Reweave function outline

May 7th, 2024

1 Classes

General notes:

- indices for positions are always 0-based.
- indices for intervals are always closed on the left and open on the right, as in python.

1.1 Main classes

`Pangraph`

`paths`	dict(path_id → `Path`)	dictionary of paths
`blocks`	dict(block_id → `Block`)	dictionary of blocks
`nodes`	dict(node_id → `Node`)	dictionary of nodes

`Path`

`id`	int	path id
`L_tot`	int	total fasta record length
`nodes`	list(node_id)	list of node ids
`circular`	bool	whether the genome is circular

`Block`

`id`	int	block id
`consensus`	str	consensus nucleotide sequence
`alignment`	dict(node_id → `Edit`)	dict of edits per node id

`Node`

`id`	int	node id
`block_id`	int	block id
`path_id`	int	path id
`strandedness`	bool	forward or reverse
`position`	(int, int)	start/end coordinates on genome (0-based and right-end excluded). It can be start > end only for a node that wraps around the end of the genome in a circular path.

1.2 Block alignment

Edit

`ins`	list(`Insertion`)	list of insertions
`dels`	list(`Deletion`)	list of deletions
`subs`	list(`Substitution`)	list of substitutions

Insertion

`pos`	int	position of the insertions, indicated as the nucleotide position on the block consensus after the insertion. pos $= 0$ indicates an insertion at the beginning of the sequence, while pos $= L$ for a block of length L indicates an insertion at the end of the block.
`ins`	str	inserted sequence

Deletion

`pos`	int	position of the first deleted nucleotide on the block consensus
`len`	int	length of the deletion

Substitution

`pos`	int	position of the substitution on the block consensus
`alt`	char	substituted nucleotide

1.3 Homology between blocks

Hit

`name`	int	block id
`length`	int	length of the full sequence (not just the match!)
`start`	int	start position of match on block consensus
`stop`	int	end position of match on block consensus

`Alignment`

`qry`	hit	query hit
`reff`	hit	reference hit
`orientation`	bool	orientation of the alignment: forward or reverse
`matches`	int	number of matched nucleotides in the cigar string
`length`	int	alignment length
`quality`	int	alignment quality
`cigar`	str	cigar string
`divergence`	float	divergence
`align`	float	alignment score
`new_block_id`	int	id for the new block created by the merger. It can also be used as alignment id. Only assigned to alignments that pass filtering.
`anchor_block`	str	Either `reff` or `qry` (can be an *enum* in Rust). Indicates which of the two blocks is the anchor for the merging. The anchor block is the one to which the sequences of the other block are appended at merging. This attribute gets assigned only to alignments that pass the filtering.

1.4 Block splitting

`Interval`

`start`	int	start position on block consensus sequence
`end`	int	end position on block consensus sequence
`aligned`	bool	*aligned* or *leftover* chunk
`new_block_id`	int	Only for `aligned` intervals. Id of the new block created by the merger. Can be used to identify the alignment.
`is_anchor`	bool	Only for `aligned` intervals. Whether the interval is on the anchor for the merging.
`orientation`	bool	Only for `aligned` intervals. Orientation of the alignment with respect to the other block, either forward or reverse.

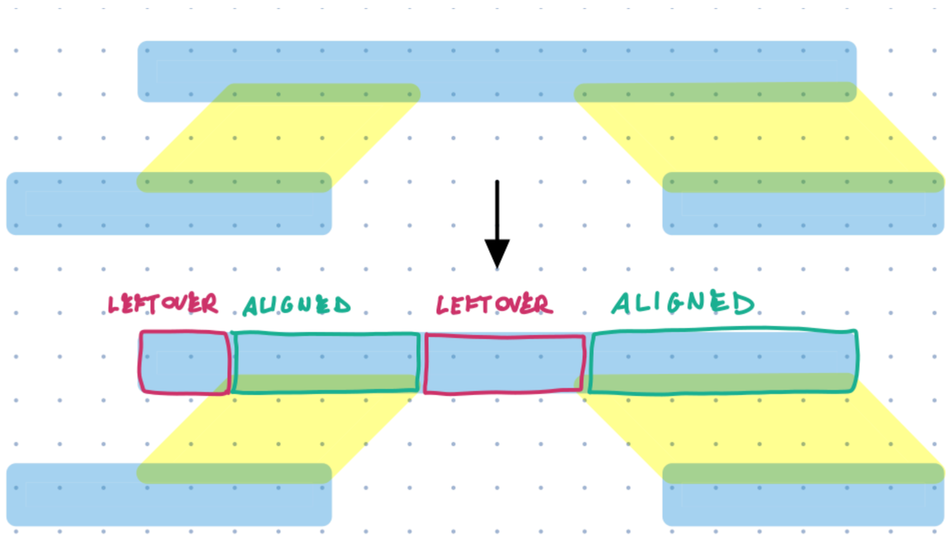


Figure 1: When splitting a block along alignments, the resulting chunks can be either *aligned* or *leftover*. The former are intervals that will be merged with other intervals from other blocks, while the latter will be directly updated in the path without further processing.

