords and the initial positions of the blocks,
        # given by @init_bds_blocks =
        #               (The set of block boundaries for the initial local alignment).
```

Then, (2) for each coordinate range, determine the degeneracy according to the groups of swappable blocks.

… The point in (2) must be to constrain the possibilities by considering the "isolated" blocks first, then, the blocks that become "isolated" after the "isolation" of the first group of blocks,

These algorithms will be incorporated in the **subroutine**:

```
        my $sub_degeneracies = sub_degeneracies_in_sub_coordinate_space (@{$ebs},
@init_bds_blocks, @init_bl_coords, @bds_bl_coords,
@inter_block_relations);
                # $sub_degeneracies→[$x_0]...[$x_{$ct_ebs}] is the sub-degeneracy
                # assigned to the coordinates ($x_0, …, $x_{$ct_ebs-1}) of the $ebs→[0], …,
$ebs→[$ct_ebs-1] th blocks, respectively.
                # (Actually, the "…," includes vertically equivalent but non-swappable blocks as
well, in order to consider ALL blocks whose ranks are between $ebs→[0] and $ebs→[$ct_ebs-1].)
```

(1) Dividing the coordinate sub-space according to the accessibility of the swappable blocks.

```
# (a) Because the subject blocks behave in the same manner
# when "interacting" with blocks that are
# vertically including, or overlapping but non-nested with, the subjects,
# it is convenient to "virtually move" all such non-subject blocks to the right-end of the alignment
# (even if their coordinate boundaries do NOT actually allow it).
# (And ignore other blocks, such as those which are vertically included in, complementary to,
siblings of, or non-interfering with, the subject, because they do NOT affect the positions of the
subjects.)

# (b) BEWARE of the "asymmetric" effects of non-swappable blocks in between swappable ones!!
```

```
# (In short, such blocks will NOT affect the positional ranges of swappable blocks (initially) on the left, but WILL affect those on the right, if they interact each other.)
                # END of "ADDED on Nov 28, 2018." #

                # ADDED on Nov 29, 2018. #
# (c) On the other hand,
# if a non-swappable, vertically equivalent block is on the left or on the right of
# ALL swappable blocks, it affects ALL swappable ones in the same manner.
# Thus, we can ignore the existence of such blocks in this algorithm.

(o) Initialize the 'working boundaries' of the blocks.

my @working_bds_blocks = copy (@init_bds_blocks);

our $R_VEND = 1000000; # The unnecessary blocks will be virtually moved to this value. # ADDED on Nov 30, 2018. #

(i) Virtually move all blocks vertically including or overlapping but non-nested with the subject blocks to the right-end.

my @including_or_overlapping = ();
my %already;
foreach my $bl (@{$ebs}) {
        my $relation_w_bl = $inter_block_relations[$bl];
        for (my $bl2=0; $bl2<$B; $bl2++) {
            if ($bl2 == $bl) { next; }
            if (defined $already{$bl2}) { next; }
            my $rel = $relation_w_bl→[$bl2];
            if (($rel eq '<') or ($rel eq 'ONN') or ($rel eq 'ONCS')) {
        # 'ONN' (for 'overlapping yet non-nesting' NOR 'complementary-sibling/parent/child'),
        # 'ONCS' (for 'overlapping yet non-nesting' but 'complementary-sibling/parent/child'
                push @including_or_overlapping, $bl2;
                $already{$bl2} = 1;
            }
        }
}

        # REPLACED on Nov 30, 2018. #
my @std_including_or_overlapping = sort {$a <=> $b } @including_or_overlapping ; # Sort them in order of their ranks, so that 'larger' gap-blocks will come earlier. #
#my @std_including_or_overlapping = sort {$init_bds_blocks[$b]->[0] <=> $init_bds_blocks[$a]→[0] } @including_or_overlapping ; # Sort them in descending order of the left-boundary. #


my %moved_to_right; # ADDED on Nov 30, 2018. #
foreach my $bl (@std_including_or_overlapping) {

                # REPLACED on Nov 30, 2018. #
        my $working_bds_bl = $working_bds_blocks[$bl];
        my ($lbd, $rbd) = @{$working_bds_bl};
        # my ($lbd, $rbd) = @{$working_bds_blocks[$bl]};

        my $size_bl = $rbd - $lbd + 1; # ADDED on Nov 30, 2018.

        my $relation_w_bl = $inter_block_relations[$bl];

        for (my $bl2=0; $bl2 < $B; $bl2++) {
```

```
            if ($bl2 == $bl) { next; }
            if (defined $moved_to_right{$bl2}) { next; }  #ADDED on Nov 30, 2018. #
            my $rel = $relation_w_bl→[$bl2];

            unless ( ($rel eq '>') or (($rel eq '=') and ($bl2 > $bl)) ) { next; } # $bl2 is vertically
NEITHER included in NOR equivalent to (and lower-ranked than)  $bl.# REPLACED and
MOVED DOWNWARD on Nov 30, 2018. #

                # REPLACED on Nov 30, 2018. #
            my $working_bds_bl2 = $working_bds_blocks[$bl2]
            my ($lbd2, $rbd2) = @{$working_bds_bl2};
            # my ($lbd2, $rbd2) = @{$working_bds_blocks[$bl2]};

            if ($rbd2 < $rbd) { next; } # $bl2 is on the left of $bl.

        # END of "ADDED on Nov 29, 2018". #

                # ADDED on Nov 30, 2018. #

                # REPLACED and MOVED DOWNWARD on Nov 30, 2018. #
            unless ( ($rel eq '>') or (($rel eq '=') and ($bl2 > $bl)) or ($rel eq 'ONN') or ($rel eq
'ONCS')) { next; } # $bl2 is vertically NEITHER included in, equivalent to (and lower-ranked
than), NOR  $bl.

                # Move the right-end, and also the left-end if applicable, of $bl2
                # to the left by the size of $bl. #
            $working_bds_bl2→[1] -= $size_bl;
            if ($rbd < $lbd2) { $working_bds_bl2→[0] -= $size_bl; }
        }

        $working_bds_bl→[0] = $working_bds_bl→[1] = $R_VEND;
        $moved_to_right{$bl} = 1;
}
```

(ii) Determine the accessible regions of swappable blocks and vertically equivalent blocks in between them.

(a) First, assume that no non-swappable vertically equivalent blocks are in between swappable blocks. # ADDED on Dec 7, 2018. #

```
my @block_sizes = ({given});  # $block_sizes[$bl] is the size of the $bl th block. # ADDED on
Dec 6, 2018. #
my $eb_lm = $ebs→[0]; # The leftmost swappable block.
my $eb_rm = $ebs→[$#{$ebs}]; # The rightmost swappable block.

                # REVISED on Dec 2, 2018. => further on Dec 6, 2018. #
my @sets_raw_regions_lbds = ();
        # @{$sets_raw_regions_lbds[$k]→[$b]} = ($bd_lbd, $coord_bd, $ct_r_blks), where gives
        # $bd_lbd is athe boundaryies of the left-bound of the raw regions (the last element is the
right boundary, the rest are the left boundaries) of the $ebs→[$k] th block (in terms of the column
number, i.e., the coordinate assigned to the local alignment),
        # $coord_bd is the corresponding coordinate (assigned to the $ebs→[$k] th block),
        # and $ct_r_blks is the number of swappable blocks that were initially on the right but that is
on the left when swapped. # ADDED on Dec 6, 2018, REVISED on Dec 7, 2018. #

for (my $eb = $eb_lm; $eb <= $eb_rm; $eb++) {
for (my $k=0; $k < @{$ebs}; $k++) {
```

```
        my ($lbd0, $rbd0) = @{$working_bds_blocks[$ebs->[$k]]};
        # my ($lbd0, $rbd0) = @{$working_bds_blocks[$eb]};

        my $init_bl_coord = $init_bl_coords->[$ebs->[$k]];
        my ($lbd_bl_coord, $rbd_bl_coord) = @{$bds_bl_coords→[$ebs->[$k]]};
        # my $init_bl_coord = $init_bl_coords->[$eb];
        # my ($lbd_bl_coord, $rbd_bl_coord) = @{$bds_bl_coords→[$eb};

                # ADDED on Dec 6 & 7, 2018. #
        my $lbd_lbd = $lbd0 + ($lbd_bl_coord - $init_bl_coord);
        my $ct_r_blks = 0; # The number of swappable blocks on the left initially but on the right
when swapped.
        if ($k > 0) {
            $lbd_lbd -= $block_sizes[$ebs→[$k-1]];
            $ct_r_blks++;
        }
        my $rbd_lbd = $lbd0 + ($rbd_bl_coord - $init_bl_coord);
        # my @raw_regions_lbd = ($lbd_lbd, $rbd_lbd);
        my @raw_regions_lbd = ([$lbd_lbd, $lbd_bl_coord, $ct_r_blks], [$rbd_lbd, $rbd_bl_coord,
0]);
        for (my $k2 = $k-2; $k2 >= 0; $k--) {
            $lbd_lbd -= $block_sizes[$ebs→[$k2]];
            $ct_r_blks++;
            # unshift @raw_regions_lbd, $lbd_lbd;
            unshift @raw_regions_lbd, [$lbd_lbd, $lbd_bl_coord, $ct_r_blks];
        }

        $sets_raw_regions_lbds = \@raw_regions_lbd;

}


my @sets_regions_lbds = ([0, $CT_CLMS-1, []]);
        # The array stores regions defined by the left-bounds of the gap-blocks.
        # elements = [$leftmost_clm, $rightmost_clm, \@set_regions_lbds], where
        # $left/rightmost_clm is the number (i.e., index in the local alignment)
        # of the left/right-most column in the region in question;
        # @{$set_regions_lbds[$i]} = ($left_bound, $right_bound, $ct_r_blks)
        # for the left-boundary of the #($eb_lm + $i)
        # $ebs→[$i] th block
        # (= empty if the block in question is not accommodated).

(Probably, @sets_regions_rbds, which stores regions defined by the right-bounds, will NOT be
necessary, because the right-bounds should be uniquely determined from the corresponding left-
bounds UNLESS the gap-blocks overlap, and because configurations with overlapping (vertically-
equivalent) gap-blocks WILL be ignored from the final probability computation in the coordinate
space at hand.)
# my @sets_regions_lbds = ([]); # element at the final stage = @set_regions_lbds, with
                                # @{$set_regions_lbds[$i]} = ($left_bound, $right_bound)
                                # for the left-boundary of the ($eb_lm + $i) th block
                                # (= empty if the block in question is not accommodated).
# my @sets_regions_rbds = ([]); # element at the final stage = @set_regions_rbds, with
                                # @{$set_regions_rbds[$i]} = ($left_bound, $right_bound)
                                # for the right-boundary of the ($eb_lm + $i) th block
                                # (= empty if the block in question is not accommodated).
                … Still, Needs be re-considered!!
                # END of "REVISED on Dec 2, 2018". #
```

```
# for (my $eb = $eb_lm; $eb <= $eb_rm; $eb++) {
for (my $k=0; $k < @{$ebs}; $k++) { # MODIFIED on Dec 7, 2018. #

        # my ($lbd0, $rbd0) = @{$working_bds_blocks[$eb]};
        my $raw_regions_lbd = $sets_raw_regions_lbds[$k];
        my $ct_raw_rgs = @{$raw_regions_lbd};
        my $ct_curr_regions = @sets_regions_lbds;
                # ADDED on Dec 7, 2018. #
        my @new_sets_regions_lbds = ();
        my $rr=0;
        while (($rr < $ct_raw_rgs) and (@sets_regions_lbds>0)) {

            my $raw_region = $raw_regions_lbd->[$rr];
            my ($bd_lbd, $coord_bd, $ct_r_blks) = @{$raw_region};
            my $info_set_regions = shift @sets_regions_lbds;
            my ($leftmost_clm, $rightmost_clm, $set_regions_lbds) = @{$info_set_regions};

            if (($bd_lbd < $leftmost_clm) or
                ($bd_lbd == $leftmost_clm) and ($rr < $ct_raw_rgs-1)) ) {
                # The raw boundary is on the left of the current region.
                $rr++;
                next;
            } elsif (($rightmost_clm < $bd_lbd) or
                  (($rightmost_clm == $bd_lbd) and ($rr == $ct_raw_rgs-1)) ) {
                if ($rr ==0) {
                        # The $k th swappable block is still on the right of the current region. #
                    $set_regions_lbds->[$k] = [];
                } else {
                        # The current boundary of the $k th block passed over the current region. #
                    my ($bd_lbd_prev, $coord_bd_prev, $ct_r_blks_prev1)
                        = @{$raw_regions_lbd->[$rr-1]};

