# Supplementary Materials

## 1 Density and recall of a transition-set

The Cloud Permutation Problem and the Clouded Eulerian Path Problem assume that the clouds are correct. However, for the MOCK5 dataset, $58,187$ out of $64,448$ putative clouds in the contracted assembly graph graph are incorrect ($76,654$ out of $77,841$ putative clouds are incorrect for the YEAST dataset). Many incorrect putative clouds are triggered by *false* transitions that do not represent consecutive edges of the genomic cycle in the contracted assembly graph. Below we introduce the concepts of the density and recall of a transition-set and describe four *transition elimination* procedures aimed at reducing the number of false transitions.

Let $CDB$ be a contracted assembly graph and $T$ be a transition-set on its edges. We refer to transitions between consecutive edges in the genomic cycle $Cycle(Genome, CDB)$ as *genomic transitions* and refer to the set of such transitions as $T_{Genome}$. All edges in the set $T \backslash T_{Genome}$ represent *false transitions* (the number of false transitions may vary from 0 to $|E(CDB)|^2 - |E(CDB)|$), where $E(CDB)$ is the edge-set of the graph $CDB$. We define the *transition density* parameter as:

$$density(CDB, T) = \frac{(|T \setminus T_{Genome}|)}{(|E(CDB)|^2 - |E(CDB)|)}$$

The lower is the density, the more information a transition-set contains, ranging from no information (density 1) to the correctly inferred transition-set with density 0 that coincides with the transition-set of the genomic cycle.

Some genome transitions may be missing from the contracted assembly graph, making it impossible to reconstruct the genomic cycle. To analyze the extent of missing transitions, we define the *transition recall* parameter as:

$$recall(CDB, T) = \frac{|T \cap T_{Genome}|}{|T_{Genome}|}$$

Appendices 2 and 3 describe the containment and split indices that we use for eliminating false transitions. Appendices 4 and 6 describe various *transition elimination* procedures that greatly reduce the transition density with minimal decrease of the transition recall.

## 2 Containment index

We refer to the set of barcodes marking an edge $e$ in a graph as the *barcode-set* of an edge and denote this set as $b(e)$. Given edges $e_1$ and $e_2$ , we refer to the set of barcodes marking both $e_1$ and $e_2$ as $b(e_1, e_2)$. We score the similarity between barcode-sets of two edges using the *containment index* $CI$ (Koslicki and Zabeti, 2017):

$$CI(e_1, e_2) = \frac{|b(e_1, e_2)|}{min(|b(e_1)|, |b(e_2)|)}$$

The normalization by $min(|b(e_1)|, |b(e_2)|)$ is important for metagenomics datasets with highly uneven coverage. Our analysis suggests that using $min(|b(e_1)|, |b(e_2)|)$ for normalization makes sense for pairs of long edges (even in case one of the long edges $e_1$ and $e_2$ corresponds to a repeat) but needs to be modified as follows in the case one of two edges is short:

$$CI(e_1, e_2) = \frac{|b(e_1, e_2)|}{max(|b(e_1)|, |b(e_2)|)}$$

Note that the normalization term in the case one of the edges is short differs from the normalization term in the case when both edges are long. We use $max(|b(e_1)|, |b(e_2)|)$ instead of $min(|b(e_1)|, |b(e_2)|)$ for normalization since a short edge often corresponds to a repeat that may accumulate many barcodes that do not belong to the same segment of the genome as the long edge.

We say that long edges $e_1$ and $e_2$ have *similar barcode-sets* if $CI(e_1, e_2)$ exceeds a threshold $CI_{long}$. We say that a short edge $e_1$ and a long edge $e_2$ have similar barcode-sets if $CI(e_1, e_2)$ exceeds a threshold $CI_{short}$. Appendix 5 describes how cloudSPAdes automatically sets the thresholds $CI_{long}$ and $CI_{short}$ depending on the specifics of a dataset.

### Analyzing the containment index.

We say that edges in the contracted assembly graph are *close* if they belong to the same genomic cycle and the genomic distance between them does not exceed a threshold $Distance$. The default value for $Distance$ is the inferred mean length of an SSLR fragment (see Appendix 16). Note that close edges are not necessarily consecutive in the genomic cycle. We say that two edges are *distant* if they are not close. Appendix 5 describes how to infer a sample of close (distant) edges. We used the MOCK5 and YEAST datasets to analyze the distribution of the containment index $CI$ for pairs of close and distant long edges (Figure 2.1). Since distant edges typically share no (or very few) barcodes, the containment index $CI$ is typically low for distant edges and high for close edges.

## 3 Split index

A set of four consecutive edges $(e_1, e_2, e_3, e_4)$ in a genomic cycle is called a *quartet* if edges $e_1$ and $e_2$ have approximately the same length and edges $e_3$ and $e_4$ have approximately the same length. Consider a quartet $(e_1, e_2, e_3, e_4)$ and four barcode-sets $b(e_1, e_3)$, $b(e_1, e_4)$, $b(e_2, e_3)$, and $b(e_2, e_4)$. Since $e_2$ and $e_3$ are the only consecutive edges among edge-pairs $(e_1, e_3)$, $(e_1, e_4)$, $(e_2, e_3)$, and $(e_2, e_4)$, we expect that the size of the barcode-set $b(e_2, e_3)$ exceeds the size of three other barcode-sets, We thus compare the size of the barcode set $b(e_2, e_3)$ with the maximal size of three other barcode-sets $max(|b(e_1, e_3)|, |b(e_1, e_4)|, |b(e_2, e_4)|)$ using the *split index* $SI$:

$$SI(e_1, e_2, e_3, e_4) = \frac{|b(e_2, e_3)|}{max(|b(e_1, e_3)|, |b(e_1, e_4)|, |b(e_2, e_4)|)}$$

We expect that the split index of a quartet significantly exceeds 1.

Since we do not expect to find many quartets in a genomic cycle, we split each edge $e$ into two halves denoted $head(e)$ and $tail(e)$. After this split, every two consecutive edges $e$ and $e'$ in a genomic cycle form a quartet $(head(e), tail(e), head(e'), tail(e'))$ and we define $SI(e, e')$ as $SI(head(e), tail(e), head(e'), tail(e'))$.

### Analyzing the split index.

We say that edges in a genomic cycle are *t-close* if they are separated by $t$ other edges in this cycle (e.g., consecutive edges in a genomic cycle are 0-close).

We used the MOCK5 and YEAST datasets to analyze the distribution of the split index $SI$ for pairs of consecutive long edges and non-consecutive $t$-close long edges (for $1 \leq t \leq 4$). Figure 3.1 shows that split index $SI$ is typically low for non-consecutive edges and high for consecutive edges.

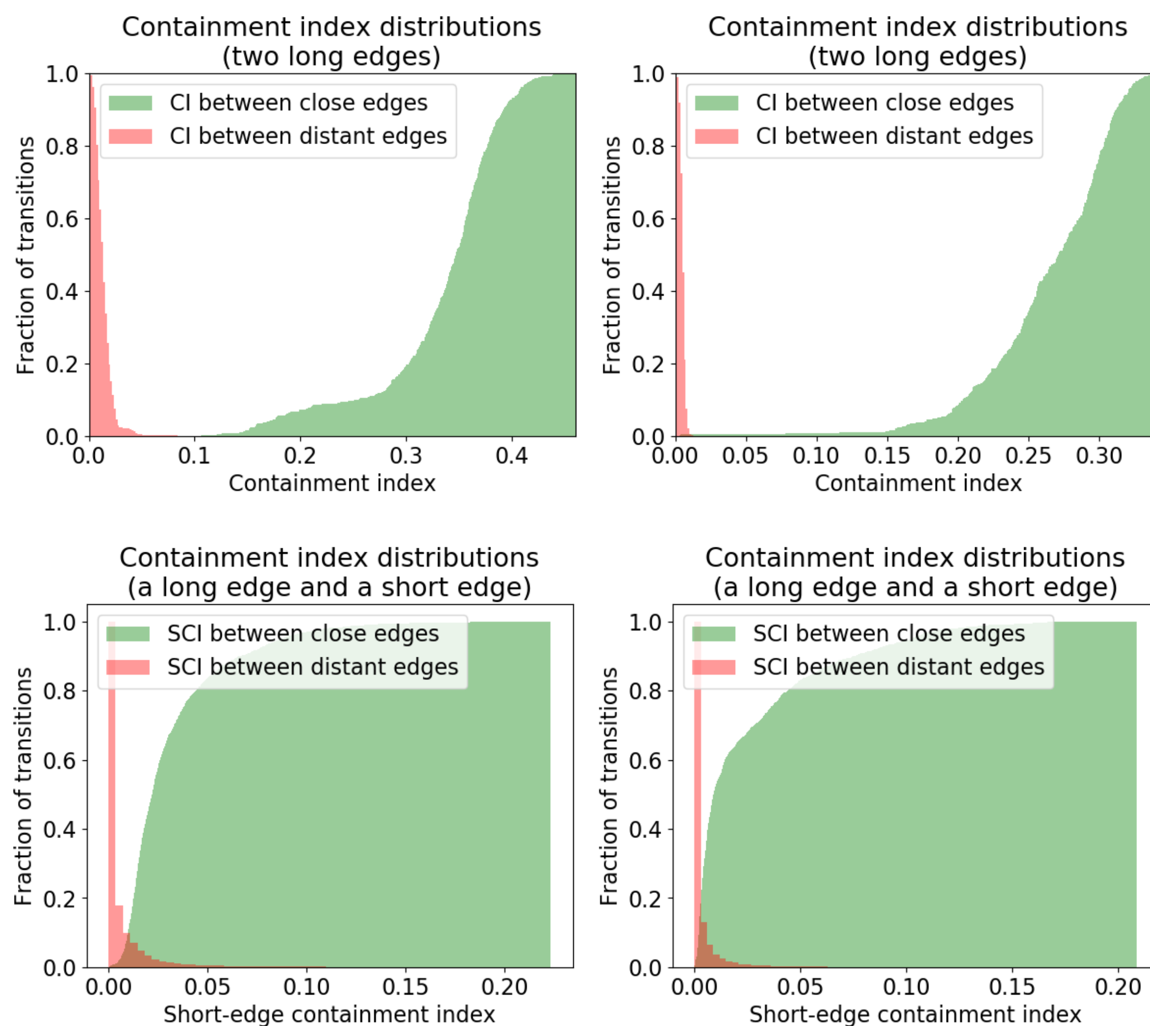


Fig. 2.1: **The containment index between two long edges for MOCK5 (top left), YEAST (top right) datasets in the case of distant (red) and close edges (green). The containment index between a long and a short edge for MOCK5 (bottom left) and YEAST (bottom right) datasets.** The green histogram shows the probability that the containment index $CI$ of two close edges does not exceed the given value. The red histogram shows the probability that the containment index $CI$ of two distant edges exceeds the given value.

## 4 Transition elimination

### Eliminating transitions based on the containment index of long edges.

Given consecutive edges $e_1$ and $e_2$ in a genomic cycle, we expect that their barcode-sets are similar (at least in the case when the genomic distance between these edges is small). We say that two edges in the contracted assembly graph are *linked* if their containment index exceeds the threshold $CI_{long}$ and eliminate all transitions between non-linked edges. Although this procedure filters out many false transitions, some non-consecutive but close edges in the genomic cycle may share many barcodes, resulting in a high containment index between these edges. Below we describe an additional transition elimination procedure to filter out such pairs of edges.

### Eliminating transitions based on the containment index of short-length paths.

Using the containment index, one can check if edges $e_1$ and $e_2$ in the contracted assembly $CDB$ have similar barcode-sets. Note that short edges in the genomic path between $e_1$ and $e_2$ in the assembly graph $DB$ are expected to have barcode-sets similar to barcode-sets of $e_1$ and $e_2$. We now describe how to check whether the barcode set of a short-edge path between $e_1$ and $e_2$ in the assembly graph has a similar barcode-set to both $e_1$ and $e_2$. If such a path does not exist, edges $e_1$ and $e_2$ likely form a false transition that needs to be eliminated.