1.5 Merge promises and graph updates

Note: when merging two blocks, one is used as ***anchor block*** and the other as ***append block***. Sequences from the latter are reconstructed and re-aligned to the consensus of the anchor block.

`ToMerge`

`block`	block	block destined to merging
`is_anchor`	bool	whether the block is the anchor in the merging
`orientation`	bool	whether the merging is forward or reverse

Note: the `ToMerge` class is used as an intermediate step between block splitting and creation of a merge promise. The splitting of a block returns block intervals that will need to be merged with other intervals from other blocks, and will be paired in merge promises. Before pairing, these alignable intervals are emitted as `ToMerge` objects.

`MergePromise`

`anchor_block`	`Block`	the anchor block
`append_block`	`Block`	the append block
`orientation`	bool	orientation of the match, forward or reverse

GraphUpdate

`b_old_id`	int	id of the old block that was split and needs to be removed from the graph.
`b_new`	list(`Block`)	list of all newly-created blocks that are *leftovers* (see Figure 1). These will be directly added to the block dictionary
`n_new`	dict(node_id→ list(`Node`))	dictionary of node subsitutions. For every old node_id, provides a list of new `Node` objects that need to be inserted in this position in the path. The orientation is already correct and does not need to be flipped based on the node strandedness. This dictionary contains *all* nodes updates, both for *leftover* and *aligned* blocks.

2 Functions

symbol	description
G	graph
b	block
π	path
n	node
ε	edit
σ	strandedness
P	merge promises
μ	match/alignment object
u	graph update
ι	interval
I	set of intervals

Table 1: Symbols used in the function descriptions.

reweave ($G, \vec{\mu}, \tau$) $\rightarrow$ (G', P)

input:

- ▸ G : graph
- ▸ $\vec{\mu}$: matches
- ▸ τ : threshold block length (default = 100 bp)

output:

- ▸ G' : updated graph
- ▸ P : merge promises

function:

- modify matches $\vec{\mu}$ inplace by:
  - assigning new block ids to matches with **assign_new_block_ids**$(\vec{\mu})$.
  - assigning the anchor block with **assign_anchor_block**$(\vec{\mu}, G)$.
- create dictionary $D = \{b_{\text{id}} \to \vec{\mu}\}$ with **target_blocks**$(\vec{\mu})$, assigning to only block ids that have a match their corresponding matches.
- for every $(b_{\text{id}}, \vec{\mu}) \in D$ (this **could be parallelized**):
  - $u, h \leftarrow$ **split_block**$(b_{\text{id}}, \vec{\mu}, G, \tau)$
  - collect graph updates in $u \to U$ and `ToMerge` objects $h \to H$.
- group `ToMerge` objects in `MergePromises` with **group_promises**(H).
- for every graph update $u \in U$
  - update the graph path with **update_graph**(G, u)
- return the updated graph G and a list of merge promises

Note: this is the main function for this update. It takes care of:
- splitting blocks along alignment matches, creating new block and nodes and their corresponding ids.
- updating all the paths with the new nodes.

- split blocks can be categorized as *aligned* and *leftover*. The latter are included directly in the graph blocks, while the former are returned as *merge promises* whose resolution will generate a new block to be later added to the graph. This can be parallelized.

2.1 Preprocessing

`assign_new_block_ids` ($\vec{\mu}$)

input:

▸ $\vec{\mu}$: matches (modified inplace)

function:

for every match μ, assigns a new block id μ_{id}. This is currently calculated as

$$\mu_{\text{id}} = \text{hash}(\text{qry}.b_{\text{id}}, \text{qry.start}, \text{qry.stop}, \text{reff}.b_{\text{id}}, \text{reff.start}, \text{reff.stop})$$

Node: currently the **id of new aligned blocks** is created by hashing the ids of the qry and ref blocks, together with the coordinates of the match, see Section 3.1. This should be deterministic and parallelizable. Alternatively, if the tree traversal is not parallelized, this could also be done with a sequential counter.