                        # RESTARTED on Dec 8, 2018. #

                    my ($leftmost_clm_prev, $rightmost_clm_prev, $prev_set_regions_lbds)
                        = @{$new_sets_regions_lbds[$#new_sets_regions_lbds]};
                    my ($left_bound_prev, $right_bound_prev, $ct_r_blks_prev2)
                        = @{$prev_set_regions_lbds->[$k]};

                    my ($left_bound, $right_bound, $ct_r_blks_prev) =
                        (defined $set_regions_lbds->[$k]) ? @{$set_regions_lbds->[$k]} : ();

                    unless (defined $left_bound) {
                        if ($coord_bd == $coord_bd_prev) { $left_bound = $right_bound_prev;  # +1
?
                        } else {  $left_bound = $right_bound_prev + 1;
                        }
                    }
                    unless (defined $right_bound) {
                        if ($coord_bd == $coord_bd_prev) { $right_bound = $right_bound_prev;
                        } else {
                            $right_bound = $bd_lbd_prev + ($rightmost_clm - $bd_lbd_prev) *
($coord_bd - $coord_bd_prev) / ($bd_lbd - $bd_lbd_prev);
                        }
                    }
                    unless (defined $ct_r_blks_prev) { $ct_r_blks_prev = $ct_r_blks_prev1; }
```

```
                $set_regions_lbds->[$k] = [$left_bound, $right_bound, $ct_r_blks_prev];

            }
            push @new_sets_regions_lbds, $info_set_regions;
            next;
        }

            # The current boundary of the $k th block splits the current region. #

        # ($leftmost_clm, $rightmost_clm, $set_regions_lbds) = @{$info_set_regions};

        my $lm_clm1 = $leftmost_clm;
        my $rm_clm2 = $rightmost_clm;
        my ($rm_clm1, $lm_clm2);
        my @set_regions1_lbds = my @set_regions2_lbds = ();

        if ($rr == 0) { # The left-most boundary. #

            ($rm_clm1, $lm_clm2) = ($bd_lbd-1, $bd_lbd);

            for (my $k2=0; $k2<$k; $k2++) {
                my $region_lbd = $set_regions_lbds->[$k2];
                if (@{$region_lbd}>0) {
                    my ($left_bound1, $right_bound2, $ct_r_blks12) = @{$region_lbd};
                    my $bound12 = ((defined $right_bound2) and ($right_bound2 <
$left_bound1)) ? $left_bound1 : $left_bound1 + ($bd_lbd - $leftmost_clm);
                    $set_regions1_lbds[$k2] = [$left_bound1, $bound12-1, $ct_r_blks12];
                    $set_regions2_lbds[$k2] = [$bound12, $right_bound2, $ct_r_blks12];
                } else {
                    $set_regions1_lbds[$k2] = [];
                    $set_regions2_lbds[$k2] = [];
                }
            }
            my ($left_bound, $right_bound, $ct_r_blks0) = @{$set_regions_lbds->[$k]};
            unless (defined $ct_r_blks0) { $ct_r_blks0 = $ct_r_blks; }
            $set_regions1_lbds[$k] = [];
            $set_regions2_lbds[$k] = [$coord_bd, $right_bound, $ct_r_blks0];

        } elsif ($rr == $ct_raw_rgs -1) { # The right-boundary. #

            ($rm_clm1, $lm_clm2) = ($bd_lbd, $bd_lbd+1);

            for (my $k2=0; $k2<$k; $k2++) {
                my $region_lbd = $set_regions_lbds->[$k2];
                if (@{$region_lbd}>0) {
                    my ($left_bound1, $right_bound2, $ct_r_blks12) = @{$region_lbd};
                    my $bound12 = ((defined $right_bound2) and ($right_bound2 <
$left_bound1)) ? $left_bound1 : $left_bound1 + ($bd_lbd - $leftmost_clm);
                    $set_regions1_lbds[$k2] = [$left_bound1, $bound12, $ct_r_blks12];
                    $set_regions2_lbds[$k2] = [$bound12+1, $right_bound2, $ct_r_blks12];
                } else {
                    $set_regions1_lbds[$k2] = [];
                    $set_regions2_lbds[$k2] = [];
                }
            }
            my ($left_bound, $right_bound, $ct_r_blks0) = @{$set_regions_lbds->[$k]};
            unless (defined $ct_r_blks0) { $ct_r_blks0 = $ct_r_blks; }
            $set_regions1_lbds[$k] = [$left_bound , $coord_bd, $ct_r_blks0];
```

```
                $set_regions2_lbds[$k] = [];

            } else {  # The left-boundaries other than the left-most one. #

                ($rm_clm1, $lm_clm2) = ($bd_lbd-1, $bd_lbd);

                for (my $k2=0; $k2<$k; $k2++) {
                    my $region_lbd = $set_regions_lbds→[$k2];
                    if (@{$region_lbd}>0) {
                        my ($left_bound1, $right_bound2, $ct_r_blks12) = @{$region_lbd};
                        my $bound12 = ((defined $right_bound2) and ($right_bound2 <
$left_bound1)) ? $left_bound1 : $left_bound1 + ($bd_lbd - $leftmost_clm);
                        $set_regions1_lbds[$k2] = [$left_bound1, $bound12-1, $ct_r_blks12];
                        $set_regions2_lbds[$k2] = [$bound12, $right_bound2, $ct_r_blks12];
                    } else {
                        $set_regions1_lbds[$k2] = [];
                        $set_regions2_lbds[$k2] = [];
                    }
                }
                my ($left_bound, $right_bound, $ct_r_blks0) = @{$set_regions_lbds→[$k]};
                $set_regions1_lbds[$k] = [$left_bound, $coord_bd-1, $ct_r_blks0];
                $set_regions2_lbds[$k] = [$coord_bd, $right_bound, $ct_r_blks];
            }
            # ($bd_lbd, $coord_bd, $ct_r_blks) = @{$raw_region};

            push @new_sets_regions_lbds, [$lm_clm1, $rm_clm1, \@set_regions_lbds1];
            unshift @sets_regions_lbds, [$lm_clm2, $rm_clm2, \@set_regions_lbds2];
            $rr++; # ADDED on Dec 9, 2018.

        # @{$sets_raw_regions_lbds[$k]→[$b]} = ($bd_lbd, $coord_bd, $ct_r_blks), where
        # $bd_lbd is a boundary of the left-bound of the raw region (the last element is the right
boundary, the rest are the left boundaries) of the $ebs→[$k] th block (in terms of the column
number, i.e., the coordinate assigned to the local alignment),
        # $coord_bd is the corresponding coordinate (assigned to the $ebs→[$k] th block),
        # and $ct_r_blks is the number of swappable blocks that were initially on the right but that is
on the left when swapped.
        }

                # Process the left-over regions. (ADDED on Dec 9, 2018.) #

        while (@sets_regions_lbds>0) {
                my $info_set_regions = shift @sets_regions_lbds;
                my $set_regions_lbds = $info_set_regions→[2];
                $set_regions_lbds→[$k] = [];
                push @new_sets_regions_lbds, $info_set_regions;
        }

                # Update the set of the regions of the left-bounds. (ADDED on Dec 8, 2018.) #
        @sets_regions_lbds = @new_sets_regions_lbds;
}
```

(b) When some non-swappable vertically equivalent blocks are in between swappable blocks... # ADDED on Dec 7, 2018. #

NOTE that the positions of the ranges of swappable blocks will depend on the positions of the non-swappable blocks (as in Fig SSSSA13 of "figure_sppl_3_bp1_ANEX.xxxx.odp").

==> WILL be *seriously* considered later, i.e., when it becomes necessary...

(2) Assigning the degeneracy to each point (actually, each region) in the coordinate sub-space.

(a) First, assume that no non-swappable vertically equivalent blocks are in between swappable blocks. # ADDED on Dec 7, 2018. #

RESTARTED on Dec 9, 2018.

(i) Using @sets_regions_lbds, divide the range of each swappable block into sub-regions again.

```
my $ct_ebs = @{$ebs};
my $ct_regions_lbds = @sets_regions_lbds;

my @sets_divided_ranges_lbds = ();
foreach (1 .. $ct_ebs) { push @sets_divided_ranges_lbds, []; }
my @sets_cts_r_blks = ();
foreach (@{$ebs}) { push @sets_cts_r_blks, []; }

for (my $i=0 ; $i < $ct_regions_lbds; $i++) {

        my $set_regions_lbds = $sets_regions_lbds[$i]→[2];

        for (my $k=0; $k < $ct_ebs; $k++) {
            my $info_region_lbd = $set_regions_lbds→[$k];
            if (@{$info_region_lbd}>0) {
                push @{$sets_divided_ranges_lbds[$k]}, $i;
                push @{$sets_cts_r_blks[$k]}, $info_region_lbd->[2];
            }
        }
}

my @set_cts_div_rngs = ();
for (my $k=0; $k < $ct_ebs; $k++) {
        $set_cts_div_rngs[$k] = scalar (@{$sets_divided_ranges_lbds[$k]});
}
```

(ii) Using the "ingredients" prepared in (i), as well as @sets_regions_lbds,
assign the degrees of degeneracy to ALL sub-regions in the direct-product space of the ranges of the swappable blocks (defined by their left-bounds).

```
my @indices_div_rgs = ();  # $indices_div_rgs[$k] specifies the divided range of the $k th swappable block. #
for (my $k=0; $k < $ct_ebs; $k++) { $indices_div_rgs[$k] = 0; }
my $indx_relv_blk = $ct_ebs-1;

my @stack = ();
for (0 .. $ct_ebs) { my @copy = @indices_div_rgs; push @stack, \@copy; }
my $wrk_indices_div_rgs = pop @stack;

my @degeneracies = initialize_set_degeneracies ();
        # $degeneracies[$k_0]→[$k_1]...[$k_{$ct_ebs-1}] = \{degeneracy in the sub-range defined by the indices, ($k_0, $k_1, …, $k_{$ct_ebs-1})}.

while (1) {
```

```
        if ($wrk_indices_div_rgs->[$indx_relv_blk] == $set_cts_div_rngs[$indx_relv_blk]) {
                # Specify the "next" sub-region within the direct product,
                # when the relevant range reached the rightmost sub-range. #

            if ($indx_relv_blk == 0) { last; } # END the exploration. #

            $indx_relv_blk--;
            $wrk_indices_div_rgs = pop @stack;
            $wrk_indices_div_rgs->[$indx_relv_blk]++;
             next;
        }
        if ($indx_relv_blk < $ct_ebs-1) { # Restore the stack. #
            for (my $k = $indx_relv_blk+1; $k < $ct_ebs; $k++) {
                my @copy = @{$wrk_indices_div_rgs};
                push @stack, \@copy;
            }
            $indx_relv_blk = $ct_ebs -1;
        }

                # Compute the degeneracy. #

                # Preliminary. #

        my @set_div_rngs = my @set_cts_r_blks = ();
        my @set_blk2ct_r_blks = (); # element = {{index in @{$ebs}} => #{blocks on the right}, …} for the blocks that can occupy the relevant divided range.
        for (my $k=0; $k<$ct_ebs; $k++) {
            my $indx = $wrk_indices_div_rgs→[$k];
            my $div_rng = $sets_divided_ranges_lbds[$k]→[$indx];

            $set_div_rngs[$k] = $div_rng;
            $set_cts_r_blks[$k] = $sets_cts_r_blks[$k]→[$indx];

             my $set_regions_lbds = $sets_regions_lbds[$div_rng]→[2];
             my %blk2ct_r_blks;
             for (my $k2=0; $k2<$ct_ebs; $k2++) {
                my $info_region_lbd = $set_regions_lbds→[$k2];
                if (@{$info_region_lbd}==0) { next; }
                $blk2ct_r_blks{$k2} = $info_region_lbd->[2];
             }
              $set_blk2ct_r_blks[$k] = \%blk2ct_r_blks;
        }

