insilicoSV

The goal of this short paper is to explain the internal algorithms and workings of *insilicoSV*, a software to design and simulate both novel and known complex SVs. This paper will assume knowledge from the READ.me and will not repeat information already listed. As an overview, the workflow is ordered as such: processing the config file, randomly placing the structural variants (SVs), applying all transformations, and exporting to the FASTA and BEDPE files. Since the procedure to read the config file is standard, the following sections will only focus on the later components.

Terminology

An event, which will be referenced in the below sections and the code, refers to the mapping between every unique symbol in the transformation sequences and the reference genome. An SV with the transformation "ABC" -> "Abcc'c'" therefore has three events. Note that an event does not necessarily need to be associated with one or even any mutations. In the example above, the event "A" has no manipulation done to its reference fragment while the event "C" has one inversion and two duplications.

A block is a grouping of all events between dispersions and plays an important role in the application of transformations. For example, the transformation "AB_C" has two blocks: "AB" and "C". Events form blocks, and blocks form SVs.

Random Placement of SVs

In order to map the symbols to reference fragments, insilicoSV iterates through the SVs and randomly selects a position and a chromosome. For each of the SV's events in the source, we store the reference fragment at the corresponding position and check that 1) its position does not overlap with the intervals of previous SVs and 2) the reference fragment contains less than 5% of "N"s. If any event fails these conditions, the process restarts. A maximum number of tries, the default of which is 100, is set by the user. Following a successful placement, insilicoSV separately checks for symbols present in the target but not the source sequence to account for foreign insertions; it is here that a randomly generated sequence will populate their Event objects. insilicoSV also compiles a list of the "block" start and end positions after a successful run. For instance, a SV "A_BC" -> "a_bc" starting at position 10 with events of length 5 will contribute (10,15), (20, 30) to the list. The ranges of dispersions are not considered as more events may be placed within this space.

While this structure may seem to be an unoptimized way to choose positions as there may be multiple tries, most SVs will only require one try given the size of the reference genome. Furthermore, the objective of this placement is that it is truly *random*, and any attempt to form an optimized algorithm that ensures fitting will inevitably add predictability to the result.

Apply Transformations

The blocks of a SV define the space in which changes occur. insilicoSV goes block-by-block to compile a list of changes formatted as (block_start, block_end, new_frag). It will construct a "new" fragment for each block based on the symbols present in the target sequence. For example, the transformation "ABC" -> "AbCc'" will form an altered sequence with B inverted and C duplicated. Note that events which remain the same, such as A, are *still* included in the new fragment.

The reason why the changes are organized by block rather than by whole SV or individual events is that blocks offer the simplest way to indicate alterations. To use a single tuple to represent the whole SV, which could include dispersions, might result in the overlapping of positions with other SVs in the final list of changes. To use the positions of individual events for each tuple would contribute unnecessary complexity as it would require divy-ing the block up. To illustrate, it is not immediately clear how to divide up the target block for a transformation like "AB" -> "aa'a'b'a'a'ba'". The current strategy uses the least complexity to achieve the desired functionality.

Exporting Process

Writing to Fasta Files

The changes from all SVs are first collected and arranged by chromosome. The export function expects its input to be a list of tuples formatted as (block_start, block_end, new_frag) with no overlap across tuples. After sorting the list, it will remake the whole chromosome by replacing the reference fragment from block_start to block_end with new_frag. More specifically, it reads and writes to the fasta file until block_start and then writes new_frag. Consequently, insertions, where block_start = block_end, will be written right before block_end.

Writing to BEDPE file

The function to export to bedpe traverses through the symbols in the target sequence and detects the basic transform indicated. For example, "A'" means a duplication while "a" means an inversion. Translocations are currently defined as the movement of a symbol across a dispersion to a different block. While going symbol-by-symbol, we also update a curr_pos and curr_chr value that tells the target position (Note: throughout the edits process, no "target" attribute was assigned for events because the target position is immediately implied from the order of symbols in the transformation). If we encounter an original symbol (ex. "A" or "B") that was present in the source, we update the value curr_pos to the end position of the event in the reference. For insertion-like operations such as insertions, translocations, and duplications, we track an "order" index that specifies the order fragments targeting the same position are inserted. Finally, all accumulated data is exported in the format described in the READ.me.