`assign_anchor_block` ($\vec{\mu}$, G)

input:

▸ $\vec{\mu}$: matches (modified inplace)
▸ G : graph

function:

for every match μ, assigns the anchor block (stored in μ.anchor_block) as:
- the block with highest *depth* (i.e. number of nodes).
- if the depth is equal, the *reference* block in the match.

`target_blocks` ($\vec{\mu}$) $\rightarrow$ (D)

input:

▸ $\vec{\mu}$: matches

output:

▸ D : dictionary of block $\rightarrow$ matches

function:

goes through all mergers μ, and associate each merger to its query and reference block ids in a dictionary $D = \{b_{\text{id}} \rightarrow \vec{\mu}\}$.

2.2 Block splitting

Following alignment, blocks need to be split into chunks, some to be merged with other chunks and some that will constitute leftover blocks.

`split_block` (b_{id}, $\vec{\mu}$, G, τ) $\rightarrow$ (u, H)

input:

- ▸ b_{id} : block id
- ▸ $\vec{\mu}$: matches
- ▸ G : graph
- ▸ τ : threshold block length (default = 100 bp)

output:

- ▸ u : graph update
- ▸ H : list of `ToMerge` objects

function:

- extract a list of hits $T \leftarrow$ **`extract_hits`**$(b_{\mathrm{id}}, \vec{\mu})$ from the alignments. These are dictuionaries containing the new block id, whether the block is an anchor block, the match orientation, and the `Hit` on the block.
- use these hits to split the block into a set of disjoint intervals $I \leftarrow$**`extract_intervals`**(T, L, τ), where L is the length of the block consensus sequence.
- for every interval $\iota \in I$:
  - slice the block along the interval $\left(b^{\mathrm{new}}, n^{\mathrm{dict}}\right) \leftarrow$ **`block_slice`**(b, ι, G). This function returns a new block b^{new} and a dictionary of old node ids to new nodes $n^{\mathrm{dict}} = \left\{n_{\mathrm{id}}^{\mathrm{old}} \to [n^{\mathrm{new}}]\right\}$. Nb: the order of nodes is the one of the old block. This already takes into account merges with reverse orientation, but not the orientation of the old block in the path.
  - the dictionary of new nodes is added to the `GraphUpdate` object u.
  - if the new block slice b^{new} needs to be aligned, it is included in a `ToMerge` object and appended to H.
  - otherwise it is added to the `GraphUpdate` object u.
- finally, for paths in which the original block appears on the reverse strand, the order of nodes in the update is flipped.

`extract_hits` ($b_{\mathrm{id}}, \vec{\mu}$) $\to$ (T)

input:

- ▸ b_{id} : block id
- ▸ $\vec{\mu}$: matches

output:

- ▸ T : list of dictionaries containing:
  - the new block id
  - whether the block is an anchor block
  - the match orientation
  - the `Hit` on the block

function:

for every match μ, checks whether the query of reference (or both) block id is equal to b. If so, it creates and append the hit dictionary to the list.

`extract_intervals` (T, L, τ) $\to$ (I)

input:

- ▶ T : list of hit dictionaries
- ▶ L : block consensus sequence length
- ▶ τ : threshold block length

output:

- ▶ I : list of intervals partitioning the block

function:

- create a set of intervals, using hits edges as breakpoints $I \leftarrow$ `create_intervals`(T, L).
- refine this set of intervals inplace using `refine_intervals`(I, τ). This function joins intervals shorter than the threshold τ with the flanking longest interval. Note that since we pre-filter hits for length, these will be unaligned intervals, and they will be joined with aligned intervals.

`create_intervals` (T, L) $\rightarrow$ (I)

input:

- ▶ T : list of hit dictionaries
- ▶ L : block consensus sequence length

output:

- ▶ I : list of intervals partitioning the block

function:

- sort the hits by start position on the block consensus sequence.
- start a cursor $k = 0$ at the initial position, and iterate over the hits h.
  - if $h_{\text{start}} > k$
    - define an *unaligned* or *leftover* interval $\iota = [k, h_{\text{start}}]$ and append it to the list of intervals I. The **id of new unaligned blocks** is assigned with a hash function: $b^{\text{new}}_{\text{id}} = \text{hash}\left(b^{\text{old}}_{\text{id}}, \iota_{\text{start}}, \iota_{\text{end}}\right)$.
    - update the cursor $k = h_{\text{start}}$.
  - define an *aligned* interval $\iota = \left[h_{\text{start}}, h_{\text{stop}}\right]$ and append it to the list of intervals I. The id of the new block and other properties are inherited from the hit dictionary h.
  - update the cursor $k = h_{\text{stop}}$.
- if $k < L$ append to I a last unaligned interval $\iota = [k, L]$.
- return I.