                # Sort the indices of @set_div_rngs so that the divided ranges will be arranged from right to left. #
        my @std_indices = sort {$set_div_rngs[$b] <=> $set_div_rngs[$a] }  (0 .. $ct_ebs-1) ;

        my %already; # = ({index of block (in @{$ebs})} => {index of divided range in @sets_regions_lbds}, …) for the blocks already incorporated. #

                # Enumerate the possible combinations of {block}s vs {divided range}s. #

        my @set_possible_combinations = (\%already);
        foreach my $indx_blk0 (@std_indices) {

            my $div_rng = $set_div_rngs[$indx_blk0];
            my $ct_r_blks = $set_cts_r_blks[$indx_blk0];
            my $blk2ct_r_blks = $set_blk2ct_r_blks[$indx_blk0];
```

```
            my @new_set_poss_combs = ();
            while (my $already =  shift @set_possible_combinations) {

                        # RESTARTED on Dec 10, 2018. #

                foreach my $blk (keys %{$blk2ct_r_blks}) {
                    if (defined $already->{$blk}) { next; } # If the block is already incorporated, the combination canNOT be completed. #
                    my $ct_r_blks2 = $blk2ct_r_blks->{$blk};

                    if ($ct_r_blks2 > 1) {  # Actually, this block needs be refined (and moved to after the while-loop), because this version canNOT handle situations where two or more blocks are in the same divided range (see some cases in Figure SSSSA12). #
                        my $ct=0;
                        foreach my $blk2 (keys %{$already}) { if ($blk2 < $blk) { $ct++; } }
                        if ($ct_r_blks2 > $ct) { next; }
                    }
                    my %copy = %{$already};
                    $copy{$blk} = $div_rng;
                    push @new_set_poss_combs, \%copy;
                }
            }

                # Update the set of possible combinations of {block}s vs {divided range}s. #
            @set_possible_combinations = @new_set_poss_combs;
        }

                # Actually, the refined condition (within the foreach-loop above) should be placed here. #

                        # END of "RESTARTED on Dec 10, 2018." #

        my $degeneracy = scalar (@set_possible_combinations);

                # push @{$sets_divided_ranges_lbds[$k]}, $i;
                # push @{$sets_cts_r_blks[$k]}, $info_region_lbd->[2];


                # Store the computed degeneracy. #

        my $rf_degeneracy = \@degeneracies;
        for (my $k=0; $k <$ct_ebs; $k++) { $rf_degeneracy = $rf_degeneracy->[$wrk_indices_div_rgs->[$k]]; }
        ${$rf_degeneracy} = $degeneracy;

                # Move to the "next" relevant sub-divided range. #
        $wrk_indices_div_rgs->[$indx_relv_blk]++;
}

        # @sets_regions_lbds = ([0, $CT_CLMS-1, []]);
        # The array stores regions defined by the left-bounds of the gap-blocks.
        # elements = [$leftmost_clm, $rightmost_clm, \@set_regions_lbds], where
        # $left/rightmost_clm is the number (i.e., index in the local alignment)
        # of the left/right-most column in the region in question;
        # @{$set_regions_lbds[$i]} = ($left_bound, $right_bound, $ct_r_blks)
        # for the left-boundary of the #($eb_lm + $i)
        # $ebs->[$i] th block
```

(= empty if the block in question is not accommodated).

(b) When some non-swappable vertically equivalent blocks are in between swappable blocks... # ADDED on Dec 7, 2018. #

As in (1)-(ii), NOTE that the positions of the ranges of swappable blocks will depend on the positions of the non-swappable blocks
(as in Fig SSSSA13 of "figure_sppl_3_bp1_ANEX.xxxx.odp").

==> WILL be *seriously* considered later, i.e., when it becomes necessary...

APPENDIX F: Constructing @inter_block_relations (added on Dec 11, 2018), @interfering_blocks, @interfering_blocksets, and @cmpl_interfering_blocksets, which will help identify change of of alignment topology.

RESTARTED on Dec 11, 2018.

Assume that we have the following list:
@info_gblocks, where
($branch, $up_or_down) = @{$info_gblocks[$bl]}[$indx_br, $indx_u_or_d], where
$branch is the ID of the branch separating the gap-block, and
$u_or_d = 'U'/'L' if the gap-block is on the 'upper/'lower'-side of $branch.
#
Also assume that we already have:
%branch2up_down2equiv_blocks = ($branch => {$up_or_down => \@equiv_blocks, …}, ..),
where $up_or_down = 'U'/'L' depending on whether the gap-block is on the "upper"/"lower"-side of the branch,
and @equiv_blocks lists the indices of vertically equivalent blocks (in ascending order).
(This should be easily constructed from the raw list of gap-blocks, as long as it stores the branches and 'U' or 'L' statuses of the gap-blocks.)

(1) First, construct @inter_block_relations

 # $inter_block_relations[$bl1]→[$bl2] = $relation,
for the relation between the $bl1 th and $bl2 th blocks.
#
Here, $relation can be:
#
'NIF' (for 'non-interfering'),
'S' (for "effective sibling" 'sibling/parent/child' of the $bl th block),
'Cp' (for 'complementary'),
'ONN' (for 'overlapping yet non-nesting' NOR 'complementary-sibling/parent/child'),
'ONCS' (for 'overlapping yet non-nesting' but 'complementary-sibling/parent/child' of the $bl th block),
'>' (for the $bl2 th block being 'vertically included' in the $bl1 th block),
'>(ch)' (for the $bl2 th block being 'vertically included' in, and an **"effective child"** of, the $bl1 th block), # ADDED on Jan 16, 2019.
'<' (for the $bl2 th block 'vertically including' the $bl1 th block),
'<(pa)' (for the $bl2 th block 'vertically including', and being an **"effective parent"** of, the $bl1 th block), # ADDED on Jan 16, 2019.
'=' (for 'vertically identical').
#
#NOTE1: Actually, the 'sibling' and 'complementary-sibling' here include the 'parent-child'
and 'complementary-parent-child' relationships, respectively.
#

```
my @inter_block_relations = ();
for (my $bl1=0; $bl1<$B; $bl1++) { # Outer for-loop (over $bl1).

    my ($br1, $u_or_d1) = @{$info_gblocks[$bl1]}[$indx_br, $indx_u_or_d];
    my $eq_br1 = $equiv_br->{$br1};
    my $pa_br1 = $node2pa->{$br1};
    # my $eq_pa_br1 = $equiv_br->{$pa_br1};
    # my $children1 = $node2ch->{$br1};
    # my $depth1 = $node2depth->{$br1};
    my $sibs1 = $node2ch->{$pa_br1}; # ADDED on Jan 16, 2019.

    for (my $bl2=0; $bl2<$B; $bl2++) {  # Middle for-loop (over $bl2).
        if ($bl2 == $bl1) {
                $relations_w_bl1[$bl2] = undef;
                next;
        }
        my ($br2, $u_or_d2) = @{$info_gblocks[$bl2]}[$indx_br, $indx_u_or_d];
        my $eq_br2 = $equiv_br->{$br2};
        my $pa_br2 = $node2pa->{$br2};
        # my $pa_br2 = $br2pa->{$br2};

        # my $children2 = $br2ch->{$br2};
        # my $depth2 = $node2depth->{$br2};
        my $sibs2 = $node2ch->{$pa_br2}; # ADDED on Jan 16, 2019.

                # (1) (vertically) equivalent or complementary. #

        if ($br1 == $br2) {
                $relations_w_bl1[$bl2] = ($u_or_d1 eq $u_or_d2) ? '=' : 'Cp' ;
                next;

        } elsif ((defined $eq_br2) and ($br1 == $eq_br2)) {
                $relations_w_bl1[$bl2] = ($u_or_d1 eq $u_or_d2) ? 'Cp' : '=' ;
                next;
        }

                # (2) $br2 is the parent of $br1, or its equivalent. #

        if ($br2 == $pa_br1) {

            my $rel;

            if ($u_or_d1 eq 'U') {
                if ($u_or_d2 eq 'U') {
                    if (@{$sibs1} == 2) { # ADDED on Jan 16, 2019.
                        $rel = '>(ch)'; # $bl2 is an "effective child" of $bl1.
                    } else {
                        $rel = '>'; # $bl2 is included in $bl1.
                    }
                } else { # if ($u_or_d2 eq 'L')
                    if (@{$sibs1} == 2) { # ADDED on Jan 16, 2019.
                        $rel = 'ONCS'; # Overlapping but non-nesting and complementary-parent-child.
                    } else {
                        $rel = 'ONN'; # Overlapping but non-nesting NOR complementary-parent-child.
                    }
                }
            } else { # if ($u_or_d1 eq 'L')
                if ($u_or_d2 eq 'U') {
```

```
                if (@{$sibs1} == 2) { # ADDED on Jan 16, 2019.
                  $rel = 'S'; # Parent-child.
                } else {
                  $rel = 'NIF'; # $bl1 and $bl2 are effectively "non-interfering".
                }
              } else { # if ($u_or_d2 eq 'L')
                if (@{$sibs1} == 2) { # ADDED on Jan 16, 2019.
                  $rel = '<(pa)'; # $bl2 is an "effective parent" of $bl1.
                } else {
                  $rel = '<'; # $bl2 includes $bl1.
                }
              }
          }

          $relations_w_bl1[$bl2] = $rel;
           next;

      } elsif ((defined $eq_br2) and ($eq_br2 == $pa_br1)) {

          my $rel;

          if ($u_or_d1 eq 'U') {
              if ($u_or_d2 eq 'U') {
                if (@{$sibs1} == 2) { # ADDED on Jan 16, 2019.
                  $rel = 'ONCS'; # Overlapping but non-nesting and complementary-parent-child.
                } else {
                  $rel = 'ONN'; # Overlapping but non-nesting NOR complementary-parent-child.
                }
              } else { # if ($u_or_d2 eq 'L')
                if (@{$sibs1} == 2) { # ADDED on Jan 16, 2019.
                  $rel = '>(ch)'; # $bl2 is an "effective child" of $bl1.
                } else {
                  $rel = '>'; # $bl2 is included in $bl1.
                }
              }
          } else { # if ($u_or_d1 eq 'L')
              if ($u_or_d2 eq 'U') {
                if (@{$sibs1} == 2) { # ADDED on Jan 16, 2019.
                  $rel = '<(pa)'; # $bl2 is an "effective parent" of $bl1.
                } else {
                  $rel = '<'; # $bl2 includes $bl1.
                }
              } else { # if ($u_or_d2 eq 'L')
                if (@{$sibs1} == 2) { # ADDED on Jan 16, 2019.
                  $rel = 'S'; # Parent-child.
                } else {
                  $rel = 'NIF'; # $bl1 and $bl2 are effectively "non-interfering".
                }
          }

          $relations_w_bl1[$bl2] = $rel;
           next;
      }

              # (3) $br1 is the parent of $br2, or its equivalent. #

      if ($br1 == $pa_br2) {
```

```
            my $rel;

            if ($u_or_d1 eq 'U') {
                if ($u_or_d2 eq 'U') {
                  if (@{$sibs2} == 2) { # ADDED on Jan 16, 2019.
                    $rel = '<(pa)'; # $bl2 is an "effective parent" of $bl1.
                  } else {
                    $rel = '<'; # $bl2 includes $bl1.
                  }
                } else { # if ($u_or_d2 eq 'L')
                  if (@{$sibs2} == 2) { # ADDED on Jan 16, 2019.
                    $rel = 'S'; # Parent-child.
                  } else {
                    $rel = 'NIF'; # $bl1 and $bl2 are effectively "non-interfering".
                  }
                }
            } else { # if ($u_or_d1 eq 'L')
                if ($u_or_d2 eq 'U') {
                  if (@{$sibs2} == 2) { # ADDED on Jan 16, 2019.
                    $rel = 'ONCS'; # Overlapping but non-nesting and complementary-parent-child.
                  } else {
                    $rel = 'ONN'; # Overlapping but non-nesting NOR complementary-parent-child.
                  }
                } else { # if ($u_or_d2 eq 'L')
                  if (@{$sibs2} == 2) { # ADDED on Jan 16, 2019.
                    $rel = '>(ch)'; # $bl2 is an "effective child" of $bl1.
                  } else {
                    $rel = '>'; # $bl2 is included in $bl1.
                  }
                }
            }