Ideally, $barcode(F)$ marks all edges in the assembly graph that are traversed by a fragment $F$. In practice, since the coverage of fragments by reads is low, short edges traversed by $F$ are often not marked by $barcode(F)$. However, for each short edge $e$ on a path between long edges $e_1$ and $e_2$, some barcodes in $b(e)$ typically also occur in $b(e_1)$ and/or $b(e_2)$. We say that a short edge and a long edge in the assembly graph are linked if the containment index between these edges exceeds $CI_{short}$ . We say that long edges $e_1$ and $e_2$ are *linked by a short-edge path* if there is a short-edge path in the assembly graph from the start of $e_1$ to the end of $e_2$ that:

- consists only of edges that are linked to both $e_1$ and $e_2$,
- for every edge $e$ of the path, there are at least $PE(e)$ paired-end reads with the right read mapping to $e$ and the left read mapping to the prefix

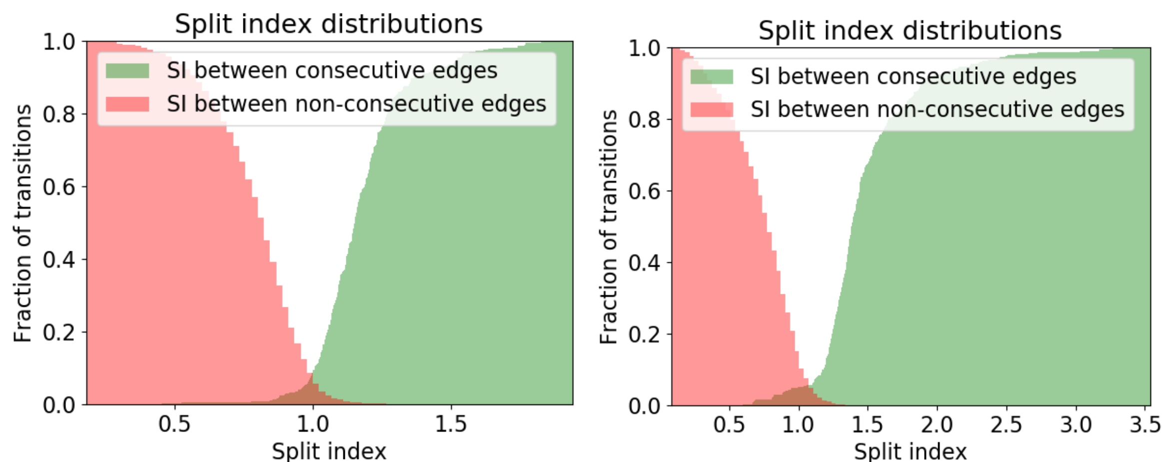


Fig. 3.1: **The split index between two long edges for MOCK5 (left) and YEAST (right) datasets in the case of non-consecutive (red) and consecutive edges (green).** The green histogram shows the probability that the split index $SI$ of two consecutive edges does not exceed the given value. The red histogram shows the probability that the split index $SI$ of two non-consecutive $t$-close ($1 \leq t \leq 4$) edges exceeds the given value.

of the path that precedes $e$. The threshold $PE(e)$ is set by the threshold selection procedure from the exSPAnder algorithm (Prjibelski *et al.*, 2014).

Note that exSPAnder stops the path extension procedure if there exist multiple paths between long edges $e_1$ and $e_2$ that satisfy the condition above. Since we do not aim to reconstruct the correct path between $e_1$ and $e_2$ at this point, we just check if a path that satisfies the conditions above exists. We eliminate all transitions that are not linked by short-edge paths.

### Eliminating transition based on the split index.

Ideally, a unique edge $e$ in the genomic cycle has a single transition to the next edge in this cycle. However, other edges in the contracted assembly graph might also be close to $e$, often triggering multiple transitions from this edge.

Many non-consecutive long edges in a genomic cycle in the assembly graph turn into incident edges in the contracted assembly graph. Such pairs of incident edges in the contracted assembly graph form false transitions that sometimes are not removed by the previously described transition elimination procedures. We found that transitions between $t$-close edges (for small values of $t$) in the genomic cycle are common among false transitions. Another common source of false transition is triggered by complementary edges, i.e., edges with reverse complementary sequence. Since the orientation of reads is unknown, complementary edges have identical barcode-sets. As a result, if an edge $e_1$ forms a genomic transition with an edge $e_2$, it often forms a false transition with the complementary edge of $e_2$.

cloudSPAdes uses the observation that the split index is high for genomic transitions and low for false transitions (even if they correspond to close edges). We thus classify a transition $(e_1, e_2)$ as false if its split index $SI(e_1, e_2)$ is below a threshold $SI_{min}$ (the default value 0.95). It turned out that the transition elimination procedure based on the split index does not eliminate any genomic transitions in the MOCK5 and YEAST datasets.

### Eliminating transitive transitions.

The previously described transition elimination procedures remove most false transitions but still retain a small number of false transitions between $t$-close edges (typically for $t < 5$). We classify a transition between edges $e_1$ and $e_2$ as *transitive* (and eliminate it) if there exists a path (consisting of more than one but less than 5 edges) between the end vertex of $e_1$ and the start vertex of $e_2$ in the contracted assembly graph such that pairs of consecutive edges in this path form transitions from the transition-set.

### Long and ultralong edges.

The results of procedures described above depend heavily on the value of the edge length threshold $LT$. The previously described length threshold selection procedure generates the relatively large threshold $LT^+$. Although the transition elimination procedure applied to edges longer than $LT^+$ eliminates almost all false transitions, it also removes some genomic transitions that correspond to pairs of distant edges. To preserve information about these missing transitions, we introduce another tier of long edges (longer than the threshold $LT$). Transition elimination procedures work under the assumption that long edges are unique. Our analysis have shown that majority of edges longer than $LT = 3,000$ are unique, and transition elimination procedures for $LT = 3,000$ result in near perfect recall for MOCK5 and YEAST datasets. We now refer to edges longer than a threshold $LT^+$ as *ultralong*, to edges longer than a threshold $LT$ but shorter or equal than $LT^+$ as long, and to all remaining edges as short.

### Results of the transition elimination procedure on the MOCK5 dataset.

Tables 4.1 and 4.2 present stage-by-stage results of the transition elimination for the MOCK5 and YEAST datasets. Since the transition elimination procedures based on the split index and transitive transitions result in a low recall for long edges, these steps are only used for ultralong edges. We refer to the transition-set after the final stage of all transition elimination procedures in the contracted assembly graph $CDB$ as $T^*(CDB)$.

## 5 Estimating thresholds for eliminating transitions

The transition elimination procedures are aimed at distinguishing the genomic transitions (consecutive pairs of long edges in the genomic cycle) from false transitions. Thus, it would be useful to infer a subset

| stage | missing genomic transitions | false transitions | genomic transitions | density | recall |
|---|---|---|---|---|---|
| Ultralong edges | | | | | |
| contracted assembly graph | 0 | 90,896 | 416 | 0.52 | 1 |
| elimination based on containment index of long edges | 10 | 1,108 | 406 | 0.009 | 0.976 |
| elimination based on containment index of short-edge paths | 22 | 334 | 394 | 0.004 | 0.947 |
| elimination based on containment index of short-edge paths combined with the exSPAnder-based test | 42 | 265 | 374 | 0.004 | 0.9 |
| elimination based on the split index | 43 | 54 | 373 | 0.002 | 0.896 |
| **elimination of transitive transitions** | **43** | **5** | **373** | **0.002** | **0.896** |
| Long edges | | | | | |
| contracted assembly graph | 0 | 67,970 | 1,138 | 0.049 | 1 |
| elimination based on containment index of long edges | 0 | 11,458 | 1,138 | 0.001 | 1 |
| elimination based on containment index of short-edge paths | 2 | 396 | 1,136 | 0.0002 | 0.998 |
| **elimination based on containment index of short-edge paths combined with the exSPAnder-based test** | **4** | **266** | **1,134** | **0.0002** | **0.996** |
| elimination based on the split index | 199 | 126 | 939 | 0.0001 | 0.825 |
| elimination of transitive transitions | 200 | 64 | 938 | 0.0001 | 0.824 |

Table 4.1. **Results of all transition elimination procedures for ultralong edges (top) and long edges (bottom) for the MOCK5 dataset.** Final results for the transition elimination procedures for ultralong and long edges are highlighted in bold.

| stage | missing genomic transitions | false transitions | genomic transitions | density | recall |
|---|---|---|---|---|---|
| Ultralong edges | | | | | |
| contracted assembly graph | 0 | 134,680 | 414 | 0.489 | 1 |
| elimination based on containment index of long edges | 20 | 500 | 394 | 0.001 | 0.95 |
| elimination based on containment index of short-edge paths | 42 | 28 | 372 | 0.0001 | 0.90 |
| elimination based on containment index of short-edge paths combined with the exSPAnder-based test | 58 | 18 | 356 | 0.0001 | 0.86 |
| elimination based on the split index | 58 | 4 | 356 | 0.00001 | 0.86 |
| **elimination of transitive transitions** | **60** | **4** | **354** | **0.00001** | **0.86** |
| Long edges | | | | | |
| contracted assembly graph | 0 | 208,816 | 794 | 0.25 | 1 |
| elimination based on containment index of long edges | 14 | 774 | 782 | 0.001 | 0.98 |
| elimination based on containment index of short-edge paths | 18 | 110 | 778 | 0.0001 | 0.97 |
| **elimination based on containment index of short-edge paths combined with the exSPAnder-based test** | **23** | **72** | **773** | **0.00008** | **0.96** |
| elimination based on the split index | 268 | 26 | 526 | 0.00003 | 0.66 |
| elimination of transitive transitions | 268 | 15 | 526 | 0.00002 | 0.66 |

Table 4.2. **Results of all transition elimination procedures for ultralong edges (top) and long edges (bottom) for the YEAST dataset.** Final results for the transition elimination procedures for ultralong and long edges are highlighted in bold.

of genomic transitions to estimate the parameters for the transition elimination procedures. Since the genomic transitions are not known, we infer these parameters by sub-partitioning ultralong edges of the assembly graph.

Given an ultralong edge $e$, and parameters $Length_1$, $Length_2$, and $MaxDistance$, we split the prefix of $e$ of length $\lfloor |e| \;/\; Length_1 \rfloor \cdot Length_1$ into the set of segments $Segments(e, Length_1)$ of length $Length_1$. Let $Distances(M, MaxDistance)$ be the sequence of possible distances starting from 0, incrementing by $\lfloor MaxDistance \;/\; M \rfloor$ and ending at $MaxDistance$ ($M = 10$ by default). We define *simulated transitions* as all pairs of substrings $(Segment_1, Segment_2)$ of the sequence of the edge $e$ separated by a distance $Distance \in Distances(M, MaxDistance)$ and satisfying the conditions $Segment_1 \in Segments(e, Length_1)$ and $|Segment_2| = Length_2$ (Figure 5.1).

We refer to the union of all simulated transitions over all ultralong edges as

$$Transitions(Length_1, Length_2, MaxDistance).$$

All simulated transitions correspond to genomic transitions which we aim to identify using the barcode information. Thus, the parameter $MaxDistance$ should not be too large to ensure that there are SSLR fragments overlapping with both $Segment_1$ and $Segment_2$. It also should not be too small, since some consecutive long edges in the genomic cycle might be located far from each other in the genome. We thus select the $Q$th percentile ($Q = 90\%$ by default) of the SSLR fragment length distribution as $MaxDistance$. Appendix 16 shows how to estimate the fragment length distribution.

We use the simulated transitions to set the thresholds $CI_{long}$ and $CI_{short}$. Let $SimulatedScores(Length_1, Length_2, MaxDistance)$ be the set of scores obtained by applying $CI$ to all pairs from $Transitions(Length_1, Length_2, MaxDistance)$. We define the threshold for $CI$ with *confidence* $T$ (0.01 by default) as the $T$th percentile of $SimulatedScores$ and use it in the transition elimination procedures.

We set $Length_1 = Length_2 = LT$ ($Length_1 = LT$ and $Length_2 = 1$) for analyzing the containment index between two long edges (between a short and a long edge).