`refine_intervals` (I, τ)

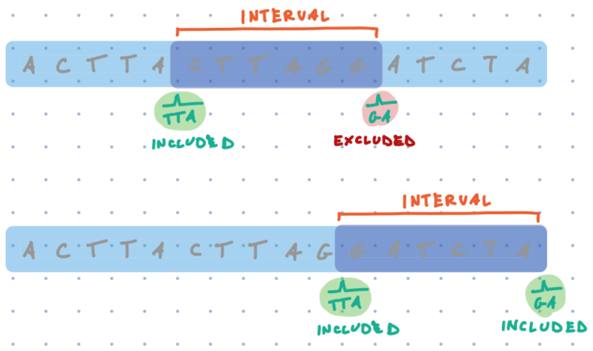
input:

- ▶ I : list of intervals partitioning the block (modified inplace)
- ▶ τ : threshold block length

```
function:
• to define a list of desired mergers, for every interval $\iota = I[n]$
    ‣ if $\text{len}(\iota) < \tau$
        – then the interval $\iota$ should be unaligned and flanked by at least one aligned interval longer than threshold.
        – evaluate $L_l = \text{len}(I[n-1])$ and $L_r = \text{len}(I[n+1])$. Set them to zero if on the border.
        – if $L_l \geq L_r$ append a desired merger $(n \to n-1)$, else append $(n \to n+1)$.
• for every desired merger, *starting from the last one* to avoid index shifting:
    ‣ if the merger is $(n \to n+1)$
        – set $I[n+1]_{\text{start}} = I[n]_{\text{start}}$
        – remove $I[n]$
    ‣ if the merger is $(n \to n-1)$
        – set $I[n-1]_{\text{stop}} = I[n]_{\text{stop}}$
        – remove $I[n]$
```

2.3 Slicing blocks

Given an interval ι and the original block b, we need to slice the block on this interval. This requires also slicing the corresponding alignment. This is straightforward except for insertions. As a rule, insertions at the beginning of the slice are included and at the end of the slice are excluded, except for intervals at the right-edge of the block, for which insertions are kept.


INTERVAL
A C T T A C T T A G G A T C T A
TTA
INCLUDED
G-A
EXCLUDED
INTERVAL
A C T T A C T T A G G A T C T A
TTA
INCLUDED
G-A
INCLUDED

block_slice (b, ι, G) → (b^{new}, n^{dict})

input:

▶ b	:	block
▶ ι	:	interval
▶ G	:	graph

output:

▶ b^{new}	:	block
▶ n^{dict}	:	dict(node_id → list(Node))

function:

- create a new block b^{new}.
- set as consensus sequence the slice the consensus sequence of b on the interval ι.
- for every node n^{old} in the block b:
  - calculate the strandedness of the new sliced nodes:
    - if the interval is not aligned, then it's the same as the old node.
    - if the interval is aligned then the orientation is changed only if the block is not the anchor block, and the alignment orientation is reverse.
  - find the new node start and end positions in the alignment, including indels, $\left(s^{\text{aln}}, e^{\text{aln}}\right) \leftarrow$ **interval_node_coords**(ι, ε, L) function, where L is the old block consensus lenght.
  - find the new node positions in the path, with the $(n^{\text{new}}.s, n^{\text{new}}.e) \leftarrow$ **new_positions** $\left(\left(n_s^{\text{old}}, n_e^{\text{old}}\right), \left(s^{\text{aln}}, e^{\text{aln}}\right), \pi.L, n^{\text{old}}, \sigma\right)$ function.
  - create a new node n^{new} with the new positions, strandedness, block id, and assign it a new node it $n_{\text{id}}^{\text{new}} = \text{hash}(b_{\text{id}}^{\text{new}}, \pi_{\text{id}}, n^{\text{new}}.s, n^{\text{new}}.e)$.
  - add the new node to the dictionary n^{dict}.
  - extract the slice of edits for the new node with the **slice_edits**(ι, ε, L) function.
  - add the new edits to the alignment dictionary of the new block, with the new node id as key.
- return the new block and the dictionary of new nodes.