            $relations_w_bl1[$bl2] = $rel;
             next;

        } elsif ((defined $eq_br1) and ($eq_br1 == $pa_br2)) {

            my $rel;

            if ($u_or_d1 eq 'U') {
                if ($u_or_d2 eq 'U') {
                  if (@{$sibs2} == 2) { # ADDED on Jan 16, 2019.
                    $rel = 'ONCS'; # Overlapping but non-nesting and complementary-parent-child.
                  } else {
                    $rel = 'ONN'; # Overlapping but non-nesting NOR complementary-parent-child.
                  }
                } else { # if ($u_or_d2 eq 'L')
                  if (@{$sibs2} == 2) { # ADDED on Jan 16, 2019.
                    $rel = '>(ch)'; # $bl2 is an "effective child" of $bl1.
                  } else {
                    $rel = '>'; # $bl2 is included in $bl1.
                  }
                }
            } else { # if ($u_or_d1 eq 'L')
                if ($u_or_d2 eq 'U') {
                  if (@{$sibs2} == 2) { # ADDED on Jan 16, 2019.
                    $rel = '<(pa)'; # $bl2 is an "effective parent" of $bl1.
                  } else {
```

```
                    $rel = '<'; # $bl2 includes $bl1.
                  }
                } else { # if ($u_or_d2 eq 'L')
                  if (@{$sibs2} == 2) { # ADDED on Jan 16, 2019.
                    $rel = 'S'; # Parent-child.
                  } else {
                    $rel = 'NIF'; # $bl1 and $bl2 are effectively "non-interfering".
                  }
                }
            }

            $relations_w_bl1[$bl2] = $rel;
             next;
        }

                # (4) $br1 is the sibling of $br2. #

        if ($pa_br1 == $pa_br2) {

            my $rel;

          if (($pa_br1 == $top_node) and (@{$sibs1} == 2)) { # ADDED on Jan 16, 2019. #
                        # In this case, $br1 and $br2 are actually equivalent to each other. #
                $relations_w_bl1[$bl2] = ($u_or_d1 eq $u_or_d2) ? 'Cp' : '=' ;
                next;

          } elsif ( (($pa_br1 == $top_node) and (@{$sibs1} == 3))
                or (($pa_br1 != $top_node) and (@{$sibs1} == 2)) ) { # ADDED on Jan 16, 2019. #
                        # This is the normal case. #

            if ($u_or_d1 eq 'U') {
                if ($u_or_d2 eq 'U') {
                    $rel = 'ONCS'; # Overlapping but non-nesting and complementary-sibling.
                } else { # if ($u_or_d2 eq 'L')
#                   $rel = '>'; # $bl2 is included in $bl1.
                    $rel = '>(ch)'; # $bl2 is an "effective child" of $bl1. # MODIFIED on Jan 16, 2019.
                }
            } else { # if ($u_or_d1 eq 'L')
                if ($u_or_d2 eq 'U') {
#                   $rel = '<'; # $bl2 includes $bl1.
                    $rel = '<(pa)'; # $bl2 is an "effective parent" of $bl1.
                } else { # if ($u_or_d2 eq 'L')
                    $rel = 'S'; # Siblings.
                }
            }

          } else { # $br1 and $br2 have more siblings than usual. # ADDED on Jan 16, 2019.

            if ($u_or_d1 eq 'U') {
                if ($u_or_d2 eq 'U') {
#                   $rel = 'ONCS'; # Overlapping but non-nesting and complementary-sibling.
                    $rel = 'ONN'; # Overlapping but non-nesting NOR complementary-parent-child. # MODIFIED on Jan 16, 2019.
                } else { # if ($u_or_d2 eq 'L')
                    $rel = '>'; # $bl2 is included in $bl1.
                }
            } else { # if ($u_or_d1 eq 'L')
```

```
                if ($u_or_d2 eq 'U') {
                    $rel = '<'; # $bl2 includes $bl1.
                } else { # if ($u_or_d2 eq 'L')
#                   $rel = 'S'; # Siblings.
                    $rel = 'NIF'; # $bl1 and $bl2 are effectively "non-interfering". # MODIFIED on Jan 16, 2019.
                }
            }

        } # END of "if (...) {...} elsif (...) {...} else {...} # ADDED on Jan 16, 2019."

            $relations_w_bl1[$bl2] = $rel;
             next;
        }

            # (4)' Parent of $br1 is equivalent to parent of $br2. # Actually, $br1 and $br2 are as unrelated as between uncle and nephew.

            # (5) Other cases. #

            # RESTARTED on Dec 13, 2018. #

        my ($common_anc, $div1, $div2) = fetch_common_ancestors ($br1, $br2, %{$node2pa});

        if (@{$div1}==0) { # $br1 is an ancestor of $br2.

            my $rel;

            if ($u_or_d1 eq 'U') {
                if ($u_or_d2 eq 'U') {
                    $rel = '<'; # $bl2 includes $bl1.
                } else { # if ($u_or_d2 eq 'L')
                    $rel = 'NIF'; # Non-interfering.
                }
            } else { # if ($u_or_d1 eq 'L')
                if ($u_or_d2 eq 'U') {
                    $rel = 'ONN'; # Overlapping but non-nesting NOR complementary-sibling/parent/child.
                } else { # if ($u_or_d2 eq 'L')
                    $rel = '>'; # $bl2 is included in $bl1.
                }
            }

            $relations_w_bl1[$bl2] = $rel;
             next;

        } elsif (@{$div2} == 0) { # $br2 is an ancestor of $br1.

            my $rel;

            if ($u_or_d1 eq 'U') {
                if ($u_or_d2 eq 'U') {
                    $rel = '>'; # $bl2 is included in $bl1.
                } else { # if ($u_or_d2 eq 'L')
                    $rel = 'ONN'; # Overlapping but non-nesting NOR complementary-sibling/parent-child.
                }
            } else { # if ($u_or_d1 eq 'L')
```

```
                    if ($u_or_d2 eq 'U') {
                        $rel = 'NIF'; # Non-interfering.
                    } else { # if ($u_or_d2 eq 'L')
                        $rel = '<'; # $bl2 includes $bl1.
                    }
                }

                $relations_w_bl1[$bl2] = $rel;
                 next;

            } else {  # $br1 and $br2 diverge from each other at some node (possibly the root) in the tree.
                my $rel;

                if ($u_or_d1 eq 'U') {
                    if ($u_or_d2 eq 'U') {
                        $rel = 'ONN'; # Overlapping but non-nesting NOR complementary-sibling/child/parent.
                    } else { # if ($u_or_d2 eq 'L')
                        $rel = '>'; # $bl2 is included in $bl1.
                    }
                } else { # if ($u_or_d1 eq 'L')
                    if ($u_or_d2 eq 'U') {
                        $rel = '<'; # $bl2 includes $bl1.
                    } else { # if ($u_or_d2 eq 'L')
                        $rel = 'NIF'; # Non-interfering.
                    }
                }

                $relations_w_bl1[$bl2] = $rel;
                 next;
            }

        } # END of the middle for-loop (over $bl2).

            $inter_block_relations[$bl1] = \@relations_w_bl1;

}# END of the outer for-loop (over $bl1).
```

(2) Second, construct @interfering_blocks:

```
# @interfering_blocks, where
# @{$interfering_blocks[$bl]} = (\@blocks, \@relations) stores information on blocks that can
# interfere with the $bl th block (or empty if it has no such blocks), where
# $blocks[$k] is the index (or rank) of the $k th interfering block,
# $relations[$k] is the relation of the $k th interfering block with the $bl th block.
#
#       Here, we will only consider the following relations:
#       'S' (for 'sibling/parent/child' of the $bl th block),
#       'Cp' (for 'complementary'),
#       '>(ch)' (for the $bl2 th block being 'vertically included' in, and an "effective child" of, the $bl1 th block), # MODIFIED on Jan 16, 2019.
#       '<(pa)' (for the $bl2 th block 'vertically including', and being an "effective parent" of, the $bl1 th block), # MODIFIED on Jan 16, 2019.
#       # '>' (for the $bl2 th block being 'vertically included' in the $bl1 th block),
#       # '<' (for the $bl2 th block 'vertically including' the $bl1 th block),
```

```
#       '=' (for 'vertically identical').
#
        # NOTES:
        # (1) Blocks with 'ONN' relations cause NO topological changes, as far as we employ the current coordinate system;
        # (2) Also, blocks with 'ONCS' relations cause NO topological changes, as far as we restrict ourselves to parsimonious indel histories (more precisely, parsimonious ancestral gap states);
        # (3) Also, as far as we restrict ourselves to parsimonious indel histories (more precisely, parsimonious ancestral gap states), the separating branches of blocks with the '>' or '<' relations with the $bl th block must be either the child or parent (or a sibling if they are children of a trivalent root) of the separating branch of the $bl th block.  (As of Jan 16, 2019, this condition trivially holds.)

my @interfering_blocks = ();
for (my $bl=0; $bl <$B; $bl++) {  # Outer for-loop (over $bl).

            # ADDED on Dec 14, 2018.
        my ($br1, $u_or_d1) = @{$info_gblocks[$bl]}[$indx_br, $indx_u_or_d];
        my $pa1 = $node2pa->{$br1};
        my $eq_br1 = $eq_br->{$br1};
            # END of "ADDED on Dec 14, 2018."

        my $rel_w_bl = $inter_block_relations[$bl];

        my @blocks = my @relations = ();

    for (my $bl2=0; $bl2<$B; $bl2++) { # Inner for-loop (over $bl2).
        if ($bl2 == $bl) { next; }

        my $rel = $rel_w_bl->[$bl2];

        # CORRECTED on Dec 14, 2018. #
#       if ( ($rel eq 'S') or ($rel eq 'Cp')
#               or ($rel eq '>') or ($rel eq '<') or ($rel eq '=')
#       ) {
        if ( ($rel eq 'S') or ($rel eq 'Cp') or ($rel eq '=')) {

            push @blocks, $bl2;
            push @relations, $rel;

        } elsif (($rel eq '>(ch)') or ($rel eq '<(pa)')) {  # REVISED on Jan 16, 2019.

            # OBSOLETE as of Jan 16, 2019. #
#       } elsif (($rel eq '>') or ($rel eq '<')) {
#
#           my ($br2, $u_or_d2) = @{$info_gblocks[$bl2]}[$indx_br, $indx_u_or_d];
#           my $pa2 = $node2pa->{$br2};
#           my $eq_br2 = $eq_br->{$br2};
#           if ( ($br1 == $pa2) or ((defined $eq_br1) and ($eq_br1 == $pa2))  # $br1 is the parent of $br2.
#               or ($br2 == $pa1) or ((defined $eq_br2) and ($eq_br2 == $pa1)) # $br2 is the parent of $br1.
#               or ($pa1 == $pa2)  # $br1 and $br2 are siblings.
#           ) {
            # END of "OBSOLETE as of Jan 16, 2019." #

            push @blocks, $bl2;
```

```
                push @relations, $rel;
            }
        }
        # END of "CORRECTED on Dec 14, 2018." #

    }   # END of the inner for-loop (over $bl2).