## 6 Repairing the cloud breaks

While the transition elimination procedures remove many false transitions, they may also remove some genomic transitions. Our analysis revealed that the contracted assembly graphs often have a small number of edges that do not participate in any transitions that start from this edge or end in this edge (*cloud breaks*). Such cloud breaks often occur when there are no fragments covering the entire distance between substrings of the genome that correspond to consecutive edges in $Cycle(Genome, CDB)$. We analyze short-edge paths in the assembly graph to repair cloud breaks in the contracted assembly graph.

An edge in the contracted assembly graph is called a *sink (source)* edge if it does not participate in any transition as its first (second) edge. Our goal is to "repair" the cloud breaks by adding transitions between some sinks and sources. To achieve this goal, we use a variation of the exSPAnder algorithm (Prjibelski *et al.*, 2014). Given a subpath of a genomic cycle in the assembly graph, exSPAnder selects the next edge in the genomic cycle based on additional information such as read-pairs or barcodes.

For a long source edge $e$ in the contracted assembly graph, we iteratively extend $path(e)$ in the assembly graph (the path in the assembly graph that was contracted into the edge $e$) by adding short edges to it with the goal to reach the first vertex of a $path(e')$, where $e'$ represents a long sink edge. If such a vertex is reached, we add the transition $(e, e')$ to the transition-set in the contracted assembly graph.

Given the length threshold $LT_{min}$ (200 nucleotides by default), we say that an edge $next$ is *reachable* from an edge $previous$ if there is a path in the assembly graph from the end of $previous$ to the beginning of $next$ that does not contain edges longer than $LT_{min}$. We ignore short edges shorter than $LT_{min}$ (*ultrashort* edges) since many ultrashort edges represent repeats or are marked by a very few barcodes. We consider all short edges (longer than $LT_{min}$) that are reachable from the end vertex of $path(e)$ and refer to them as *candidate* edges. We then select an *extension edge* from the set of candidate edges and add it to $path(e)$. Below we describe how cloudSPAdes selects the extension edge.

### Barcode-set of a path in the assembly graph.

Given a subpath $path$ of a genomic cycle in the assembly graph, our goal is to find out what barcodes contribute to this subpath and exclude barcodes from other regions of the graph (these barcode may contribute to the repeat edges of $path$). To achieve this goal, we infer barcodes only from unique edges of $path$ and exclude barcodes from its repeat edges.

Below we assume that a subpath $path$ starts from a unique long edge $e$ and use it to infer other unique edges in this path. We classify an edge $e'$ in $path$ as unique if $coverage(e') < c \cdot coverage(e)$, where $coverage$ is the coverage of an the edge $e$ by reads, and $c$ is a constant (1.5 by default). We define the barcode-set $b(path)$ as the set of all barcodes from all unique edges in $path$.

### Selecting an extension edge.

The containment index $CI$ between a long and a short edge reflects our confidence that these edges are close in the genomic cycle. Below we compute the containment index between a path $path(e)$ (rather than a single long edge as before) and each candidate edge (playing the role of a short edge) to find the extension edge.

We say that a (short) candidate edge $e'$ is *valid*, if $CI(path(e), e') \geq CI_{short}$. If there is only one valid candidate edge, we select it as an extension edge. If there are multiple valid candidate edges, we select the candidate edge $winner$ with the largest value of $CI$. If the $CI$ value of $winner$ is at least $r$ times (the default value $r = 2$) larger than the $CI$ value of all other candidates, we select $winner$ as an extension edge. Otherwise, we stop the path extension procedure.

## 7 Filtering clouds in the contracted assembly graph.

A set consisting of one (two) elements is called a *singleton (doubleton)*. All other sets are called *multitons*. We classify a putative cloud as *correct* if it represents a composition of a subpath of the genomic cycle $Cycle(Genome, CDB)$, and *incorrect* otherwise. The Clouded Eulerian Path problem assumes that all input clouds are correct. However, $1,030$ out of $6,632$ putative clouds in the MOCK5 dataset and 270 out of $2,265$ putative clouds in the YEAST dataset constructed using the contracted assembly graph $CDB = DB_{LT}$ and the transition-set $T = T^*(CDB)$ are incorrect (note the difference between the numbers here and numbers in Appendix 1, where clouds are constructed using $CDB$ only). We thus aim to remove all (or nearly all) false clouds before solving the CEP problem.

We say that clouds $c_1$ and $c_2$ *clash* if $c_1$ crosses $c_2$ and there is no $T$-compatible path in $CDB$ that conforms with both $c_1$ and $c_2$. If the cloud-set $C = Clouds(CDB, T, Reads)$ contains clashing clouds, then there is no $T$-compatible clouded Eulerian path in $CDB$ that conforms with $C$. Thus, our goal is to remove some clouds so that there is no clashing clouds left. Specifically, we want to remove all clouds from $C$ that clash with correct clouds.

Given a subset of edges $c$ in the contracted assembly graph, we define $b(c)$ as the set of all barcodes marking edges of $c$. Let $c_{true}$ be a correct cloud that clashes with a false cloud $c_{false}$. Since $c_{true}$ corresponds to a subpath in the genomic cycle and $c_{false}$ does not, the set $c_{true} \cap c_{false}$ usually shares more barcodes with $c_{true} \setminus c_{false}$ than with $c_{false} \setminus c_{true}$. We use the score function $CI$ from Appendix 2 to decide which clashing cloud in the pair $(c_1, c_2)$ is correct

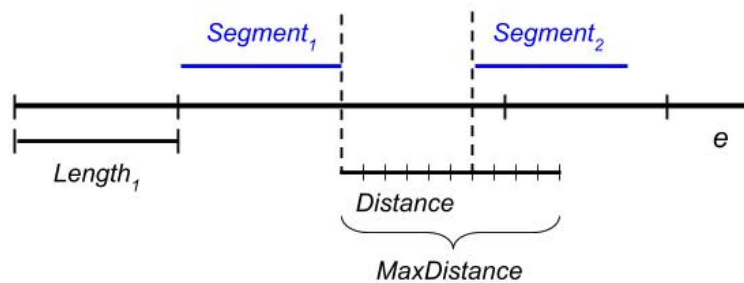


Fig. 5.1: **Generating simulated transitions.** Prefix of an ultralong edge $e$ is split into segments of length $Length_1$. Two blue segments represent a simulated transition.

$$clash(c_1, c_2) = \frac{CI(c_1 \cap c_2, c_1 \setminus c_2)}{CI(c_1 \cap c_2, c_2 \setminus c_1)}$$

If $clash(c_1, c_2)$ exceeds a threshold $r$ (2 by default), we assume that the cloud $c_1$ is correct and remove the cloud $c_2$ from the cloud-set $C$. If $clash(c_1, c_2) < \frac{1}{r}$, we remove the cloud $c_1$ from $C$. If $\frac{1}{r} \leq clash(c_1, c_2) \leq r$, we remove both $c_1$ and $c_2$ from $C$. This procedure is applied to all pairs of clashing clouds in the cloud-set $C$.

Our analysis of the MOCK5 and YEAST datasets revealed that the most common example of clashing clouds are doubletons sharing an edge. If we successfully resolve all clashing doubletons, finding the clouded Eulerian path turns into a trivial problem (since we obtain all transitions of the genomic cycle as the correct doubletons). This is usually the case when the mean fragment length is small compared to the long edge threshold $LT$. However, when the mean fragment length is large, it becomes harder to resolve clashing doubletons than clashes that involve multitons. For example, for a subpath $e_1, e_2, e_3, e_4$ of a genomic cycle, $clash(\{e_1, e_2, e_3\}, \{e_1, e_2, e_4\})$ is usually larger than $clash(\{e_2, e_3\}, \{e_2, e_4\})$, since there are fewer barcodes that mark only $e_2$ and $e_3$ compared to the barcodes that mark $e_2$ and $e_4$, but do not mark $e_3$. In this case, we use the **CloudedPath** algorithm (Appendix 13) to find the clouded Eulerian path using larger clouds.

## 8 Fragmentation of the contracted assembly graph

Although the transition elimination procedures greatly reduce the number of false transitions, they also remove some correct transitions (Table 4.1) and thus break genomic cycles in the contracted assembly graph into multiple paths. Since the Clouded Eulerian Path Problem can not be applied to the entire contracted assembly graph in such cases, we partition it into smaller subgraphs and solve a separate assembly problem in each subgraph, localizing the effect of false and missing transitions in the graph. Below we describe the graph fragmentation approach and apply it for reconstructing the genomic cycle.

### Contracted assembly graph on ultralong edges.

We observed that inferring the sequence of long edges in a putative cloud is usually an easier task than restoring the paths of short edges between long edges. cloudSPAdes thus constructs the contracted assembly graph based on ultralong edges (longer than a length threshold $LT^+$) so that most clouds contain only 2-4 ultralong edges. After reconstructing the order of these ultralong edges in the genomic cycle, we fragment the contracted assembly graph $DB_{LT}$ into subgraphs between consecutive ultralong edges and split the genomic cycle reconstruction into smaller subproblems. Below we describe an algorithm for reconstructing the order of ultralong edges in $DB_{LT^+}$ and the algorithm for reconstructing a path between consecutive ultralong edges in $DB_{LT}$.

### Filtering false clouds.

Let $Component$ be a weakly connected component of the contracted assembly graph $CDB = DB_{LT^+}$. Our goal is to extract the subpath of the genomic cycle $Cycle(Genome, CDB)$ from $Component$ by solving the CEP problem. Let $T = T^*(Component)$ be a transition-set, and $Clouds(Component, T, Reads)$ be a set of putative clouds obtained from $Component$. Even though the fraction of false putative clouds in the contracted assembly graph $CDB$ is low (4% for the MOCK5 dataset), we cannot apply the **CloudedPath** algorithm to $Component$ since most components in $CDB$ still contain false clouds. We thus apply the cloud filtering procedure described in Appendix 7 to eliminate false clouds from $Clouds(Component, T, Reads)$ and refer to the resulting cloud-set as $Clouds^*(Component, T, Reads)$.

We estimate the effectiveness of the false transition elimination and putative cloud filtering procedures using the MOCK5 and YEAST datasets with known genomes. We classify $Component$ as *correct* if it contains a subpath of $Cycle(Genome, CDB)$ as an Eulerian path and if this subpath conforms with all clouds from $Clouds^*(Component, T, Reads)$. For the MOCK5 dataset, 22 out of 26 *nontrivial* components (components with more than two vertices) in the contracted assembly graph $DB_{LT^+}$ are correct, and for the YEAST dataset, 92 out of 98 nontrivial components are correct. We thus can apply an algorithm for solving the Clouded Eulerian Path problem to a majority of weakly connected components in the contracted assembly graph.

By applying the **CloudedPath** algorithm from Appendix 13 to weakly connected components and filtering putative clouds, we obtain the set of subpaths $Subpaths(DB_{LT^+})$ of $Cycle(Genome, DB_{LT^+})$. We refer to the set of pairs of consecutive ultralong edges from $Subpaths(DB_{LT^+})$ as $Pairs(DB_{LT^+})$.

### Filling the gap between consecutive ultralong edges.

Let $(e_1, e_2)$ be a pair of consecutive ultralong edges from $Pairs(DB_{LT^+})$. To fill the gap between these edges in the contracted assembly graph $CDB = DB_{LT}$, we analyze all edges that have a large containment index with both $e_1$ and $e_2$, i.e., all edges $e$ with $CI(e_1, e) > CI_{long}$ and $CI(e, e_2) > CI_{long}$. The subgraph $CDB[e_1, e_2]$ is constructed as the induced subgraph on these edges.
Let $Path[e_1, e_2]$ be a subpath of the genomic cycle $Cycle(Genome, CDB)$ that starts from $e_1$ and ends in $e_2$. We reconstruct $Path[e_1, e_2]$ by applying the **CloudedPath** algorithm from Appendix 13 to $CDB[e_1, e_2]$ and the set of clouds $Clouds^*(CDB[e_1, e_2], Reads)$. This algorithm results in a subpath of $Cycle(Genome, CDB)$ for every pair of consecutive ultralong edges. We refer to the set of all such reconstructed subpaths as $Paths(DB, LT)$.