interval_node_coords (ι, ε, L) $\rightarrow$ (s^{aln}, e^{aln})

input:

▶ ι	:	interval
▶ ε	:	edits
▶ L	:	block consensus sequence length

output:

▶ s^{aln}	:	start position of the slice in the block alignment, including indels
▶ e^{aln}	:	end position of the slice in the block alignment, including indels

function:

- set initially $(s, e) = (\iota.s, \iota.e)$
- for every deletion, decrease the start/end position if the deletion is before the start/end, by an amount equal to the deletion length that lays before the start/end.
- for every insertion, increase the start/end position if the insertion is before the start/end, by an amount equal to the insertion length.
  - insertion at the end position are not included, unless the interval end coincides with the right edge of the block.

new_positions ($\vec{c}_{\text{path}}$, $\vec{c}_{\text{aln}}$, L, σ) $\rightarrow$ (n_s^{new}, n_e^{old})

input:

▶ $\vec{c}_{\text{path}}$	:	old node start/end positions $\left(n_s^{\text{old}}, n_e^{\text{old}}\right)$ in the path
▶ $\vec{c}_{\text{aln}}$	:	positions $\left(s^{\text{aln}}, e^{\text{aln}}\right)$ of the interval on the node sequence, including indels
▶ L	:	block consensus sequence length
▶ σ	:	old node strandedness

output:

- n_s^{new} : new node start position in the path
- n_e^{old} : new node end position in the path

function:

The new node positions are calculated by adding/subtracting to the old node start/end positions the distance of the interval start/end $\left(s^{\mathrm{aln}}, e^{\mathrm{aln}}\right)$ from the old node start/end. This needs to take into account the node strandedness and the periodic boundary conditions.

slice_edits (ι, ε, L) $\rightarrow$ ($\varepsilon^{\mathrm{new}}$)

input:

- ι : interval
- ε : edits
- L : block consensus sequence length

output:

- $\varepsilon^{\mathrm{new}}$: edits

function:

- for every insertion, keep it only if it is included in the interval. Insertion at the left edge of the interval are kept, while insertions on the right edge are excluded, unless the interval extends to the right edge of the block.
- for every deletion, keep it only if it is included in the interval. Possibly change start and length to reduce to the interval.
- for every substitution, keep it only if it is included in the interval.
- the positions of all of the edits are updated to make the 0 position coincide with the interval start.

2.4 Promises and graph update

group_promises (H) $\rightarrow$ (P)

input:

- H : list of `ToMerge` objects

output:

- P : list of `MergePromise` objects

function:

- groups pairs `ToMerge` objects with the same new block id, corresponding to the same alignment.
- for each pair creates a `MergePromise`, assigning the anchor and append blocks and the orientation of the match from the `ToMerge` objects.

update_graph (G, u)

input:

- G : graph
- u : graph update

function:

applies graph modifications inplace:

- remove the old block from G.blocks
- add new *leftover* blocks to G.blocks
- for each node update $\{n_{\text{id}}^{\text{old}} \to [n^{\text{new}}]\}$:
  - finds the path π containing node $n_{\text{id}}^{\text{old}}$
  - creates the list $\overrightarrow{n_{\text{id}}^{\text{new}}}$ of new node ids
  - removes the old node id $n_{\text{id}}^{\text{old}}$ from the path π and replaces it with $\overrightarrow{n_{\text{id}}^{\text{new}}}$
  - removes the old node id n^{old} from the G.nodes dictionary and adds the new nodes $\{n_{\text{id}}^{\text{new}} \to n^{\text{new}}\}$