        $interfering_blocks[$bl] = (@blocks>0) ? [\@blocks, \@relations] : [ ] ;  # CORRECTED on 2019/03/08.
        ## $interfering_blocks[$bl] = [\@blocks, \@relations];

}  # END of the outer for-loop (over $bl).
```

(3) Third, construct @interfering_blocksets:

@interfering_blocksets, where
@{$interfering_blocksets[$bl]} = (\@blocksets, \@relations) are nearly the same as above, except that **@{$blocksets[$k]}** is now a set of indices (or ranks) of the $k th interfering block set. (NOTE: Each @{$blockset[$k]} contains only blocks #with the same horizontal size (OBSOLETE as of 2019/01/22)#
that form a (complementary) monophyletic group.)

For the moment, we will only consider the following relations:

'S' (for 'sibling/parent/child' of the $bl th block),
'>(ch)' (for the block-set being 'vertically included' in, and an **"effective child"** of, the $bl1 th block), # MODIFIED on Jan 16, 2019.
#'>' (for the blockset being 'vertically included' in the $bl1 th block).

NOTES:
(5) As in note (3), the separating branch of the blockset with the **'>'** relation must be either the **child or parent or sibling** of the separating branch of the $bl th block;
(6) Each blockset with the '**Cp**' relation likely involves a single block with the 'S' relation;
(7) Likewise, **'='** likely involves a single block with the 'S' relation;
(8) And **'<'** likely involves a single block with the '=', '>', and/or the 'S' relations.

(i) First, construct block-sets:

%branch2up_down2blocksets = ($branch => {$up_or_down => \@blocksets, …}, …),
where @blocksets stores block-sets that are separated by, and on the $up_or_down side of, $branch.

```
# Assuming that we already have:
# %branch2up_down2equiv_blocks = ($branch => {$up_or_down => \@equiv_blocks, …}, ..),
# where $up_or_down = 'U'/'L' depending on whether the gap-block is on the "upper"/"lower"-side of the branch,
# and @equiv_blocks lists the indices of vertically equivalent blocks (in ascending order).
# Also, assuming that we also have:
# @block_sizes, with $block_sizes[$bl] is the size of the $bl th block.

        # (o) Initialization. #

my %pre_br2up_down2blocksets; # CORRECTED on 2019/03/09.
#my %pre_br2up_down2equiv_blocks;
foreach my $br (keys %branch2up_down2equiv_blocks) {
```

```
        my $up_down2equiv_blocks = $branch2up_down2equiv_blocks;
        unless (defined $up_down2equiv_blocks) { next; }
        my %up_down2blocksets = ();
        foreach my $u_or_d (keys %{$up_down2equiv_blocks}) {
            my $equiv_blocks = $up_down2equiv_blocks->{$u_or_d};
            unless (defined $equiv_blocks) { next; }
            my @blocksets = ();
            foreach my $bl (@{$equiv_blocks}) { push @blocksets, [$bl]; }
            $up_down2blocksets{$u_or_d} = \@blocksets;
        }
        $pre_br2up_down2blocksets{$br} = \%up_down2blocksets; # CORRECTED on
2019/03/09.
        #$pre_br2up_down2equiv_blocks{$br} = \%up_down2blocksets;
}

        # (a) The bottom-up traversal. #

foreach my $br (sort {$node2depth{$b} <=> $node2depth{$a}} keys %br_attr) { # Outer foreach-
loop (over branches in descending order of their depths).

     my $children = $node2ch->{$br};

    unless (defined $children) { # When $br is external.

#       my $u_or_d2eq_blks = $branch2up_down2equiv_blocks{$br};
#       unless (defined $u_or_d2eq_blks) { next; }
#       my $d_eq_blks = $u_or_d2eq_blks->{'L'};
#       unless (defined $d_eq_blks) { next; }
#       my @blocksets = ();
#       foreach my $blk (@{$d_eq_blks}) { push @blocksets, [$blk]; }
#       $branch2up_down2blocksets{$br} = {'L' => \@blocksets };
        next;
    }

        # When $br is internal. #

    my $ct_children = @{$children};

                # MODIFIED on Dec 14, 2018. #
#     my $ch1 = $children->[0];
#     my $u_or_d2blksets_ch1 = $pre_br2up_down2blocksets{$ch1};
#     unless (defined $u_or_d2blksets_ch1) { next; }
#     my $d_blksets_ch1 = $u_or_d2blksets_ch1->{'L'};
#     unless (defined $d_blksets_ch1) { next; }
#     my @blocksets = ();
#      foreach my $bs (@{$d_blksets_ch1}) { my @copy = @{$bs}; push @blocksets, \@copy; }

    my @sets_d_blksets_ch = ();

    for (my $i= 0; $i<$ct_children; $i++) {  # Middle for-loop (over children).
#   for (my $i=1; $i<$ct_children; $i++) {  # Middle for-loop (over children).

        my $ch = $children->[$i];
        my $u_or_d2blksets_ch = $pre_br2up_down2blocksets{$ch};
        unless (defined $u_or_d2blksets_ch)
            @sets_d_blksets_ch = ();
            # @blocksets = ();
            last;
```

```
        }
        my $d_blksets_ch = $u_or_d2blksets_ch→{'L'};
        unless (defined $d_blksets_ch) {
            @sets_d_blksets_ch = ();
            # @blocksets = ();
            last;
        }

        push @sets_d_blksets_ch, $d_blksets_ch;

    }  # END of the middle for-loop (over children).

    if (@sets_d_blksets_ch == 0) { next; }

    my @blocksets = ([]);

    foreach my $d_blksets_ch (@sets_d_blksets_ch) { # Middle foreach-loop (over block-sets on children).
        my @new_blocksets = ();
        foreach my $old_blkset (@blocksets) { # Inner foreach-loop (over old block-sets).

            # my $block_size1 = (@{$old_blkset}>0) ? $block_sizes[$old_blkset→[0]] : undef; # OBSOLETE as of 2019/01/22.

            foreach my $add_blkset (@{$d_blksets_ch}) { # Innermost foreach-loop (over block-sets to be added).
                # my $block_size2 = $block_sizes[$add_blkset→[0]];  # OBSOLETE as of 2019/01/22.
                # if ((@{$old_blkset} == 0) or ($block_size1 == $block_size2)) {  # OBSOLETE as of 2019/01/22.
                    my @new_blkset = (@{$old_blkset}, @{$add_blkset});
                    push @new_blocksets, \@new_blkset;
                #}  # OBSOLETE as of 2019/01/22.
            }  # Innermost foreach-loop (over block-sets to be added).
        }   # Inner foreach-loop (over old block-sets).

        if (@new_blocksets >0) {
            @blocksets = @new_blocksets; # Update the @blocksets. #

        } else {
            @blocksets = ();
            last;
        }

    }  # END of the middle foreach-loop (over block-sets on children).

    if (@blocksets>0) {

        my $pre_up_down2blocksets = $pre_br2up_down2blocksets{$br}; # CORRECTED on 2019/03/09.
        #my $pre_up_down2blocksets{$br} = $pre_br2up_down2blocksets{$br};
        unless (defined $pre_up_down2blocksets) {
            $pre_up_down2blocksets = $pre_br2up_down2blocksets{$br} = {};
        } # ADDED on 2019/03/09.
        my $pre_d_blocksets = $pre_up_down2blocksets→{'L'};
        unless (defined $pre_d_blocksets) { $pre_d_blocksets = $pre_up_down2blocksets→{'L'} = []; }
        push @{$pre_d_blocksets}, @blocksets;
```

```
    }

} # Outer foreach-loop (over branches in descending order of their depths).


        # (c) The top-down traversal. #

foreach my $br (sort {$node2depth{$a} <=> $node2depth{$b}} keys %br_attr) { # Outer foreach-loop (over branches in ascending order of their depths).
        my $pa = $node2pa->{$pa};

        my @sets_prepre_blocksets = ();

        if (defined $br_attr{$pa}) { # The parent node represents a branch as well.
            my $up_down2blocksets = $pre_br2up_down2blocksets{$pa};
            unless (defined $up_down2blocksets) { next; }

            my $pre_u_blocksets_pa = $up_down2blocksets->{'U'};
            unless (defined $pre_u_blocksets_pa) { next; }

            push @sets_prepre_blocksets, $pre_u_blocksets_pa;
        }

        my $sibs = $node2ch->{$pa};
        #my $ct_sibs = @{$sibs}-1;

        foreach my $sib (@{$sibs}) { # 1st middle foreach-loop (over siblings).

            if ($sib == $br) { next; }
            my $up_down2blocksets = $pre_br2up_down2blocksets{$sib};
            unless (defined $up_down2blocksets) {
                @sets_prepre_blocksets = 0;
                last;
            }
            my $pre_d_blocksets_sib = $up_down2blocksets->{'L'};
            unless (defined $pre_d_blocksets_sib) {
                @sets_prepre_blocksets = ();
                last;
            }
            push @sets_prepre_blocksets, $pre_d_blocksets_sib;

        } # END of the 1st middle foreach-loop (over siblings).

        if (@sets_prepre_blocksets == 0) { next; }


        my @blocksets = ([]);

    foreach my $prepre_blksets (@sets_prepre_blocksets) { # 2nd middle foreach-loop (over block-sets on siblings and parent (if at all)).

        my @new_blocksets = ();
        foreach my $old_blkset (@blocksets) { # Inner foreach-loop (over old block-sets).

            # my $block_size1 = (@{$old_blkset}>0) ? $block_sizes[$old_blkset->[0]] : undef; # OBSOLETE as of 2019/01/22.
```

```
            foreach my $add_blkset (@{$prepre_blksets}) { # Innermost foreach-loop (over block-sets to be added).

                # my $block_size2 = $block_sizes[$add_blkset→[0]]; # OBSOLETE as of 2019/01/22.
                # if ((@{$old_blkset}==0) or ($block_size1 == $block_size2)) { # OBSOLETE as of 2019/01/22.
                    my @new_blkset = (@{$old_blkset}, @{$add_blkset});
                    push @new_blocksets, \@new_blkset;
                # } # OBSOLETE as of 2019/01/22.

            } # END of the innermost foreach-loop (over block-sets to be added).

        } # END of the inner foreach-loop (over old block-sets).

        if (@new_blocksets>0) {
            @blocksets = @new_blocksets; # Update @blockstes.
        } else {
            @blocksets = ();
            last;
        }

    } # END of the 2nd middle foreach-loop (over block-sets on siblings and parent (if at all)).

    if (@blocksets>0) {

        my $pre_up_down2blocksets = $pre_br2up_down2blocksets{$br}; # CORRECTED on 2019/03/09.
        # my $pre_up_down2blocksets{$br} = $pre_br2up_down2blocksets{$br};
        unless (defined $pre_up_down2blocksets) {
            $pre_up_down2blocksets = $pre_br2up_down2blocksets{$br} = {};
        } # ADDED on 2019/03/09.
        my $pre_u_blocksets = $pre_up_down2blocksets→{'U'};
        unless (defined $pre_u_blocksets) { $pre_u_blocksets = $pre_up_down2blocksets→{'U'} = []; }
        push @{$pre_u_blocksets}, @blocksets;
    }

} # Outer foreach-loop (over branches in ascending order of their depths).


            # RESTARTED on Dec 14, 2018. #

        # (d) Cleaning-up the output, by removing trivial block-sets (each consisting of a single block). #

my %branch2up_down2blocksets;

foreach my $br (keys %pre_br2up_down2blocksets) {  # Outer foreach-loop (over branches).

    my $up_down2pre_blksets = $pre_br2up_down2blocksets{$br};
    my %up_down2blocksets;

    foreach my $u_or_d (keys %{$up_down2pre_blksets}) { # Middle foreach-loop (over 'U' and 'L', if relevant).
        my $pre_blksets = $up_down2pre_blksets→{$u_or_d};
        my @blocksets = ();
        foreach my $blkset (@{$pre_blksets}) { # Inner foreach-loop (over block-sets).
```

```
            if (@{$blkset}>1) { push @blocksets, $blkset; }
        }   # END of the inner foreach-loop (over block-sets).

        if (@blocksets>0) { $up_down2blocksets{$u_or_d} = \@blocksets; }

    } # Middle foreach-loop (over 'U' and 'L', if relevant).

    if (0 < scalar keys %up_down2blocksets) { $branch2up_down2blocksets{$br} = \
%up_down2blocksets; }

}  # END of the outer foreach-loop (over branches).
```

(ii) Construct @interfering_blocksets using %branch2up_down2blocksets.

@{$interfering_blocksets[$bl]} = (\@blocksets, \@relations) are nearly the same as above, except that **@{$blocksets[$k]}** is now a set of indices (or ranks) of the $k th interfering block set. (NOTE: Each @{$blockset[$k]} contains only blocks with the same horizontal size that form a (complementary) monophyletic group.)

For the moment, we will only consider the following relations:

'S' (for 'sibling/parent/child' of the $bl th block),
'>(ch)' (for the blockset being 'vertically included' in, and an "effective child" of, the $bl1 th block). # Modified on Jan 18, 2019.
'>' (for the blockset being 'vertically included' in the $bl1 th block).

NOTES:
(5) As in note (3), the separating branch of the blockset with the **'>'** relation must be either the **child or parent or sibling** of the separating branch of the $bl th block;
(6) Each blockset with the '**Cp**' relation likely involves a single block with the 'S' relation;
(7) Likewise, **'='** likely involves a single block with the 'S' relation;
(8) And **'<'** likely involves a single block with the '=', '>', and/or the 'S' relations.