## 9 Combining information about clouds with information about read-pairs

Although the contracted assembly graph contributes to constructing scaffolds using clouds, it does not utilize information about the read-pairs generated by the SSLR technology. To combine both types of information (clouds and read-pairs), we use long edge sequences $Paths(DB, LT)$ to scaffold metaSPAdes contigs $Contigs(Reads, k)$ obtained from the assembly graph $DB(Reads, k)$ and read-pairs $Reads$ using the exSPAnder algorithm (Prjibelski *et al.*, 2014). Every contig $contig$ from $Contigs(Reads, k)$ represents a path $path(contig)$ in the assembly graph $DB(Reads, k)$. Let $first(contig, LT)$ and $last(contig, LT)$ be the first and last long edges of $path(contig)$, respectively.

Let $Pairs(DB, LT)$ be the set of pairs of consecutive edges in $Paths(DB, LT)$. We merge a pair of contigs $(contig_1, contig_2)$ from $Contigs$ if the pair of long edges $(last(contig_1, LT), first(contig_2, LT))$ belongs to $Pairs(DB, LT)$, resulting in the set of (possibly gapped) scaffolds $Scaffolds(Contigs, Paths, LT)$. Afterwards, cloudSPAdes constructs the *scaffold graph* $DB_{Scaffolds}$ by merging every path from $Scaffolds = Scaffolds(Contigs, Paths, LT)$ into a single *scaffold edge* (which might contain unknown nucleotides if some gaps were not closed by the gap closing procedure). Most scaffold edges in $DB_{Scaffolds}$ are formed by multiple long edges in $DB$: the scaffold edges in the MOCK5 dataset (mean length 154 kb) are much longer than long edges (mean length 29 kb). Also, the gaps between scaffold edges in $DB_{Scaffolds}$ are smaller than the gaps between long edges in $DB$ (mean gap is 885 bp for scaffold edges vs 7 kb for long edges in the MOCK5

dataset), making it easier to connect the scaffold edges using the read-pairs or clouds.

To utilize the advantages of the scaffold edges over the long edges of the assembly graph, cloudSPAdes constructs the contracted assembly graph $CDB = CDB(DB_{Scaffolds}, LT)$ and a transition-set $T = T^*(CDB)$. Let $outdegree_T(e)$ and $indegree_T(e)$ be the number of transitions starting from $e$ and ending at $e$, respectively. cloudSPAdes extracts *reliable transitions* from $T$, i.e., pairs of scaffold edges $(e_1, e_2)$ such that $indegree_T(e_2) = outdegree_T(e_1) = 1$, and merges scaffolds corresponding to $e_1$ and $e_2$ into a single scaffold. We refer to the algorithm that generates the resulting scaffolds (and generates the final cloudSPAdes output) as $\mathbf{MergeScaffolds}(Scaffolds, LT)$.

## 10 Closing gaps between long edges

When cloudSPAdes merges contigs using the long-edge paths in the contracted assembly graph, it typically inserts a gap between merged contigs that separates consecutive edges in these contigs. It further attempts to close this gap by reconstructing a segment of the genomic cycle in the assembly graph between these edges. Below we describe how cloudSPAdes combines read-pairs and clouds to reconstruct segment $Segment$ of the genomic path between consecutive long edges $e_1$ and $e_2$ in the scaffold. Similarly to the algorithm described in Appendix 6, we use exSPAnder to reconstruct $Segment$ starting from $e_1$ and iteratively extending it (by adding short edges) with the goal to reach $e_2$.

We say that a short edge $e$ is *supported* by $e_1$ and $e_2$, if $e$ is reachable from $e_1$, $e_2$ is reachable from $e$, $CI(e_1, e) > CI_{short}$, and $CI(e_2, e)) > CI_{short}$. Supported edges are likely to be located between $e_1$ and $e_2$ in the genomic cycle.

cloudSPAdes constructs the set of supported edges and uses them to improve the results of exSPAnder. At each iteration, we filter a set of candidates (edges starting at the last vertex of $Segment$) by discarding all edges that are neither supported by $e_1$ or $e_2$ nor are a part of a path that leads to a supported edge. The described filtering resolves many situations where it was difficult to reliably select the next edge due to a long repeats or a complex graph structure (Figure 10.1).

In the case when the described algorithm fails to reconstruct a path from $e_1$ to $e_2$, cloudSPAdes attempts to reconstruct the reverse path from $e_2$ to $e_1$. Since the path extension procedure is asymmetric, in some cases the reverse search closes the gap when the direct search fails to close it. We refer to the algorithm that closes gaps in a given $Scaffolds$ as $\mathbf{CloseGaps}(Scaffolds, Reads)$.

## 11 Solving the Cloud Permutation Problem

Below we define a condition on a cloud-set that guarantees that the Cloud Permutation Problem has a unique solution and show how to effectively find the solution if the condition holds.

A set consisting of one (two) elements is called a *singleton (doubleton)*. All other sets are called *multitons*. A cloud is proper if it is a proper subset of $char(C)$. A proper subset of a set (cloud) is *non-trivial subset (non-trivial cloud)* if it is not a singleton. We say that two sets *overlap* if they share at least one element. We say that sets $c_1$ and $c_2$ *cross* ($c_1 \frown c_2$) iff $c_1$ and $c_2$ overlap, $c_1 \not\subseteq c_2$, and $c_2 \not\subseteq c_1$. A cloud-set $C$ *crosses* a subset $s$ of $char(C)$ if it contains a cloud that crosses $s$. A set of clouds $C$ is *complete* if it crosses each non-trivial subset of $char(C)$. For the sake of convenience, we consider cloud-sets that contain all singletons of $char(C)$, and a cloud that consists of the entire set $char(C)$. Also we will assume that $char(C)$ consists of at least two elements.

We denote the reverse string of a permutation $G$ as $\bar{G}$. Given a substring $s$ of a permutation $G$, its *reversal* is a rearrangement of symbols of this susbstring (all other symbols of $G$ do not change their positions) that substitutes the $i$-th symbol of $s$ by its $(|s| + 1 - i)$-th symbol (for all symbol in the substring). For example, the reversal of a substring $bcd$ in $abcdef$ results in $adcbef$.

We say that a cloud crosses a subset of a permutation if it crosses a set formed by the elements of this subset.

Lemma 1. *Let $G$ be a permutation that conforms with a cloud-set $C$. If $C$ does not cross a nontrivial subset $s$ of $char(C)$ then there is a permutation $G'$ different from $G$ and $\bar{G}$ that also conforms with $C$.*

Proof. If the subset $s$ does not contain the first element of the permutation $G$, consider a proper substring $s'$ that starts at the first element of $s$ in $G$ and ends at the last element of $s$ in $G$. For example, if $G = abcdef$ and $s = \{b, d\}$ then $s' = bcd$. Below we show that the $s'$-reversal of $G$ conforms with $C$. Indeed, since $C$ does not cross the subset $s$, each cloud in $C$ either (i) does not overlap with $s'$, or (ii) contains $s'$, or (iii) is contained in $s'$.

In the case (i), a cloud $c \in C$ corresponds to a substring in $G$ that does not include symbols from $s'$, This substring is also present in the $s'$-reversal of $G$ since this reversal only affect symbols from $s'$. Therefore, the cloud $c$ conforms with the $s'$-reversal of $G$.

In the case (ii), a cloud corresponds to a substring in $G$ that includes all symbols of $s'$, Thus, this cloud also corresponds to a substring of the $s'$-reversal of $G$.

In the case (iii), a cloud corresponds to a substring in $G$ formed by symbols in $s'$. Thus, the same symbols form a substring of the $s'$-reversal of $G$.

If the subset $s$ contains the first element of the permutation $G$, let $s_L$ be the maximal prefix in $G$ that is contained in $s$. Since $s$ is a proper subset, $s_L \neq G$, and there is a substring $s_R$ in $G$ such that $G = s_L s_R$. Since $C$ does not cross $s$, every proper cloud in $C$ is contained in either $s_L$ or $s_R$. Thus, the string $s_R s_L$ conforms with $C$. □

Note that if a permutation $G$ conforms with the overlapping clouds $c_1$ and $c_2$, then it also conforms with the sets $c_1 \cup c_2$ and $c_1 \cap c_2$. Moreover, if $c_1$ crosses $c_2$, then $G$ also conforms $c_1 \setminus c_2$ and $c_2 \setminus c_1$. Let $Expansion(C)$ be an expanded set of clouds constructed as a closure of $C$ under these four operations. Clearly, a permutation conforms with $C$ iff it conforms with $Expansion(C)$. Also a set $s$ crosses a cloud-set $C$ iff $s$ crosses $Expansion(C)$ since for any clouds $c_1, c_2$ that do not cross $s$ there union, intersection and differences (in case $c_1$ crosses $c_2$) also do not cross $s$.

We use the notation $v \prec w$ to indicate that a set $v$ is a proper subset of $w$. Given a cloud-set $C$, we construct a directed acyclic *cloud-graph* $Graph(C)$ where each vertex corresponds to a cloud in $Expansion(C)$ and vertices $v$ and $w$ are connected by an edge iff $v \prec w$. We label an edge $(v, w)$ of the cloud-graph by the set of symbols of $w$ that do not occur in $v$. Given a cloud-set $C$ and a pair of clouds $v \prec w$, we define $div(v, w)$ as the number of vertices in a longest path in $Graph(C)$ starting at $v$ and ending at $w$.

Lemma 2. *If $a$ is a singleton in a complete cloud-set $C$ then $div(a, c) = |c|$ for each cloud $c \in C$ that contains $a$.*

Proof. If a longest path $P$ from $a$ to $c$ in the cloud-graph $Graph(C)$ is shorter than $|c|$, then at least one of its edges $(v, w)$ is labeled by a set with multiple symbols, i.e., by a non-trivial set $s$. Since the cloud-set $C$ is complete, there is a cloud $b \in C$ crossing $s$.

If $b$ overlaps with the cloud $v$ then the set $u = v \cup (w \cap b)$ belongs to $Expansion(C)$. If $b$ does not overlap with $v$ then the set $u = w \setminus b$ belongs to $Expansion(C)$. In either case, $v \prec u \prec w$.

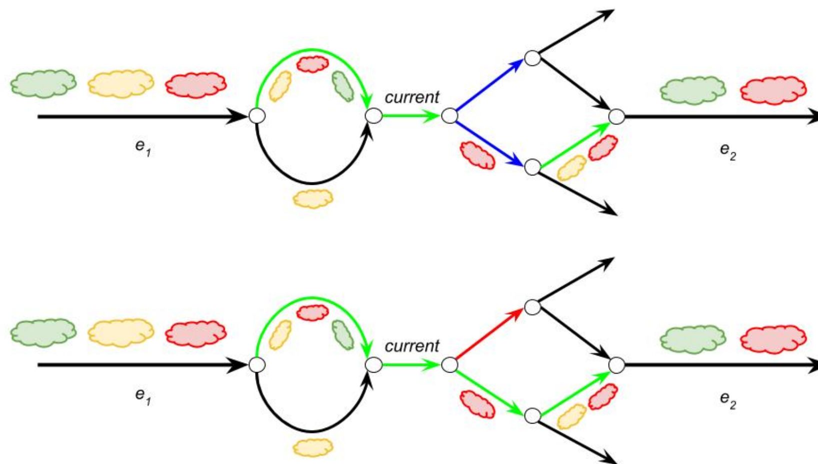


Fig. 10.1: **Gap closing using clouds. (Top)** Subgraph between two long unique edges $e_1$ and $e_2$. The path extension procedure attempts to extend the path by finding an edge that follows the edge $current$. Two candidates are highlighted in blue. Edges that are supported by clouds are highlighted in green. **(Bottom)** Since only one blue edge can be extended using green edges supported by clouds, we add it to the path since the supported edge is reachable from it. The other candidate (highlighted in red) is discarded.