2.5 Tests

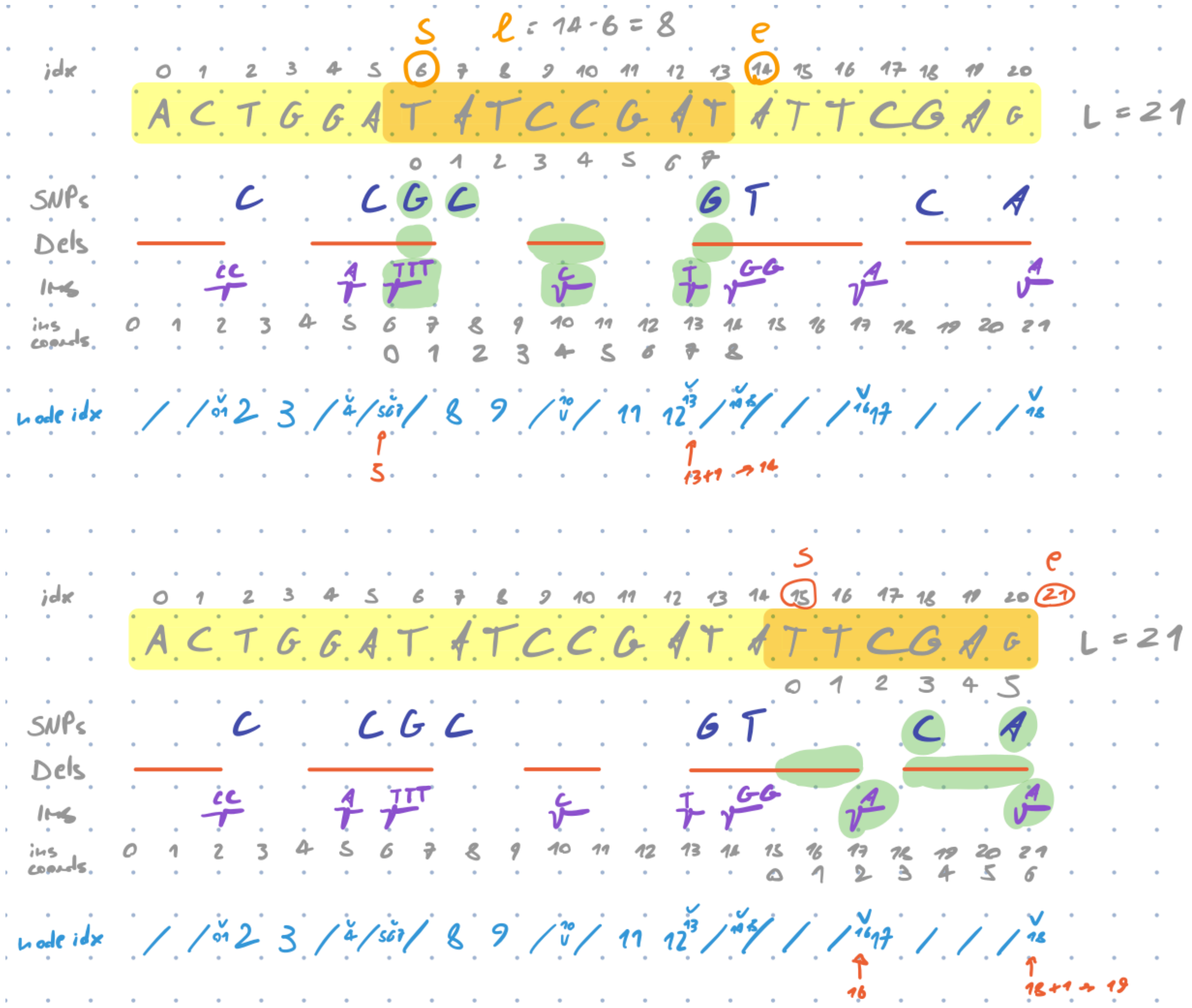


Figure 3: test cases for the functions to split edits in a node.

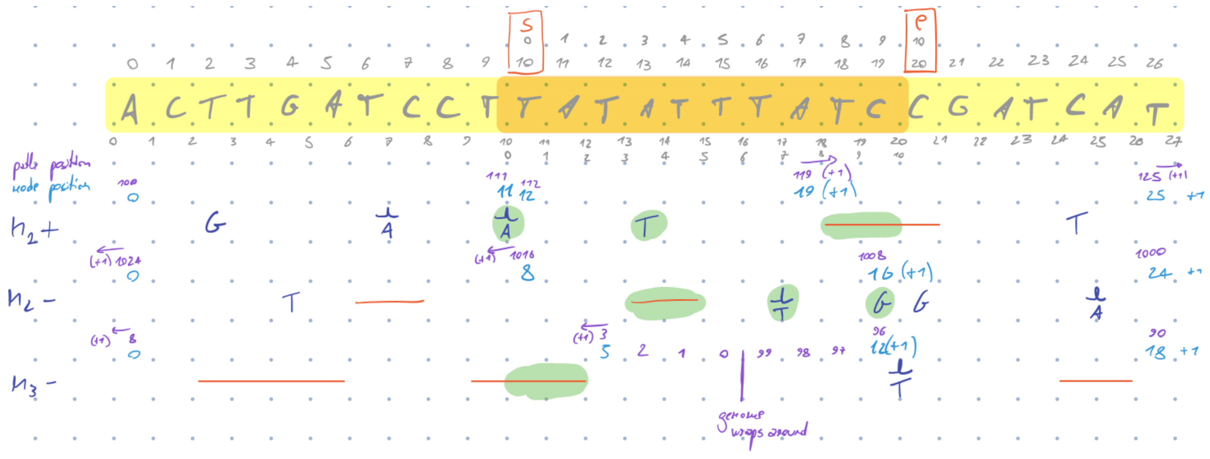


Figure 4: test case for the `block_slice` function.