```
my @interfering_blocksets = ();

for (my $bl1=0; $bl1<$B; $bl1++) { # Outer for-loop (over $bl1).

    my ($br1, $u_or_d1) = @{$info_gblocks[$bl1]}[$indx_br, $indx_u_or_d];
    my $eq_br1 = $equiv_br->{$br1};
    my $pa1 = $node2pa→{$br1};
    my $children1 = $node2ch→{$br1};
    my $eq_children1 = (defined $eq_br1) ? $node2ch→{$eq_br1} : [ ];
    my $sibs1 = $node2ch→{$pa1};

    my @blocksets = my @relations = ();

        # (a) Examine the parent of $br1. #

    my $up_down2blocksets_pa = $branch2up_down2blocksets{$pa1};
    if (defined $up_down2blocksets_pa) {
        foreach my $u_or_d2 (keys %{$up_down2blocksets_pa}) { #1st inner foreach-loop
(over $u_or_d2).
            my $blsets_pa = $up_down2blocksets_pa→{$u_or_d2};
            my $rel;
            if ($u_or_d1 eq 'U') {
                if ($u_or_d2 eq 'U') {
                    if (@{$sibs1} == 2) { $rel = '>(ch)'; } # REVISED on Jan 16, 2019.
```

```
#                       $rel = '>';
                    } else { # if ($u_or_d2 eq 'L')
                        next; # $rel = 'NNCS'.
                    }
                } else { # if ($u_or_d1 eq 'L')
                    if ($u_or_d2 eq 'U') {
                        if (@{$sibs1} == 2) { $rel = 'S'; } # REVISED on Jan 16, 2019.
#                       $rel = 'S';
                    } else { # if ($u_or_d2 eq 'L')
                        next; # $rel = '<'.
                    }
                }
                if (defined $rel) {
                    foreach my $blset (@{$blsets_pa}) {
                        push @blocksets, $blset;
                        push @relations, $rel;
                    }
                }
            } # END of the 1st inner foreach-loop (over $u_or_d2).
        }

        # RESTARTED on Dec 15, 2018. #

                # (b) Examine the children of $br1. #

  if (@{$children1} == 2) { # ADDED on Jan 16, 2019.
    foreach my $ch (@{$children1}) { # 2nd middle foreach-loop (over children).
        my $up_down2blocksets_ch = $branch2up_down2blocksets{$ch};

        if (defined $up_down2blocksets_ch) {
            foreach my $u_or_d2 (keys %{$up_down2blocksets_ch}) { # 2nd inner foreach-loop
(over $u_or_d2).
                my $blsets_ch = $up_down2blocksets_ch->{$u_or_d2};
                my $rel;
                if ($u_or_d1 eq 'U') {
                    if ($u_or_d2 eq 'U') {
                        next; # $rel = '<'.
                    } else { # if ($u_or_d2 eq 'L')
                        $rel = 'S';
                    }
                } else { # if ($u_or_d1 eq 'L')
                    if ($u_or_d2 eq 'U') {
                        next; # $rel = 'NNCS'.
                    } else { # if ($u_or_d2 eq 'L')
                        $rel = '>(ch)'; # REVISED on Jan 16, 2019.
#                       $rel = '>';
                    }
                }
                if (defined $rel) {
                    foreach my $blset (@{$blsets_ch}) {
                        push @blocksets, $blset;
                        push @relations, $rel;
                    }
                }
            } # END of the 2nd inner foreach-loop (over $u_or_d2).
        }

    }  # END of the 2nd middle foreach-loop (over children).
```

```
  } # END of "if (@{$children1} == 2) { # ADDED on Jan 16, 2019. ...}"

                # (c) Examine the children of $eq_br1. #

  if (@{$eq_children1} == 2) { # ADDED on Jan 16, 2019.
   foreach my $eq_ch (@{$eq_children1}) { # 3rd middle foreach-loop (over children of $eq_br1).
        my $up_down2blocksets_eq_ch = $branch2up_down2blocksets{$eq_ch};

        if (defined $up_down2blocksets_eq_ch) {
            foreach my $u_or_d2 (keys %{$up_down2blocksets_eq_ch}) { # 3rd inner foreach-loop
(over $u_or_d2).
                my $blsets_eq_ch = $up_down2blocksets_eq_ch->{$u_or_d2};
                my $rel;
                if ($u_or_d1 eq 'U') {
                    if ($u_or_d2 eq 'U') {
                        next; # $rel = 'NNCS'.
                    } else { # if ($u_or_d2 eq 'L')
                        $rel = '>(ch)'; # REVISED on Jan 16, 2019.
#                       $rel = '>';
                    }
                } else { # if ($u_or_d1 eq 'L')
                    if ($u_or_d2 eq 'U') {
                        next; # $rel = '<'.
                    } else { # if ($u_or_d2 eq 'L')
                        $rel = 'S';
                    }
                }
                if (defined $rel) {
                    foreach my $blset (@{$blsets_eq_ch}) {
                        push @blocksets, $blset;
                        push @relations, $rel;
                    }
                }
            } # END of the 3rd inner foreach-loop (over $u_or_d2).
        }

    }  # END of the 3rd middle foreach-loop (over children of $eq_br1).
  } # END of "if (@{$eq_children1} == 2) { # ADDED on Jan 16, 2019. ...}"


                # (d) Examine the siblings of $br1. #

  if ( (($pa1 == $top_node) and (@{$sibs1} == 3)) or  (($pa1 != $top_node) and (@{$sibs1} == 2))
) { # ADDED on Jan 16, 2019.
    foreach my $sib (@{$sibs1}) { # 5th middle foreach-loop (over siblings).
        if ($sib == $br1) { next; }
        my $up_down2blocksets_sib = $branch2up_down2blocksets{$sib};

        if (defined $up_down2blocksets_sib) {
            foreach my $u_or_d2 (keys %{$up_down2blocksets_sib}) { # 5th inner foreach-loop
(over $u_or_d2).
                my $blsets_sib = $up_down2blocksets_sib->{$u_or_d2};
                my $rel;
                if ($u_or_d1 eq 'U') {
                    if ($u_or_d2 eq 'U') {
                        next; # $rel = 'NNCS'.
                    } else { # if ($u_or_d2 eq 'L')
                        $rel = '>(ch)'; # REVISED on Jan 16, 2019.
```

```
#                           $rel = '>';
                        }
                    } else { # if ($u_or_d1 eq 'L')
                        if ($u_or_d2 eq 'U') {
                            next; # $rel = '<'.
                        } else { # if ($u_or_d2 eq 'L')
                            $rel = 'S';
                        }
                    }
                    if (defined $rel) {
                        foreach my $blset (@{$blsets_sib}) {
                            push @blocksets, $blset;
                            push @relations, $rel;
                        }
                    }
                } # END of the 5th inner foreach-loop (over $u_or_d2).
            }

    }   # END of the 5th middle foreach-loop (over siblings).

  } # END of "if ( (($pa1 == $top_node) and (@{$sibs1} == 3)) or  (($pa1 != $top_node) and (@{$sibs1} == 2)) ) { # ADDED on Jan 16, 2019. ...}"

        $interfering_blocksets[$bl1] = (@blocksets>0) ? [\@blocksets, \@relations] : [ ]; # CORRECTED on 2019/03/09.
        #$interfering_blocksets[$bl1] = [\@blocksets, \@relations];

} # END of the outer for-loop (over $bl1).
```

(4) Fourth, Construct @cmpl_interfering_blocksets using @interfering_blocksets:

```
# the "complement" of @interfering_blocksets,
# denoted as @cmpl_interfering_blocksets, for which the subject is
# each constituent of each block-set recorded in @interfering_blocksets.
# More precisely,
# @{$cmpl_interfering_blocksets[$bl]} = (\@cmpl_blocksets, \@relations)
# records block-blockset pairs in @interfering_blocksets in which the $bl th block is
# a constituent of @blockset = @{$interfering_blocksets[$bl2]->[0][$k2]} for some $bl2 and $k2.
# (It is empty if the $bl th block is not a constituent of any block-sets.)
# We will use the convention:
# @{$cmpl_blocksets[$k]} = ($bl2, @blockset with $bl removed)
# and
# $relations[$k] = the "complement" of $interfering_blocksets[$bl2]→[1][$k2].
#
# Thus, the relations should be:
#       'S' if the corresponding relation in @interfering_blocksets is 'S',
#       '<(pa)' if the corresponding relation in @interfering_blocksets is '>(ch)'. # REVISED on Jan 16, 2019.
#       # '<' if the corresponding relation in @interfering_blocksets is '>'.

        # (o) Initialization. #

my @cmpl_interfering_blocksets = ();
for (my $bl=0; $bl < $B; $bl++) { $cmpl_interfering_blocksets[$bl] = [[], []]; }

        # (i) Raw construction. #
```

```
for (my $bl2=0; $bl2 < $B; $bl2++) { # Outer for-loop (over $bl2). #

    my ($blocksets2, $relations2) = @{$interfering_blocksets[$bl2]};
    unless (defined $blocksets2) { next; }

    my $ct_blksets2 = @{$blocksets2};
    for (my $k2=0; $k2 < $ct_blksets2; $k2++) { # Middle for-loop (over block-sets).

        my $blkset2 = $blocksets2→[$k2];
        my $ct_blks2 = @{$blkset2};
        my $rel2 = $relations2→[$k2];
        my $rel = ($rel2 eq 'S') ? 'S' : '<(pa)' ; # REVISED on Jan 16, 2019.
#       my $rel = ($rel2 eq 'S') ? 'S' : '<' ;

        my @blkset_1st_half = ($bl2);
        my @blkset_2nd_half = @{$blkset2};
        for (my $i = 0; $i < $ct_blks2; $i++) {

            my $bl = shift @blkset_2nd_half;
            my @blkset = (@blkset_1st_half, @blkset_2nd_half);

            my ($cmpl_blocksets, $relations) = @{$cmpl_interfering_blocksets[$bl]};
            push @{$cmpl_blocksets}, \@blkset;
            push @{$relations}, $rel;

                # Prepare for the next round. #
            push @blkset_1st_half, $bl;
        }

    } # END of the middle for-loop (over block-sets).

}  # END of the outer for-loop (over $bl2). #


        # (ii) Cleaning up.

for (my $bl=0; $bl<$B; $bl++) { # Outer for-loop (over $bl).
        my ($cmpl_blocksets, $relations) = @{$cmpl_interfering_blocksets[$bl]};
        unless ((defined $cmpl_blocksets) and (@{$cmpl_blocksets}>0))
{ $cmpl_interfering_blocksets[$bl] = []; } # CORRECTED on 2019/03/10.
        #unless (defined $cmpl_blocksets) { $cmpl_interfering_blocksets[$bl] = []; }
}  # END of outer for-loop (over $bl).
```

APPENDIX F-sppl G: Computing sizes of blocks. # ADDED on Jan 15, 2019. # Re-labelled on Jan 18, 2019.

Here, we describe how to compute **@block_sizes**,
where **$block_sizes[$bl]** is the size of the **$bl** th block.