Therefore, inserting $u$ between $v$ and $w$ in the path $P$ increases its length, a contradiction to the fact that $P$ is a longest path from $a$ to $c$. □

A permutation $G$ conforming with a cloud-set $C$ is *unique* if $G$ and $\bar{G}$ are the only permutations that conform $C$.

Theorem. *Let $G$ be a permutation that conforms with a cloud-set $C$. Then $G$ is unique iff $C$ is complete.*

Proof. If $G$ is a unique permutation that conforms with an incomplete cloud-set $C$ then there exists a non-trivial set $s$ that does not cross $C$. However, Lemma 1 implies that there is a permutation different from $G$ and $\bar{G}$ that also conforms with $C$, a contradiction to the fact that $G$ is unique.

If a complete cloud-set $C$ conforms with a permutation $G = g_1 \dots g_N$ then $G$ also conforms with a cloud-set $Expansion(C)$. Lemma 2 implies that there exists an $N$-vertex path in the cloud-graph $Graph(C)$ starting at $g_1$ and passing through vertices:

$$\{g_1\} \to \{g_1, g_2\} \to \dots \to \{g_1, \dots g_N\}$$

i.e., each prefix of $G$ forms in a cloud in $Expansion(C)$. Applying the same argument to the reverse permutation of $G$, each suffix of $G$ forms a cloud in $Expansion(C)$. Each substring of $G$ can be represented as an intersection of a prefix and a suffix of $G$. Therefore, each substring of $G$ forms a cloud in $Expansion(C)$. In particular, $Expansion(C)$ contains all 2-element substrings of $G$ as doubletons, implying that $G$ and $\bar{G}$ are the only strings that conform with the cloud-set $C$. □

Two cloud-sets $C_1$ and $C_2$ are *equivalent* if $Expansion(C_1) = Expansion(C_2)$. Given a cloud-set $C$, below we show how to derive the set of doubletons in $Expansion(C)$ by constructing a series of equivalent but “simpler” cloud-sets to eventually arrive to a cloud-set consisting of doubletons only. Specifically, given a multiton $c$ in a complete cloud-set $C$, we will construct an equivalent cloud-set by removing $c$ from $C$ and substituting it by at two smaller clouds. Iterating this process, we will arrive at a cloud-set where each non-trivial cloud is a doubleton.

We say that a cloud-set is *pseudo-complete* if every nontrivial cloud $c$ from $Expansion(C)$ crosses $C$.

Lemma 3. *For each proper multiton cloud $c$ in a pseudo-complete cloud-set $C$, there exist crossing clouds $c_1$ and $c_2$ in $Expansion(C)$ such that $c = c_1 \cup c_2$. The cloud-set $C'$ where $c$ is replaced by $c_1$ and $c_2$ is equivalent to $C$.*

Proof. Since the cloud-set $C$ is pseudo-complete, there exists a cloud $c' \in C$ crossing the multiton cloud $c$. Therefore, both $c \cap c'$ and $c \setminus c'$ belong to $Expansion(C)$. Since $c$ has more than two elements, one of these sets (denoted $c_1$) has at least two elements. Let $c_2 = c \setminus c_1$. Since $C$ is pseudo-complete, it contains a cloud $c_3$ crossing $c_1$. The union of $c_1$ and $c_2$ equals $c$ however these two sets do not overlap. We will now use $c_3$ to find two crossing sets whose union is also $c$.

Since $c_3$ crosses $c_1$, it can not be a subset of $c_2$. Thus, it either crosses $c_2$ (i), or contains $c_2$ (ii), or does not overlap with $c_2$ (iii).

In the case (i), clouds $c_1$, $c_3 \cup c_2$ cross and their union is $c$.

In the case (ii), clouds $c_1$, $c_3 \cap c$ cross and their union forms $c$.

In the case (iii), clouds $c_1$ and $c \setminus c_3$ cross and their union forms $c$.

Finally, replacing a cloud $c$ with two crossing clouds from $Expansion(C)$ whose union is $c$ results in an equivalent cloud-set since $c$ still belongs to the expansion of the modified cloud-set. □

Lemma above leads to an efficient algorithm for constructing all doubletons in the expansion of a pseudo-complete cloud-set (Figure 11.1). However this algorithm does not directly generalize to incomplete cloud-sets that arise in practice. Appendix 12 describes how to address this complication.

We will conclude this section by stating the relation between complete and pseudo-complete cloud-sets. A cloud-set is *non-trivial* if it contains a non-trivial cloud and *trivial* otherwise (a trivial cloud-set consists only from singletons and $char(C)$). A cloud-set is *viable* if there exists a permutation conforming with this cloud-set.

Lemma 4. *A viable non-trivial cloud-set is complete iff it is pseudo-complete.*

Proof. Let $C$ be a viable non-trivial cloud-set and $G$ be a permutation that conforms with $C$. It is easy to see that if $C$ is complete then it is pseudo-complete. We will now prove that if it pseudo-complete then it is complete.

Given a cloud $c$ in $C$, we define $G(c)$ as a substring of $G$ with composition equal to $c$. We say that a cloud $c$ is *complete* if $Expansion(C)$ contains compositions of all 2-element substrings of $G(c)$. We will prove that if $C$ is pseudo-complete then $char(C)$ is a complete cloud and thus $C$ is a complete cloud-set.

Let $c$ be a smallest cloud in $Expansion(C)$ that is not complete. Since all singletons and doubletons in $Expansion(C)$ are complete then $c$ is a multiton. If $c$ is a proper multiton then, according to lemma 3, it can be represented as as a union of two crossing subclouds $c_1$ and $c_2$. Since both $c_1$ and $c_2$ are smaller than $c$, they are both complete, which implies that $c$ is complete since each 2-element substring from $c$ belongs to at least one of the clouds $c_1$ and $c_2$.

If $c$ is not a proper multiton (i.e., $c = char(C)$), we consider a cloud $c_1$ that is a maximal proper cloud in $Expansion(C)$. Since $C$ is not trivial, $c_1$ is not a singleton and thus there is a cloud $c_2 \in C$ that crosses $c_1$. Since $c_1 \cup c_2$ belongs to $Expansion(C)$ and contains $c_1$ , then $c_1 \cup c_2 = char(C)$. Thus, cloud $c$ is a union of two smaller (and thus complete) crossing clouds. It implies that $c = char(C)$ is complete and $C$ is complete. □

**Figure 11.1** Pseudocode of the $\mathbf{FindCloudSuperstring}$ algorithm that solves the Cloud Permutation Problem for cloud-set $C$. $\mathbf{SplitCloud}(c)$ represents a cloud $c$ as a union of two smaller clouds (as described in lemma 3) and returns these two clouds. $\mathbf{IsSimplePath}(Graph)$ checks if $Graph$ is a simple path and $\mathbf{SimplePath}(Graph)$ returns a string representing this path.

```
1: procedure FindCloudSuperstring(C)
2:   while C contains multitons do
3:     c ← a largest cloud in C
4:     c1, c2 ← SplitCloud(c)
5:     remove c from C
6:     add c1, c2 to C
7:   Graph ← empty graph on the vertex-set Char(C)
8:   for all doubletons (v1, v2) ∈ C do
9:     add edge (v1, v2) to Graph
10:  if IsSimplePath(Graph) then
11:    return SimplePath(Graph)
12:  else
13:    return ∅
```

## 12 Analyzing incomplete cloud-sets

Although the Cloud Permutation Problem has a unique solution when the cloud-set is complete, the SSLRs often define incomplete cloud-sets, making it difficult to reconstruct the genomic cycle in the contracted assembly graph using barcodes marking long edges. Below we introduce the concept of block trees that effectively describe $Expansion(C)$ for an incomplete cloud-set $C$.

### Partinioning a cloud-set into blocks.

We define a *cell* of a cloud-set $C$ as a proper cloud in $Expansion(C)$ that does not cross $C$. Note that cells also do not cross any clouds from $Expansion(C)$ since crossing $C$ is the same as crossing $Expansion(C)$. Maximal cells of a cloud-set $C$ are called *blocks*.

Lemma 5. *Blocks of a cloud-set C form a partition of $char(C)$, i.e., they contain all elements of $char(C)$ and do not overlap.*

Proof. Since each singleton is a cell, each element of $char(C)$ lies within a block. Therefore, blocks contain all elements of $char(C)$.

To conclude the proof we will to show that if a block $b_1$ overlaps a block $b_2$ then $b_1 = b_2$. Since blocks are cells and cells do not cross any other clouds, $b_1$ and $b_2$ do not cross each other Thus either $b_1 \subset b_2$ or $b_2 \subset b_1$. However, since $b_1$ and $b_2$ are maximal cells, $b_1 = b_2$. □

We refer to the set of all blocks in a cloud-set $C$ as the *block partition* of $C$.

Given a cell $s$ and a cloud $c$, we define the *s-glued cloud* $c^s$ as a cloud where all elements of $s$ are glued together, i.e., substituted by a single character that we denote as $s^*$ (if $s$ does not overlap $c$, $c^s = c$). Given a cell $s$ in a cloud-set $C$, we define its *s-glued cloud-set* $C^s$ as a cloud-set where all elements of $s$ are glued together. $C^s$ is a cloud-set in a reduced alphabet where all characters from $s$ (in each cloud that contains characters from $s$) are substituted by a single character $s^*$. The following two lemmas describe the relation between expansions and cells of $C$ and $C^s$.

Lemma 6. *Let s be a cell in a cloud-set. If clouds $c_1$ and $c_2$ overlap then $(c_1 \cup c_2)^s = c_1^s \cup c_2^s$ and $(c_1 \cap c_2)^s = c_1^s \cap c_2^s$. If additionally $c_1 \frown c_2$ then $(c_1 \setminus c_2)^s = c_1^s \setminus c_2^s$*

Proof. We will prove this lemma for only for set intersection since for both other operations the proof is very similar.

Since $s$ is a cell, $c_1$ and $c_2$ do not cross $s$. If $c_1$ or $c_2$ is a subset of $s$ then $c_1 \cap c_2 \subset s$ and thus $(c_1 \cap c_2)^s = \{s^*\} = c_1^s \cap c_2^s$. Thus both $c_1$ and $c_2$ either contain or do not intersect $s$. If both $c_1$ and $c_2$ contain $s$ then $(c_1 \cap c_2)^s = (((c_1 \setminus s) \cap (c_2 \setminus s)) \cup s)^s = ((c_1 \setminus s) \cap (c_2 \setminus s)) \cup \{s^*\} = (c_1^s \setminus \{s^*\}) \cap (c_2^s \setminus)) \cup \{s^*\} = c_1^s \cap c_2^s$. If at least one of $c_1$ and $c_2$ does not intersect $s$ then $(c_1 \cap c_2)$ does not intersect $s$ and $(c_1 \cap c_2)^s = c_1 \cap c_2 = c_1^s \cap c_2^s$.

Lemma 7. *Let s be a cell of cloud-set C and let $c \in Expansion(C)$ be a cloud that is not a proper subset of s. Then c is a cell in C iff $c^s$ is a cell in $C^s$.*

Proof. Let $c$ be a cell in $C$ and $c_1 \in C$ be any cloud from $C$. If $c$ overlaps $c_2$ then by lemma 6 we have $(c \cup c_1)^s = c^s \cup c_1^s$ and $(c \cap c_1)^s = c^s \cap c_1^s$. Since $c$ is a cell, either $c \subset c_1$ or $c_1 \subset c$. Thus either $c^s \subset c_1^s$ or $c_1^s \subset c^s$ for any cloud $c_1^s \in C^s$ and $c^s$ is a cell in $C^s$.

If $c$ does not overlap $c_1$ then $c^s$ can overlap $c_1^s$ only if both $c$ and $c^s$ overlap $s$. In this case since $s$ is a cell, $s$ is a subset of both $c$ and $c_1$ which is a contradiction with the assumption that $c$ does not overlap $c_1$. Thus for any cloud $c_1^s \in C^s$, $c_1$ does not overlap $c^s$ and thus $c$ is a cell.