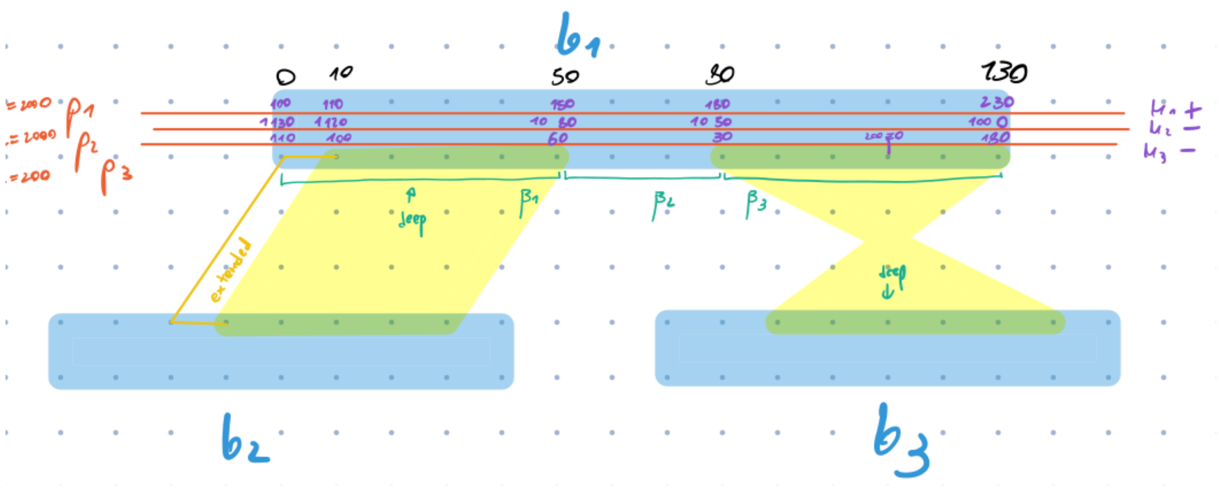
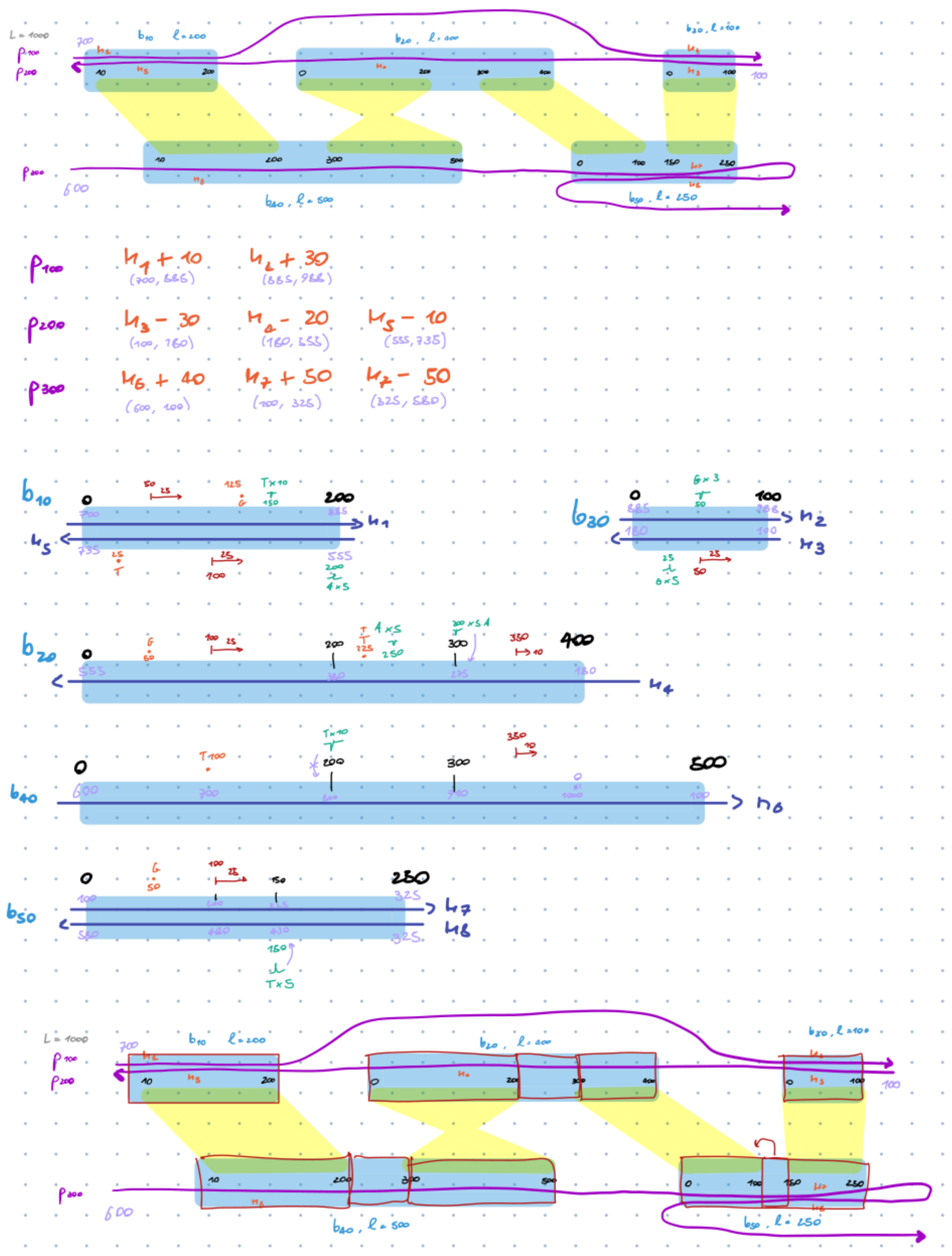