```
my @block_sizes = ();

for (my $bl = 0; $bl < $B; $bl++) { # Outer for-loop (over $bl).

        my ($lbd, $rbd) = @{$bds_blocks[$bl]};
```

```
        my $size = $rbd - $lbd + 1;

                # Subtract the sizes of (vertically) 'including' or 'equivalent' blocks, if at all. #
                        # (ADDED on Nov 29, 2018.)

        my $relations_w_bl = $inter_block_relations[$bl];
        my %clm2including;  # ADDED on Jan 15, 2019.

        for (my $bl2 = 0; $bl2 < $B; $bl2++) { # Middle for-loop (over $bl2).
                if ($bl2 == $bl) { next; }
                my $rel = $relations_w_bl→[$bl2];
                unless (($rel eq '<') or (($rel eq '=') and ($bl2 < $bl))
                        or ($rel eq 'ONN') or ($rel eq 'ONCS') ) { next; } # $bl2 is NEITHER
vertically including NOR vertically equivalent to (and higher-ranked than) $bl, NOR (overlap in a
non-nested manner with $bl) (ADDED on Jan 15, 2019).

                my ($lbd2, $rbd2) = @{$bds_blocks[$bl2]};

                        # MODIFIED on Jan 15, 2019. #

                if (($rbd < $lbd2) or ($rbd2 < $lbd)) { next; } # $bl2 does NOT overlap $bl.

                for (my $c = $lbd2; $c <= $rbd2; $c++) { # Inner for-loop (over columns in $bl2).
                    my $including = $clm2including{$c};
                    unless (defined $including) { $including = $clm2including{$c} = []; }
                    push @{$including}, $bl2;
                } # End of the inner for-loop (over columns).

                        # OBSOLETE as of Jan 15, 2019. #
#               unless (($lbd <= $lbd2) and ($rbd2 <= $rbd)) { next; } # $bl2  is NOT included in
$bl.
#               $size2 = $rbd2 - $lbd2 + 1;
#               $size -= $size2;
                        # END of "OBSOLETE as of Jan 15, 2019." #

                        # END of "MODIFIED on Jan 15, 2019." #

        }  # End of the middle for-loop (over $bl2).

                # ADDED on Jan 15, 2019. #

        for (my $c = $lb; $c <= $rb; $c++) { # 2nd middle for-loop (over columns in $bl).

                if (defined $clm2including{$c}) { $size--; }
        } # End of the 2nd middle for-loop (over columns in $bl).

        $block_sizes[$bl] = $size;

                # END of "ADDED on Jan 15, 2019." #

}   # END of the outer for-loop (over $bl).

        # END of "ADDED on Jan 15, 2019." #


                # ADDED on Jan 17, 2019. (2) #
```

APPENDIX F-sppl2 H: CEncoding alignment topology. # (Re-labelled on Jan 18, 2019; Title revised on Jan 22, 2019.) #

my **$code_topology** = **encode_alignment_topology** (@bds_blocks, @inter_block_relations, @interfering_blocks (added 2019/01/18), @interfering_blocksets (added 2019/01/18), @blocks_w_spec_lb (added 2019/01/20), {other necessary things});

END of "ADDED on Jan 17, 2019. (2)"

ADDED on Jan 18, 2019. (2)

 # @interfering_blocks, where
 # @{$interfering_blocks[$bl]} = (\@blocks, \@relations) stores information on blocks that can
interfere with the $bl th block (or empty if it has no such blocks), where
 # $blocks[$k] is the index (or rank) of the $k th interfering block,
 # $relations[$k] is the relation of the $k th interfering block with the $bl th block.
#
Here, we will only consider the following relations:
'S' (for 'sibling/parent/child' of the $bl th block),
'Cp' (for 'complementary'),
'>(ch)' (for the $bl2 th block being 'vertically included' in, and an **"effective child"** of, the $bl1 th block), # MODIFIED on Jan 16, 2019.
'<(pa)' (for the $bl2 th block 'vertically including', and being an **"effective parent"** of, the $bl1 th block), # MODIFIED on Jan 16, 2019.
'=' (for 'vertically identical').
#
NOTES:
(1) Blocks with '**ONN**' relations cause NO topological changes, as far as we employ the current coordinate system;
(2) Also, blocks with **'ONCS'** relations cause NO topological changes, *as far as* we restrict ourselves to parsimonious indel histories (more precisely, parsimonious ancestral gap states);
(3) Also, *as far as* we restrict ourselves to parsimonious indel histories (more precisely, parsimonious ancestral gap states), the separating branches of blocks with the **'>'** or **'<'** relations with the $bl th block *must be* either the **child or parent** (or a sibling if they are children of a trivalent root) of the separating branch of the $bl th block. (As of Jan 16, 2019, this condition trivially holds.)

@{$interfering_blocksets[$bl]} = (\@blocksets, \@relations) are nearly the same as above, except that **@{$blocksets[$k]}** is now a set of indices (or ranks) of the $k th interfering block set. (NOTE: Each @{$blockset[$k]} contains only blocks with the same horizontal size that form a (complementary) monophyletic group.)

For the moment, we will only consider the following relations:

'S' (for 'sibling/parent/child' of the $bl th block),
'>(ch)' (for the blockset being 'vertically included' in, and an "effective child" of, the $bl1 th block). # Modified on Jan 18, 2019.

NOTES:
(5) As in note (3), the separating branch of the blockset with the **'>'** relation must be either the **child or parent or sibling** of the separating branch of the $bl th block;
(6) Each blockset with the '**Cp**' relation likely involves a single block with the 'S' relation;
(7) Likewise, **'='** likely involves a single block with the 'S' relation;
(8) And **'<'** likely involves a single block with the '=', '>', and/or the 'S' relations.

(1) The main subroutine, **"encode_alignment_topology (….)"**

```
# ASSUME that all relevant binary relations can uniquely determine the alignment topology.
# (=> MUST be proven later.)

my @binary_relations = ();

for (my $bl1=0; $bl1 < $ub_bl1+1; $bl1++) { # Outermost for-loop (over $bl1).

    my ($blocks, $relations) = @{$interfering_blocks[$bl1]};
    my ($lb1, $rb1) = @{$bds_blocks[$bl1]}; # Added on Jan 20, 2019.
    my $rels_w_bl1 = $inter_block_relations[$bl1];  # Added on Jan 20, 2019.
    my $ct_blks = @{$blocks};
    for (my $k=0; $k <$ct_blks; $k++) { # 1st middle for-loop (over blocks). #
        my $bl2 = $blocks→[$k];
        unless ($bl1 < $bl2) { next; }
        my $rel = $relations→[$k];
        my ($lb2, $rb2) = @{$bds_blocks[$bl2]};  # Added on Jan 20, 2019.
        my $rels_w_bl2 = $inter_block_relations[$bl2]; # Added on Jan 20, 2019.

            # ADDED on Jan 20, 2019. (1) #
        if ($rel eq '=') {
            next; # $bl1 and $bl2 must always be separated, when encoding the alignment topology.
This means that, in this case, the positional relation between $bl1 and $bl2 will NEVER influence
the alignment topology.

        } elsif ($rel eq 'Cp') { # $bl1 and $bl2 must NEVER overlap (horizontally) when encoding
the alignment topology. #
            my $if_separated = 1;
            if ($rb1 +1 < $lb2) {  # $bl2 is on the right of $bl1, and they are separated at least
superficially. #

                ($if_separated, my $lb2_new, my $rb2_new) = extend_left_end_to_left ($lb2, $rb2,
$lb1, $rb1, @bds_blocks, @blocks_w_spec_rb, @{$rels_w_bl2}); # See (2a) below. (added on
2019/01/22)  # ADDED on Jan 21, 2019.

            } elsif ($rb2 + 1 < $lb1) {# $bl2 is on the left of $bl1, and they are separated at least
superficially. #
                ($if_separated, my $lb2_new, my $rb2_new) = extend_right_end_to_right ($lb2,
$rb2, $lb1, $rb1, @bds_blocks, @blocks_w_spec_lb, @{$rels_w_bl2});  # See (2b) below. (added
on 2019/01/22)  # ADDED on Jan 21, 2019.

            } else {
                $if_separated = 0;
            }

            my $bin_rel = ($if_separated) ?  join ('l',  $bl1, $bl2) : join ('-', $bl1, $bl2); # 'l' denotes
"separated", '-' denotes "overlapping or immediately adjacent".
            push @binary_relations, $bin_rel;

        } elsif ($rel eq 'S') {

            my $bin_rel ;
            if ($lb1 < $lb2) {
                if ($rb1 < $rb2) { # Non-nested.
                    $bin_rel = join ('^', $bl1,  $bl2);

                } else {  # Nested ($bl1 horizontally includes $bl2).
                    $bin_rel = join (>', $bl1, '$bl2);
                }
```

```
        } elsif ($lb2 < $lb1) {
            if ($rb2 < $rb1) { # Non-nested.
                $bin_rel = join ('^', $bl1, $bl2);

            } # Nested ($bl1 is horizontally included in $bl2).
                $bin_rel = join ('<', $bl1, $bl2);
            }
        } else { # if ($lb1 == $lb2)
            if ($rb1 < $rb2) { # Nested ($bl1 is horizontally included in $bl2).
                $bin_rel = join ('<', $bl1, $bl2);

            } elsif ($rb2 < $rb1) { # Nested ($bl1 horizontally includes $bl2).
                $bin_rel = join ('>', $bl1, $bl2);

            } else { # Nested ($bl1 and $bl2 are horizontally identical.)
                $bin_rel = join ('=', $bl1, $bl2);
            }
        }

         push @binary_relations, $bin_rel; # Added on Jan 21, 2019.

            # Consider whether a third block can change the horizontal positional relation between siblings, $bl1 and $bl2.
            # Assume that we currently have $lb1 < $lb2, for example.
            # The left-bound, $lb2, of $bl2 could be "extended" further to the left
            #  with the aid of, e.g., $bl3, which either (i) includes $bl2 or (ii) overlaps $bl2 in a non-nested manner.
            # (i) when $bl3 includes $bl2, $bl3 either (a) includes $bl1 as well, (b) overlaps $bl1 in a non-nested manner, or (c) is a (vertical) complement of $bl1.
            #  In case (b) or (c), however, $bl3 must split $bl1, which can NEVER happen in the current coordinate-assigning convention. Thus, $bl3 must ALWAYS include $bl1 as well.
            # (ii) when $bl3 overlaps $bl2 in a non-nested manner, $bl3 must always include $bl1.
            # Thus, regardless of case (i) or case (ii),  $bl3 must ALWAYS include $bl1.
            # Then, $bl3 can NEVER "extend" the left-end of $bl2 UP TO or BEYOND $lb1;
            # otherwise, $lb1 should have been equal to (or on the right of) $lb2 EVEN BEFORE $bl3 influences the boundary of $bl2.
            #
            # In conclusion, NO third block CAN change the horizontal positional relation between siblings, $bl1 and $bl2.
            #
            # # (Actually, this argument holds even when $bl1 and $bl2 are vertically "non-interfering", thus can be applied also when $bl1 is a "sibling" of a blockset.) ... WRONG!! # (However, we could restrict the possibility to the cases where $bl3 vertically includes some members of the blockset, because non-nested overlap must NEVER happen. ... WRONG AGAIN!! )

    } elsif (($rel eq '>(ch)') or ($rel eq '<(pa)')) { # In this case, the binary relation itself is, after all, "parent-child", but the roles of $bl1 and $bl2 can be swapped.

        my ($bl_ch, $bl_pa) = ($rel eq '>(ch)') ? ($bl2, $bl1) : ($bl1, $bl2);
        my $rels_w_ch = $inter_block_relations[$bl_ch];
        my ($lb_ch, $rb_ch) = @{$bds_blocks[$bl_ch]};
        my ($lb_pa, $rb_pa) = @{$bds_blocks[$bl_pa]};

            # ADDED on Jan 21, 2019. (1) #

        my $if_separated = 1;
```

```
            if ($rb_pa +1 < $lb_ch) {  # $bl_ch is on the right of $bl_pa, and they are separated at least superficially. #

                ($if_separated, my $lb_ch_new, my $rb_ch_new) = extend_left_end_to_left ($lb_ch, $rb_ch, $lb_pa, $rb_pa, @bds_blocks, @blocks_w_spec_rb, @{$rels_w_ch}); # See (2a) below. (added on 2019/01/22)  # ADDED on Jan 21, 2019.

            } elsif ($rb_ch + 1 < $lb_pa) {# $bl_ch is on the left of $bl_pa, and they are separated at least superficially. #

                ($if_separated, my $lb_ch_new, my $rb_ch_new) = extend_right_end_to_right ($lb_ch, $rb_ch, $lb_pa, $rb_pa, @bds_blocks, @blocks_w_spec_lb, @{$rels_w_ch}); # See (2b) below. (added on 2019/01/22)  # ADDED on Jan 21, 2019.