Conversely, if $c$ is not a cell then there is a cloud $c_1 \in C$ that crosses $c$. By lemma 6 we have $c^s \setminus c_1^s = (c \setminus c_1)^s \neq \varnothing$, $c_1^s \setminus c^s = (c_1 \setminus c)^s \neq \varnothing$ and $c^s \cap c_1^s = (c \cap c_1)^s \neq \varnothing$. Consequently, $c^s$ crosses $c_1^s$ and thus $c^s$ is not a cell. □

Lemma 8. *Let s be a cell in cloud-set C and b be a block in C. Then $b^s$ is a block in $C^s$.*

Proof. Since $b$ is a block, it is not a proper subset of the cell $s$, and thus, according to lemma 7, $b^s$ is a cell in $C^s$. If $b^s$ is not a block then there is a block $b_1$ in $C^s$ that contains $b^s$. Lemma 6 implies that $Expansion(C)^s = Expansion(C^s)$ and thus there is a cloud $b_2 \in Expansion(C)$ such that $b_1 = b_2^s$. Since $b_2$ contains $b$, $b_2$ can not be a proper subset of $s$. Thus, by lemma 7 $b_2$ is a cell that contains $b$ as a proper subset, that is a contradiction since $b$ is a block. □

The block partition of a cloud set can be constructed by iterating through all clouds in $Expansion(C)$ in order from the largest to the

smallest as described in the pseudocode in Fig. 12.1). At each step, a cloud is selected as a block if it does not cross $C$ and does not overlap a previously constructed block.

**Figure 12.1** Pseudocode of the $\mathbf{BlockPartition}$ algorithm. $\mathbf{AreCrossing}(C, c)$ returns true iff a cloud $c$ crosses a cloud-set $C$. $\mathbf{SortBySize}(Expansion)$ sorts clouds in $Expansion$ in the decreasing order of their size. $Expansion(C)$ computes the expansion of a cloud-set $C$.

```
1: procedure BlockPartition(C)
2:   UsedChars ← ∅
3:   Blocks ← ∅
4:   for all c ∈ SortBySize(Expansion(C)) do
5:     if not AreCrossing(C, c) and c ∩ UsedChars = ∅ then
6:       add c to Blocks
7:       add characters from c to UsedChars
8:   return Blocks
```

### Block tree of a cloud-set.

Below we introduce the concept of a block tree of a cloud-set.

Let $G$ be a permutation that conforms with a cloud-set $C$ and $S = \{s_1, \ldots, s_k\}$ be a set of non-intersecting cells in $C$. Let $c$ be a cloud in $C$ such that every $s \in S$ either lies within $c$ or does not overlap $c$. We define *S-glued cloud* $c^S$ as a cloud where for all $s \in S$ such that $s \subset c$ all elements in $s$ are glued into a single character $s^*$. We define *S-glued cloud-set* $C^S$ as a cloud-set where every cloud $c \in C$ is $S$-glued. $C^S$ can be defined iteratively in the following way: let $A_0 = C$, and $A_i = A_{i-1}^{s_i}$ for $0 < i \leq k$. Then $C^S = A_k$. Thus lemmas 6 and 7 can be applied to the complex gluing defined above iteratively.

Let $G[c]$ be a substring of $G$ conforming with a cloud $c$. Similarly, we define the string $G^S$ as the string obtained from $G$ by compressing $G[s]$ into a single character $char(s)$ for every $s \in S$.

We will use the following property of the block partition to introduce a tree which represents $Expansion(C)$.

Lemma 9. *If $B$ is the block partition of a viable cloud-set $C$ then $C^B$ is pseudo-complete (and thus it is either complete or trivial).*

Proof. Let $s$ be a cell in cloud-set $C$. Let $B_1$ be a block partition of $C^s$. Lemma 8 implies that every cloud in $B^s$ is a block in $B_1$. Thus $|B_1| \geq |B|$. Since every element in $composition(c^s)$ is contained by a block in $B^s$, and blocks in $B_1$ do not overlap, $|B_1| \leq |B|$. Thus the number of blocks in the block partition remains the same after gluing cells. Thus the block partition of $C^B$ consists of singletons that are the results of gluing each block in the block partition of $C$. Thus all cells in $C^B$ are singletons and every nontrivial cloud in $C^B$ crosses $C^B$. Lemma 4 implies that $C^B$ is either complete or trivial. □

Lemma 9 implies that, for every block partition $B$, the ordering of blocks in a permutation that conforms $C^B$ is either completely reconstructable (if $C^B$ is complete), or completely unknown (if $C^B$ is trivial). We use this property to represent the cloud-set $C$ as a tree.

Let $C$ be a cloud-set and $s$ be a subset of $char(C)$. A *reduced cloud-set* $C_s$ is defined as the set of all clouds in $Expansion(C)$ that are subsets of $s$.

We define the *block tree* of a cloud-set $C$ (denoted $BT(C)$) as follows. Every vertex of $BT(C)$ is a cell of $C$, and the root of $BT(C)$ is $char(C)$. Let $v$ be a vertex of $BT(C)$, and $Blocks(v) = \{v_1, \ldots, v_k\}$ be a block partition of $C_v$. Then $v$ has $k$ children $v_1, \ldots, v_k$. If $|v| = 1$, then $v$ is a leaf. Every internal vertex of $BT(C)$ is either *ordered* or *unordered*. Vertex $v$ is ordered iff $(C_v)^{Blocks(v)}$ is complete. If $b_1 \ldots b_k$ is the permutation that conforms with $(C_v)^{Blocks(v)}$, we say it is a *children ordering* of $v$ (denoted as $Ordering(v)$).

Figure 12.2 describes the algorithm $\mathbf{BlockTree}(C)$ that returns the root of the constructed block tree $BT(C)$.

**Figure 12.2** Pseudocode of the $\mathbf{BlockTree}$ algorithm. $SingleVertex$ denotes a block tree with a single vertex that has no children. $AddChild(a, b)$ adds a child vertex $b$ to a vertex $a$. $Ordered(Vertex)$ is true iff a vertex $Vertex$ is ordered.

```
1: procedure BlockTree(C)
2:   if |char(C)| = 1 then
3:     return SingleVertex
4:   Root ← SingleVertex
5:   Blocks ← BlockPartition(C)
6:   for all Block ∈ Blocks do
7:     AddChild(Root, BlockTree(C_Block))
8:   if C is a trivial cloud-set then
9:     Ordered(Root) ← False
10:  else
11:    Ordered(Root) ← True
12:    Ordering(Root) ← FindCloudSuperstring(C^Blocks)
13:  return Root
```

## 13 Clouded Eulerian Path Problem

Ideally, genomic cycle in the assembly graph $DB$ corresponds to an Eulerian cycle in the contracted assembly graph $CDB$. However, genomic cycle in the assembly graph is usually broken into paths due to coverage breaks. As a result, genomic cycle in the contracted assembly graph often breaks into multiple paths. Below we apply the Clouded Eulerian Path Problem to reconstructing subpaths of the Eulerian cycle in the subgraphs of the contracted assembly graph.

### Using the block tree to solve the Clouded Eulerian Path Problem.

The Clouded Eulerian Path Problem is *resolvable* (for a graph $G$, its cloud-set $C$, and its transition-set $T$) if it has a solution. Let $v$ be a vertex of the block tree that has children $v_1, \ldots, v_k$, and $Paths(v)$ be a set of clouded Eulerian paths that conform with $C_v$. Lemma 10 below implies that if $v$ is ordered, every element of $Paths(v)$ can be represented as a concatenation of paths $p_1 \ldots p_k$, where $p_i \in Paths(v_i)$.

To solve the CEP problem, we will use the block tree to split a subset $s$ of edges of the graph $G$ into smaller subsets using information provided by the cloud-set $C$. We solve the CEP problem iteratively starting from leaves of the block tree and going up to the root.

Let $G$ be a directed graph, $T$ be a transition-set, and $C$ be a cloud-set over the edge-set of $G$. Let $Paths(G, T, C)$ be a set of $T$-compatible Eulerian paths in $G$ that conform with $C$. Let $P$ be a path in $Paths(G, T, C)$, and $v$ be a vertex in the block tree $BT(C)$. We denote the subgraph induced in $P$ by $v$ as $P[v]$. Note that every vertex $v$ in the block tree $BT(C)$ corresponds to a cell in $C$. Thus, for every path $P \in Paths(G, T, C)$ and a vertex $v$ in $BT(C)$, the induced subgraph $P[v]$ is a subpath of $P$. We use this property to apply a bottom-up approach to solve the Clouded Eulerian Path problem. First, we find clouded Eulerian subpaths for all children of a given vertex of the block tree, and then concatenate them in a larger path. The following lemma describes how to

concatenate subpaths in the correct order.

Lemma 10. *Let $G$ be a directed graph, $C$ be a cloud-set over its edge-set, $T$ be a set of transitions, and $P$ be a clouded Eulerian path for $G$, $T$ and $C$. Let $v$ be an ordered vertex in $BT(C)$, and $P[v]$ be a subpath of $P$ corresponding to $v$. Let $Blocks(v) = \{v_1, \ldots, v_k\}$ be the children of $v$. Then $P[v] = P[v_1] \ldots P[v_k]$.*

Proof. Since $(C_v)^{Blocks(v)}$ is complete, Theorem 11 implies that the children ordering of $v$ $Ordering(v)$ is the only string that conforms with $(C_v)^{Blocks(v)}$. Let $G(v)$ be a string of vertices corresponding to $P[v]$. Then $G(v)^{Blocks(v)}$ conforms with $(C_v)^{Blocks(v)}$, which concludes the proof. □

Let $v$ be a vertex in the block tree that has children $v_1, \ldots, v_k$, and $Paths(v)$ be a set of $T$-compatible Eulerian paths in induced subgraph $G[v]$ that conform with $C_v$. Lemma 10 shows that if $v$ is ordered, every element of $Paths(v)$ can be represented as concatenation of paths $p_1 \ldots p_k$, where $p_i \in Paths(v_i)$. Not every sequence of paths can be concatenated, since the edge-pair formed by the last edge of $p_i$ and the first edge of $p_{i+1}$ is not necessarily a transition. To avoid checking concatenation of every possible sequence $p_1 \ldots p_k$, we represent $Paths(v)$ as a set of starts and ends of all paths in $Paths(v)$ and concatenate these sets instead. Below we introduce the matrix representation of $Paths(v)$.

Let $v$ be a vertex in the block tree. If an edge in $v$ is a start/end of a path from $Path(v)$, we say that it is a *starting/ending* edge of $v$. Let $Starts(v)$ be the set of starting edges of $v$, and $Ends(v)$ be the set of ending edges of $v$. Let *start-end matrix* of $v$ (denoted $SE(v)$) be a matrix with index sets $|Starts(v)|$, $|Ends(v)|$, where $SE(v)_{i,j}$ is a number of paths in $Paths(v)$ starting with edge $i$ and ending with edge $j$.

Let $v$ be a vertex in $BT(C)$ with children ordering $v_1, \ldots, v_k$. Let $A_i$ be the adjacency matrix of the bipartite graph with partitions $Ends(v_i)$, $Starts(v_{i+1})$ and edges induced from transitions. Then

$$SE(v) = SE(v_1)A_1SE(v_2)A_2 \ldots A_{k-1}SE(v_k).$$

### Clouded Eulerian path algorithm.

We are now ready to describe the algorithm for solving the Clouded Eulerian Path Problem. Let $G$ be a directed graph, $C$ be a cloud-set over its edge-set, and $T$ be a transition-set. We assume that at least one clouded Eulerian path exists. If there exists exactly one clouded Eulerian path $P$, the procedure $\textbf{StartEndMatrix}(G, T, BlockTreeRoot)$ (Figure 13.1) returns a matrix with exactly one non-zero cell $(s, e)$, where $s$ is the starting edge of $P$ and $e$ is the ending edge of $P$. In that case, the procedure $\textbf{CloudedPath}(G, T, C)$ returns the path $P$ using backtracking. If there are multiple clouded Eulerian paths, the procedure $\textbf{CloudedPath}(G, T, C)$ returns nothing.