Figure 5: test case for the `split_block` function.



L = 1000
p100
p200
p300
b10, ℓ = 200
b20, ℓ = 400
b30, ℓ = 100
b40, ℓ = 500
b50, ℓ = 250
p100 h1 + 10 (700, 885) h2 + 30 (885, 988)
p200 h3 − 30 (100, 180) h4 − 20 (180, 555) h5 − 10 (555, 735)
p300 h6 + 40 (600, 100) h7 + 50 (100, 325) h8 − 50 (325, 580)
b10
b20
b30
b40
b50
h1
h2
h3
h4
h5
h6
h7
h8

Figure 6: test case for the reweave function.

3 Notes

3.1 New Block and Node IDs assignment

When merging we create new blocks and nodes and we need to assign new unique IDs to them. This process needs to be:

- *deterministic*: executing the build command on the same input file should generate the same output.
- *parallelizable*: the assignment should be robust to parallelization. This can be either parallelization in the tree traversal or in the block splitting and merging.

Currently I do the following:

- new **node ids** are assigned deterministically to nodes as:

$$n_{\text{id}} = \text{hash}(b_{\text{id}}, \pi_{\text{id}}, \text{start}, \text{end})$$

- new **block ids** are assigned differently for *aligned* and *leftover* blocks.
  - the id for *aligned* blocks needs to be the same for the pair of blocks that will be merged. It is therefore assigned to the alignment object by the `assign_new_block_ids` function as:

$$\mu_{\text{id}} = \text{hash}(\text{qry}.b_{\text{id}}, \text{qry.start}, \text{qry.stop}, \text{reff}.b_{\text{id}}, \text{reff.start}, \text{reff.stop})$$

  - the id for *leftover* blocks is assigned at block splitting during interval creation as:

$$\text{new } b_{\text{id}} = \text{hash}(b_{\text{id}}, \text{start}, \text{end})$$

  see the `create_intervals` function.

Should we worry about hash collision? Maybe when adding new entries to the hash tables we can check if the key is already present and throw an error if so.

If we do not plan on parallelizing tree traversal, we could also use a sequential counter to assign new block ids.

3.2 Todo

- ○ Discuss: **block id assignment** through hash of alignment. Better to use a counter? Currently divided in 2: aligned and unaligned blocks.
- ○ graph consistency checks:
  - consecutive node positions in paths
  - node end - start length (with periodic boundary conditions) vs edit lengths.
  - input sequence can be reconstructed exactly from path.
- ○ Future: function to polish alignments.
- ○ Future: reconsensus.
- ○ Future: remove transitive edges for periodic boundary conditions.
- ○ Throw error at hash collision?