            } else {
                $if_separated = 0;
            }

            my $bin_rel = ($if_separated) ?  join ('|', $bl1,  $bl2) : join ('-', $bl1,  $bl2); # '|' denotes "separated", '-' denotes "overlapping or immediately adjacent".
            push @binary_relations, $bin_rel;

                # END of "ADDED on Jan 21, 2019. (1)" #

        } # End of "if ($rel eq '=') {...} elsif ($rel eq 'Cp') {...} elsif ($rel eq 'S') {...} elsif (($rel eq '>(ch)') or ($rel eq '<(pa)')) {...}". #

#           'S' (for 'sibling/parent/child' of the $bl th block),
#           'Cp' (for 'complementary'),
#           '>(ch)' (for the $bl2 th block being 'vertically included' in, and an "effective child" of, the $bl1 th block), # MODIFIED on Jan 16, 2019.
#           '<(pa)' (for the $bl2 th block 'vertically including', and being an "effective parent" of, the $bl1 th block), # MODIFIED on Jan 16, 2019.
#           '=' (for 'vertically identical').

                # END of "ADDED on Jan 20, 2019. (1)" #


    } # End of the 1st middle for-loop (over blocks). #

    my ($blocksets, $relations) = @{$interfering_blocksets[$bl1]};
    my $ct_blsets = @{$blocksets};

    for (my $k=0; $k <$ct_blsets; $k++) {  # 2nd middle for-loop (over block-sets). #
        my $blset = $blocksets→[$k];
        my $rel = $relations→[$k];

        my $cnct_blset = join (',', @{$blset});

        if ($rel eq 'S') {


                # NOTE (added on 2019/01/21): The situation gets extremely complicated if we consider the extension of the boundaries of the relevant blocks. Thus, FOR THE MOMENT, we will OMIT the boundary extension. (And we will REVISIT the issue when we have time.)
                # (I have a hunch, however, that extending the boundaries will NOT influence the results (as in the case of single-block siblings, $bl1 and $bl2).)
```

```
#               my ($lb_min, $rb_min) = ($lb1, $rb1); # Added on Jan 21, 2019. & OBSOLETE immediately after that.

                my $if_nesting = 1; # Whether the block-set "nest"s the $bl1 or not.
                my $if_equal = 1; # Whether all blocks in the block-set share the same boundaries.
                my ($lb_min, $rb_min) = my ($lb_sh, $rb_sh) = @{$bds_blocks[$blset->[0]]};
                foreach my $bl2 (@{$blset}) { # 1st inner foreach-loop (over $bl2).

                    # ADDED on Jan 21, 2019. (5a) #

                    my ($lb2, $rb2) = @{$bds_blocks[$bl2]};

                    if (($lb1 < $lb2) or ($rb2 < $rb1)) {
                        $if_nesting = 0;
                        # $if_equal = 0;
                        #last;
                    }
                    unless (($lb_sh == $lb2) and ($rb_sh == $rb2)) { $if_equal = 0; }
                    # unless (($lb1  == $lb2) and ($rb1  == $rb2)) { $if_equal = 0; }

                    if ($lb_min < $lb2) { $lb_min = $lb2; }
                    if ($rb2 < $rb_min) { $rb_min = $rb2; }

                    if ($rb_min < $lb_min) {
                        $if_nesting = $if_equal = 0;
                        last;
                    }

                    # End of "ADDED on Jan 21, 2019. (5a)" #

                } # End of the 1st inner foreach-loop (over $bl2).

                    # ADDED on Jan 21, 2019. (5b) #

                my $bin_rel;
                if ($if_equal) {
                    if (($lb_sh == $lb1) and ($rb_sh == $rb1)) {
                        $bin_rel = join ('=', $bl1, $cnct_blset);
                    } elsif ($if_nesting) {
                        $bin_rel = join ('<', $bl1, $cnct_blset);
                    } elsif (($lb1 <= $lb_sh) and ($rb_sh <= $rb1)) {
                        $bin_rel = join ('>', $bl1, $cnct_blset);

                    } else {
                        $bin_rel = join ('^', $bl1, $cnct_blset);
                    }

                } elsif (($lb_min <= $rb_min) and ($lb1 <= $lb_min) and ($rb_min <= $rb1)){
                     my @min_blks = ();
                     foreach my $bl2 ($bl1, @{$blset}) { # 2nd inner foreach-loop (over $bl2 (including $bl1)).
                            my ($lb2, $rb2) = @{$bds_blocks[$bl2]};
                            if (($lb2 == $lb_min) and ($rb2 == $rb_min)) { push @min_blks, $bl2; }
                    }  # End of the 2nd inner foreach-loop (over $bl2 (including $bl1)).
                    $bin_rel = (@min_blks>0) ?
                            join ('', $bl1, ',', $cnct_blset, '>', join (',', @min_blks))  :
                            join ('^', $bl1, $cnct_blset) ;
```

```
            } elsif ($if_nesting) {
                $bin_rel = join ('<', $bl1, $cnct_blset);

            } else {
                $bin_rel = join ('^', $bl1, $cnct_blset);
            }

            push @binary_relations, $bin_rel;

                # End of "ADDED on Jan 21, 2019. (5b)" #


        } elsif ($rel eq '>(ch)') {

            my $bl_pa = $bl1;
            my ($lb_pa, $rb_pa) = ($lb1, $rb1);

            my $if_separated = 0;
            foreach my $bl_ch (@{$blset}) { # 3rd inner foreach-loop (over $bl_ch).

                # ADDED on Jan 21, 2019. (2) #

                my ($lb_ch, $rb_ch) = @{$bds_blocks[$bl_ch]};
                my $rels_w_ch = $inter_block_relations[$bl_ch];

                my $if_separated2 = 1;
                if ($rb_pa +1 < $lb_ch) {  # $bl_ch is on the right of $bl_pa, and they are separated at least superficially. #

                    ($if_separated2, my $lb_ch_new, my $rb_ch_new) = extend_left_end_to_left ($lb_ch, $rb_ch, $lb_pa, $rb_pa, @bds_blocks, @blocks_w_spec_rb, @{$rels_w_ch}); # See (2a) below. (added on 2019/01/22)  # ADDED on Jan 21, 2019.

                } elsif ($rb_ch + 1 < $lb_pa) {# $bl_ch is on the left of $bl_pa, and they are separated at least superficially. #

                    ($if_separated2, my $lb_ch_new, my $rb_ch_new) = extend_right_end_to_right ($lb_ch, $rb_ch, $lb_pa, $rb_pa, @bds_blocks, @blocks_w_spec_lb, @{$rels_w_ch}); # See (2b) below. (added on 2019/01/22)  # ADDED on Jan 21, 2019.

                } else {
                    $if_separated2 = 0;
                }

                if ($if_separated2) {
                    $if_separated = 1;
                    last;
                }

                # End of "ADDED on Jan 21, 2019. (2)" #

            } # End of the 3rd inner foreach-loop (over $bl_ch).


            my $bin_rel = ($if_separated) ?  join ('|',  $bl1, $cnct_blset) : join ('-', $bl1, $cnct_blset); # '|' denotes "separated", '-' denotes "overlapping or immediately adjacent".  # ADDED on Jan 21, 2019.
            push @binary_relations, $bin_rel; # ADDED on Jan 21, 2019.
```

```
        }  # End of "if ($rel eq 'S') {…} elsif  ($rel eq '>(ch)') {...}"

#           'S' (for 'sibling/parent/child' of the $bl th block),
#           '>(ch)' (for the blockset being 'vertically included' in, and an "effective child" of, the $bl1 th block). # Modified on Jan 18, 2019.

    }  # End of the 2nd middle for-loop (over block-sets). #

} # END of outermost for-loop (over $bl1).

my $code_alignment = join ('; ', @binary_relations);

                # END of "ADDED on Jan 18, 2019. (2)" #


                # ADDED on Jan 22, 2019. (1) #
```

(2a) Satellite subroutine, "**extend_left_end_to_left ($$$$\@\@\@) {...}"**

```
sub extend_left_end_to_left ($$$$\@\@\@) {  # ADDED on Jan 21, 2019. (MOVED from within (a) on Jan 22, 2019.)

        my ($lb2, $rb2, $lb1, $rb1, $bds_blocks, $blocks_w_spec_rb, $rels_w_bl2) = @_;  # ADDED on Jan 21, 2019.

        my $if_separated = 1;  # ADDED on Jan 21, 2019.
        my $left_flanking_bl2 = $blocks_w_spec_rb→[$lb2 -1];
        while (@{$left_flanking_bl2}>0) { # Attempt to "extend" the left-end of $bl2 to the left.
            my $if_rlv = 0;
            foreach my $bl3 (@{$left_flanking_bl2}) {
                my $rel23 = $rels_w_bl2→{$bl3};
                unless (($rel23 eq '<') or ($rel23 eq '<(pa)') or ($rel23 eq 'ONN') or ($rel23 eq 'ONCS')) { next; }
                my ($lb3, $rb3) = @{$bds_blocks→[$bl3]};
                $lb2 = $lb3;  # Update $lb2.
                if (($rel23 eq 'ONN') or ($rel23 eq 'ONCS')) { $rb2 -= $rb3 - $lb3 + 1; }
                $if_rlv = 1;
                last;
            }
            unless ($if_rlv) { last; }

            if ($rb1 + 1 < $lb2) {
                $left_flanking_bl2 = $blocks_w_spec_rb[$lb2-1]; # Update $left_flanking_bl2.
            } else {
                if ($lb1 <= $rb2+1) { $if_separated = 0; }
                last;
            }
        } # End of "while (@{$left_flanking_bl2}>0) {...}"

        return ($if_separated, $lb2, $rb2);  # ADDED on Jan 21, 2019.

}    # END of "sub extend_left_end_to_left ($$$$\@\@\@) {...}"  # Paired with "ADDED on Jan 21, 2019. (MOVED from within (a) on Jan 22, 2019.)"
```

(2b) Satellite subroutine, "**extend_right_end_to_right ($$$$\@\@\@) {...}"**

```
sub extend_right_end_to_right ($$$$\@\@\@) {  # ADDED on Jan 21, 2019. (MOVED from within (a) on Jan 22, 2019.)

        my ($lb2, $rb2, $lb1, $rb1, $bds_blocks, $blocks_w_spec_lb, $rels_w_bl2) = @_;  # ADDED on Jan 21, 2019.

        my $if_separated = 1;  # ADDED on Jan 21, 2019.

        my $right_flanking_bl2 = blocks_w_spec_lb→[$rb2 +1];
        while (@{$right_flanking_bl2}>0) { # Attempt to "extend" the right-end of $bl2 to the right.
            my $if_rlv = 0;
            foreach my $bl3 (@{$right_flanking_bl2}) {
                my $rel23 = $rels_w_bl2→{$bl3];
                unless (($rel23 eq '<') or ($rel23 eq '<(pa)') or ($rel23 eq 'ONN') or ($rel23 eq 'ONCS')) { next; }
                my ($lb3, $rb3) = @{$bds_blocks[$bl3]};
                $rb2 = $rb3;  # Update $rb2.
                if (($rel23 eq 'ONN') or ($rel23 eq 'ONCS')) { $lb2 += $rb3 - $lb3 + 1; }
                $if_rlv = 1;
                last;
            }
            unless ($if_rlv) { last; }

            if ($rb2 + 1 < $lb1) {
                $right_flanking_bl2 = $blocks_w_spec_lb[$rb2+1]; # Update $right_flanking_bl2.
            } else {
                if ($lb2 <= $rb1+1) { $if_separated = 0; }
                last;
            }
        } # End of "while (@{$right_flanking_bl2}>0) {...}"

        return ($if_separated, $lb2, $rb2);  # ADDED on Jan 21, 2019.

}    # END of "sub extend_right_end_to_right ($$$$\@\@\@) {...}"  # Paired with "ADDED on Jan 21, 2019. (MOVED from within (a) on Jan 22, 2019.)"

                # End of "ADDED on Jan 22, 2019. (1)" #
```