## 14 cloudSPAdes outline

cloudSPAdes takes a set of barcoded reads $Reads$ and integers $k$, $LT$, $LT^+$ as an input. The pseudocode in Figure 14.1 outlines the steps of the cloudSPAdes algorithms.

## 15 Information about the datasets

The YEAST dataset contains $22{,}223{,}405$ paired-end Illumina reads (read length 71, average insert length 309), all of which are barcoded. (the barcode length is 18 nt). The SLR library for the YEAST dataset includes $1{,}044{,}855$ containers.

The STAPH dataset contains $16{,}043{,}114$ paired-end Illumina reads (read length 116, average insert length 168), all of which are barcoded. (the barcode length is 18 nt). The SLR library for the STAPH dataset includes $651{,}927$ containers.

The ECOLI dataset contains $4{,}641{,}046$ paired-end Illumina reads (read length 71, average insert length 299), all of which are barcoded. (the barcode length is 18 nt). The SLR library for the ECOLI dataset includes $406{,}270$ containers.

The MOCK5 dataset contains $107{,}022{,}096$ paired-end Illumina reads (read length 150, average insert length 380), $101{,}416{,}640$ out of them are barcoded. (the barcode length is 16 nt). The SSLR library for the MOCK5 dataset includes $1{,}659{,}903$ containers. Table 15.1 presents species abundances for the MOCK5 dataset (Danko *et al.*, 2019).

The MOCK20 dataset contains $166{,}268{,}091$ paired-end Illumina reads (read length 135, average insert length 409), $163{,}213{,}517$ out of them are barcoded. (the barcode length is 16 nt). The SSLR library for the MOCK5 dataset includes $2{,}169{,}937$ containers. Since some of the 19 similar genomes have low abundances (and/or significantly differ from similar genomes), we selected 10 out of 19 genomes with at least 96% genome fraction (as reported by metaQUAST) as the *reference genomes* for the MOCK20 dataset. Table 15.2 presents species abundances for the MOCK20 dataset (Bishara *et al.*, 2018).

The GUT dataset contains $152{,}445{,}141$ paired-end Illumina reads (read length 150, average insert length 411), $131{,}165{,}841$ out of them are barcoded. (the barcode length is 16 nt). The SSLR library for the GUT dataset includes $2{,}334{,}485$ containers.

Since the individual genomes forming the GUT dataset are unknown, we attempted to infer them using the assembled genomes from the RefSeq database (O'Leary *et al.*, 2016). Specifically, we extracted 27 genomes from this database that share at least 80% of their $k$-mers with contigs assembled by metaSPAdes launched on the GUT dataset (with $k$=55). We used the Mash tool (Ondov *et al.*, 2016) to estimate the fraction of shared $k$-mers. We then used metaQUAST to estimate the *genome fraction* of the selected 27 assemblies covered by metaSPAdes contigs. Afterwards, we selected 13 out of 27 genomes with at least 90% genome fraction and at most 20 metaSPAdes misassemblies (as reported by metaQUAST) as the *reference assemblies* for the GUT dataset. The rationale for retaining only references with 20 or less metaSPAdes misassemblies is that large numbers of assembly errors are likely caused by differences with the reference genomes making such diverged references inappropriate for our benchmarking (previous benchmarking studies revealed that metaSPAdes is rather accurate). Finally, we broke contigs in the reference assemblies along the regions that were poorly covered by paired-end reads (less than 20% of the mean coverage).

### Data availability

All sequencing reads from this study are available at `https://s3.us-east-2.amazonaws.com/readclouds/cloudspades_data.tar.gz`.

| Organism | reported DNA fraction |
|---|---|
| Escherichia coli | 0.2939 |
| Enterobacter cloacae | 0.3117 |
| Micrococcus luteus | 0.1219 |
| Pseudomonas antarctica | 0.1148 |
| Staphylococcus epidermidis | 0.1557 |

Table 15.1. **Species abundances for the MOCK5 dataset.**

**Figure 13.1** Pseudocode of the $\mathbf{CloudedPath}$ algorithm. Procedure $\mathbf{StartEndMatrix}(G, T, Vertex)$ computes the start-end matrix for a vertex $Vertex$, graph $G$ and transition set $T$. $\mathbf{MergeMartices}(G, T, Matrices, Ordering)$ construct the start-end matrix of the vertex from the start-end matrices $Matrices$ of its children with ordering $Ordering$ as described above. $\mathbf{UniquePath}(G, T, s, e, BlockTreeRoot)$ returns the Clouded Eulerian Path with the starting edge $s$ and the ending edge $e$ by performing a backtracking procedure using block tree with root $BlockTreeRoot$.

1: **procedure** $\mathbf{CloudedPath}(G, T, C)$
2: $BlockTreeRoot \leftarrow \mathbf{BlockTree}(C)$
3: $StartEndMatrix \leftarrow \mathbf{StartEndMatrix}(G, T, BlockTreeRoot)$
4: **if** there are $s$, $e$ such that $StartEndMatrix_{s,e} = 1$ and $StartEndMatrix_{i,j} = 0$ for all $i \neq s$, $j \neq e$ **then**
5: **return** $\mathbf{UniquePath}(G, T, s, e, BlockTreeRoot)$
6: **else**
7: **return** $\varnothing$
8:
9: **procedure** $\mathbf{StartEndMatrix}(G, T, Vertex)$
10: **if** $Vertex = \{e\}$ **then**
11: **return** matrix $A$ with a single element $A_{e,e} = 1$
12: $Children \leftarrow$ children of $Vertex$
13: $Matrices \leftarrow EmptyList$
14: **for all** $Child \in Children$ **do**
15: add $\mathbf{StartEndMatrix}(G, T, Child)$ to $Matrices$
16: **if** $Ordered(Vertex)$ **then**
17: **return** $\mathbf{MergeMatrices}(G, T, Matrices, Ordering(Vertex))$
18: **else**
19: $StartEndMatrix \leftarrow EmptyMatrix$
20: $UniquePaths \leftarrow$ empty set
21: **for all** $Ordering$ in permutations of $Children$ **do**
22: $NewMatrix \leftarrow \mathbf{MergeMartices}(G, T, Matrices, Ordering)$
23: $StartEndMatrix = StartEndMatrix + NewMatrix$
24: **return** $StartEndMatrix$

| Organism | reported DNA fraction |
|---|---|
| Streptococcus mutans | 0.18 |
| Porphyromonas gingivalis | 0.18 |
| Staphylococcus epidermidis | 0.18 |
| Escherichia coli | 0.18 |
| Rhodobacter sphaeroides | 0.18 |
| Bacillus cereus | 0.018 |
| Pseudomonas aeruginosa* | 0.018 |
| Streptococcus agalactiae | 0.018 |
| Clostridium beijerinckii | 0.018 |
| Staphylococcus aureus | 0.018 |
| Acinetobacter baumannii** | 0.0018 |
| Neisseria meningitidis | 0.0018 |
| Propionibacterium acnes | 0.0018 |
| Helicobacter pylori | 0.0018 |
| Lactobacillus gasseri | 0.0018 |
| Bacteroides vulgatus | 0.0002 |
| Deinococcus radiodurans | 0.0002 |
| Actinomyces odontolyticus | 0.0002 |
| Bifidobacterium adolescentis | 0.0002 |
| Enterococcus faecalis | 0.0002 |

Table 15.2. **Species abundances for the MOCK20 dataset.**

## 16 Analysis of SSLRs

Although all SLR technologies follow the pipeline illustrated in Figure 1, they differ in the mean fragment length ($fragmentLength$), mean coverage of each fragment by short reads ($coverage$), mean number of fragments in a single container ($fragmentNumber$), and the number of containers ($containerNumber$). For example, the TSLR technology is characterized by the following typical parameters: $fragmentLength =$ 10 kb, $fragmentNumber = 300$, $containerNumber = 384$, $coverage = 10$x (Bankevich and Pevzner, 2016; McCoy *et al.*, 2014; Kuleshov *et al.*, 2014). In contrast, the SSLR technology is characterized by the following typical parameters: $fragmentLength = 80$ Kb, $fragmentNumber = 10$, $containerNumber = 1,500,000$, $coverage = 0.05$x (Kuleshov *et al.*, 2016; Mostovoy *et al.*, 2016).

The SSLR parameters listed above represent typical values that may significantly deviate from parameter that characterize a specific SSLR dataset. For example, SSLR libraries for eukaryotic genomes have very different parameters ($fragmentLength$, $fragmentNumber$, $coverage$) than metagenomic SSLR libraries, moreover, different metagenomic SSLRs may have very different parameters. These differences suggest that, to generate a good assembly, one should take into account the parameters of a specific dataset. Below we analyse SSLR parameters of various metagenomic samples.

### Estimating parameters of SSLR datasets.

Given a set of reference genomes forming a metagenome, we map all short reads from a single container to these references. We use the Lariat SSLR aligner (Bishara *et al.*, 2015) to filter out multiple alignments of barcoded reads.

We use the short read alignments to estimate the unknown $fragmentLength$, $fragmentNumber$ and $coverage$ parameters of

**Figure 14.1 The pseudocode of the cloudSPAdes algorithm**. $\textbf{AssemblyGraph}(Reads, k)$ constructs the assembly graph using metaSPAdes. $\textbf{ContractedGraph}(DB, LT)$ contracts all edges shorter than $LT$ in the assembly graph $DB$. $\mathbf{T}^*(DB)$ infers transitions from $DB$ and eliminates false transitions as described in Appendix 4. $\textbf{Clouds}^*(Graph, Transitions, Reads)$ extracts putative clouds from the contracted assembly graph $Graph$ and filters them using $Transitions$ as described in Appendix 7. $\textbf{CloudedPath}(Graph, Transitions, Clouds)$ reconstructs clouded Eulerian path in $Graph$ as described in Appendix 13. Construction of the gap subgraph between consecutive edges $e_1$ and $e_2$ is described in Appendix 8. $\textbf{Contigs}(DB, Reads)$ applies exSPAnder algorithm to generate contigs from the assembly graph and paired-end reads. $\textbf{Scaffolds}(Contigs, Paths_{LT}, LT)$ merges given $Contigs$ using consecutive pairs from $Paths_{LT}$ as described in Appendix 9. $\textbf{CloseGaps}(Scaffolds, Reads)$ closes gaps in $Scaffolds$ using paired-end barcoded reads $Reads$ as described in Appendix 10. $\textbf{MergeScaffolds}(Scaffolds, LT)$ merges $Scaffolds$ by constructing the contracted assembly graph on scaffold edges longer than $LT$ as described in Appendix 9.

```
1: procedure cloudSPAdes(Reads, k, LT, LT+)
2:    DB ← AssemblyGraph(Reads, k)                              ▷ constructing assembly graph using metaSPAdes
3:    DB_LT+ ← ContractedGraph(DB, LT+)                         ▷ constructing contracted assembly graph on ultralong edges
4:    Transitions_LT+ ← T*(DB_LT+, Reads)                       ▷ constructing transitions on ultralong edges
5:    Components ← weakly connected components of DB_LT+
6:    Paths_LT+ ← empty set of paths
7:    for all Component in Components do
8:        Clouds ← Clouds*(Component, Transitions_LT+, Reads)   ▷ generating clouds
9:        Path ← CloudedPath(Component, Transitions_LT+, Clouds) ▷ constructing clouded Eulerian path
10:       add Path to Paths_LT+
11:   Paths_LT ← empty set of paths
12:   DB_LT ← ContractedGraph(DB, LT)                           ▷ constructing contracted assembly graph on long edges
13:   Transitions_LT ← T*(DB_LT, Reads)                         ▷ constructing transitions on long edges
14:   for all pair of consecutive edges (e1, e2) in Paths_LT do
15:       Subgraph ← GapSubgraph(DB_LT, e1, e2)                 ▷ getting gap subgraph between e1 and e2
16:       Clouds ← Clouds*(Subgraph, Transitions_LT, Reads)     ▷ generating putative clouds
17:       Path ← CloudedPath(Subgraph, Transitions_LT, Clouds)  ▷ constructing clouded Eulerian path
18:       add Path to Paths_LT
19:   Contigs ← Contigs(DB, Reads)                              ▷ constructing metaSPAdes contigs
20:   Scaffolds ← Scaffolds(Contigs, Paths_LT, LT)              ▷ scaffolding contigs using paths in DB_LT
21:   Scaffolds ← CloseGaps(Scaffolds, Reads)                   ▷ closing gaps within scaffolds
22:   Scaffolds ← MergeScaffolds(Scaffolds, LT)                 ▷ merging scaffolds
23:   Scaffolds ← CloseGaps(Scaffolds, Reads)                   ▷ closing gaps in merged scaffolds
24:   return Scaffolds
```

a SSLR library. We use *single linkage clustering* to partition the mapped reads into clusters corresponding to alignments of (unknown) fragments to reference genome. Two reads are combined into the same cluster if they are mapped to the same genome and the distance between them does not exceed a threshold $Distance$ (see the description of the threshold selection procedure below).

We further compute the *span* of each of the resulting clusters (as the distance between its endpoints) and limit attention to clusters of length at least $minSpan$ (the default value is 2000). Span of the cluster corresponds to the $fragmentLength$ parameter. We estimate the $coverage$ of a cluster as the ratio of total read length falling into this cluster and its span. We estimate $fragmentNumber$ as the number of resulting clusters.

## Setting the default value for the $Distance$ parameter.

There is a trade-off between selecting small and large values of the $Distance$ parameter. Small values of $Distance$ lead to fragmented clusters, while large values lead to combining multiple clusters into a single one, thus creating false clusters.

We estimate an optimal value of $Distance$ using the median span of the clusters. Typically, the median span of the clusters increases with the increase of $Distance$ because the set of clusters corresponding to a single read cloud becomes less fragmented. Since there are few collisions, we presume that two distinct read clouds rarely merge even at high values of $Distance$. Therefore the increase of the mean span of the clusters with the growth should decline when $Distance$ is equal to maximum $fragmentLength$.

Figure 16.1 illustrates this point and reveals a slight drop in the rate of the cluster span growth around $Distance = 40$ kb for the MOCK5 dataset and $Distance = 5$ kb for the GUT dataset, making them reasonable parameter choices for cluster construction. Figure 16.2 shows the cluster span distribution for different values of $Distance$.

## SLR statistics of metagenomic datasets.

Figures 16.3, 16.4 and 16.5 show the distribution of spans of the resulting clusters ($fragmentLength$), the distribution of the number of clusters ($fragmentNumber$) per container, and the distribution of the coverage of clusters ($coverage$) for the MOCK5 and GUT datasets. For GUT dataset, we use contigs longer than 50kb from 16 inferred reference assemblies to obtain cluster statistics. These Figures illustrate that the SSLR statistics of different metagenomic datasets are quite different. Thus, each SSLR metagenome assembler has to adapt to parameters of a specific SSLR dataset.

## SLR statistics per reference.

Figures 16.7 and 16.6 show the distribution of spans of the clusters ($fragmentLength$), the distribution of the coverage of clusters ($fragmentCoverage$), and the distribution of the number of clusters ($fragmentNumber$) per container for four reference genomes in

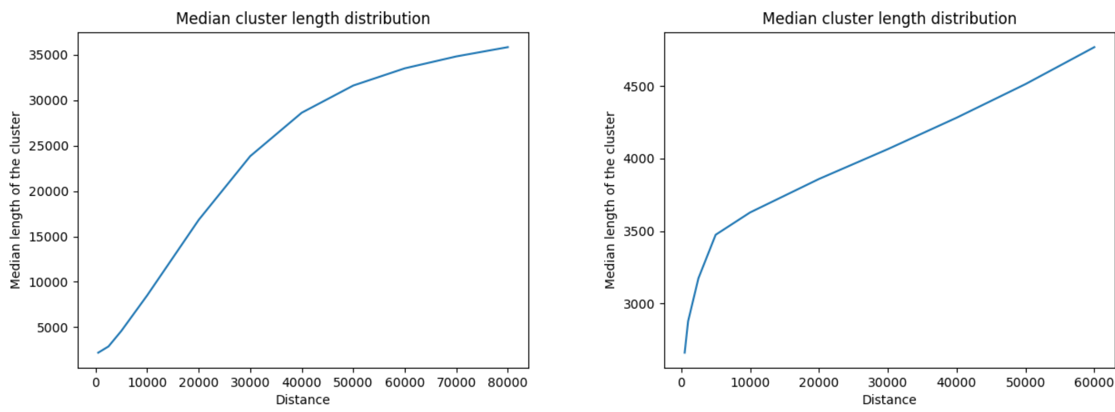


Fig. 16.1: **Distribution of the median cluster length for the MOCK5 (left) and GUT (right) datasets.**

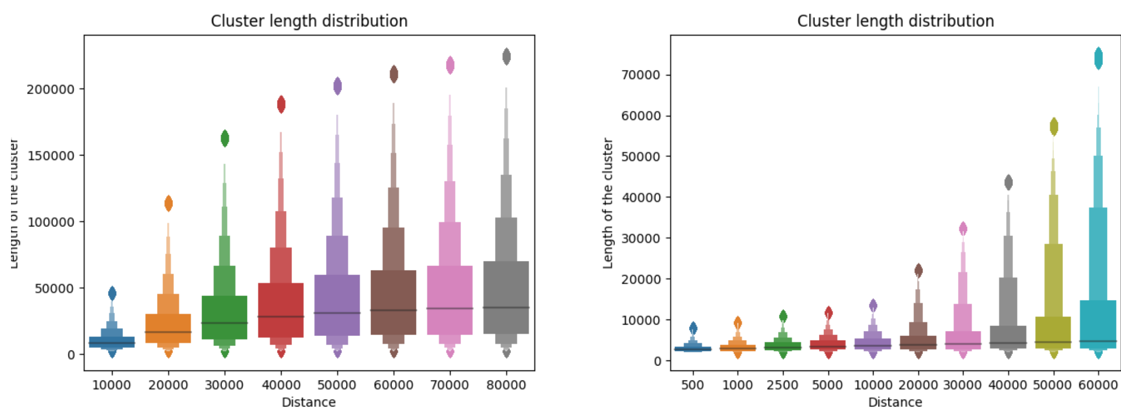


Fig. 16.2: **Distribution of the cluster spans for different values of $Distance$ for the MOCK5 (left) and GUT (right) datasets.**

MOCK5 dataset. As these Figures illustrate there are significant variations in the SSLR parameters across various genomes within the same metagenomic dataset.

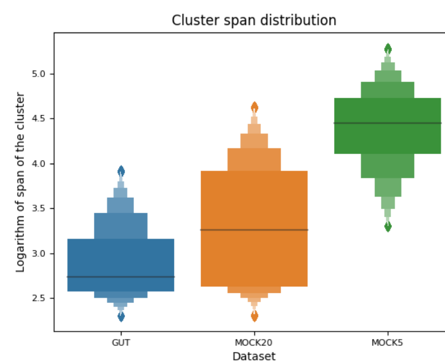


Fig. 16.3: **Distributions of cluster spans (in the decimal logarithmic scale) for the GUT, MOCK20 and MOCK5 datasets.** Mean cluster length is $1,218$ for GUT dataset, $5,707$ for MOCK20 dataset and $39,139$ for MOCK5 dataset.

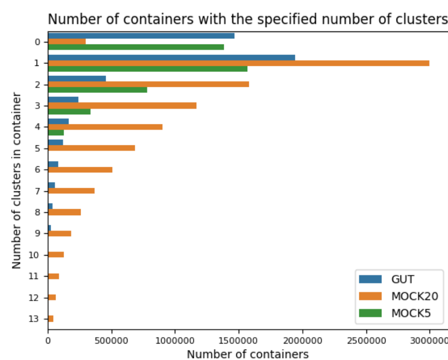


Fig. 16.4: **Distribution of the number of clusters per container for the GUT, MOCK20 and MOCK5 datasets.**

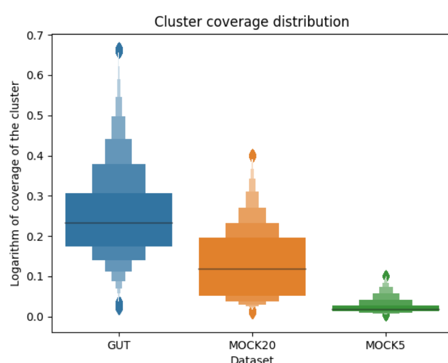


Fig. 16.5: **Distribution of cluster coverage (in the decimal logarithmic scale) for the GUT, MOCK20 and MOCK5 dataset.** The mean cluster coverage is 0.83 for GUT dataset, 0.38 for MOCK20 dataset and 0.05 for MOCK5 dataset

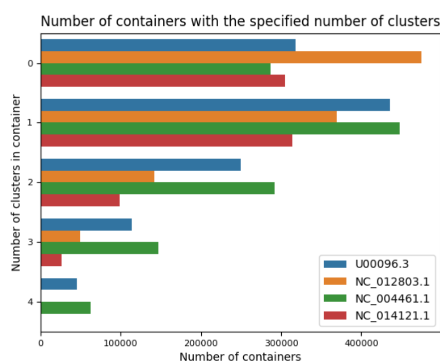


Fig. 16.6: **Distribution of the number of clusters per container for four reference genomes in the MOCK5 dataset.** Reference genomes are represented by their RefSeq IDs.

## 17 Benchmarking on simulated datasets

We have simulated 13 datasets with various $fragmentLength$, $coverage$, $fragmentNumber$, and $containerNumber$ parameters based on the MOCK5 dataset to demonstrate how SSLR library parameters affect tool performance. Given these four parameters, parameters $readLength$ and $insertSize$, and reference genomes, simulating process performs the following steps:

1. The total number of fragments $TotalFragments$ is defined as $fragmentNumber \cdot containerNumber$.
2. From every reference genome $fraction \cdot TotalFragments$ fragments are generated, where $fraction$ is the DNA fraction of the genome in the dataset (in the case of MOCK5 dataset, fractions from Table 15.1 were used). Every fragment $f$ corresponds to a substring of the reference genome, and is assigned length $length(f)$, number of corresponding read-pairs $reads(f)$, container $container(f)$ and a start position in the reference $start(f)$.
3. Length $length(f)$ is drawn from an exponential distribution with the rate $\lambda = \frac{1}{fragmentLength}$.
4. Number of read-pairs $reads(f)$ is drawn from the Poisson distribution with the rate $\lambda = \frac{coverage \cdot length(f)}{2readLength}$
5. Container $container(f)$ is drawn from a uniform distribution $unif\{0, containerNumber\}$.
6. Start position $start(f)$ is drawn from a uniform distribution $unif\{0, length(Genome)\}$.
7. For every fragment $f$, $reads(f)$ read-pairs are generated. Every read-pair $read$ corresponds to a substring of the genome string, starting from position $start(read)$ which is drawn uniformly from the segment $[start(f), start(f)+length(f)-insertSize]$. Every read-pair is assigned a barcode marking $container(f)$.

We used parameters $fragmentLength = 10000$, $coverage = 0.1$, $fragmentNumber = 10$, $containerNumber = 50000$, $readLength = 150$, and $insertSize = 400$ for a baseline simulated dataset. We created 3 additional datasets for every SSLR parameter ($fragmentLength$, $coverage$, $fragmentNumber$, and $containerNumber$) by changing the value of that parameter and leaving other parameters from the baseline dataset unchanged. We refer to the simulated dataset by the name and value of its changed parameter (e.g. dataset $(Fragment\ length, 20000)$ denotes simulated dataset with parameters $fragmentLength = 20000$, $coverage = 0.1$, $fragmentNumber = 10$, $containerNumber = 50000$). Figure 17.1 shows mean NGA50 metric per reference and mean number of misassemblies per reference for metaSPAdes, Athena, cloudSPAdes, Architect, ARCS and Supernova for every simulated dataset. All tools except Architect demonstrate stable performance against changes of $fragmentNumber$ and $containerNumber$ parameters. All tools except Architect show lower NGA50 on the $(Fragment\ length, 20000)$ dataset than on the $(Fragment\ length, 10000)$ dataset.

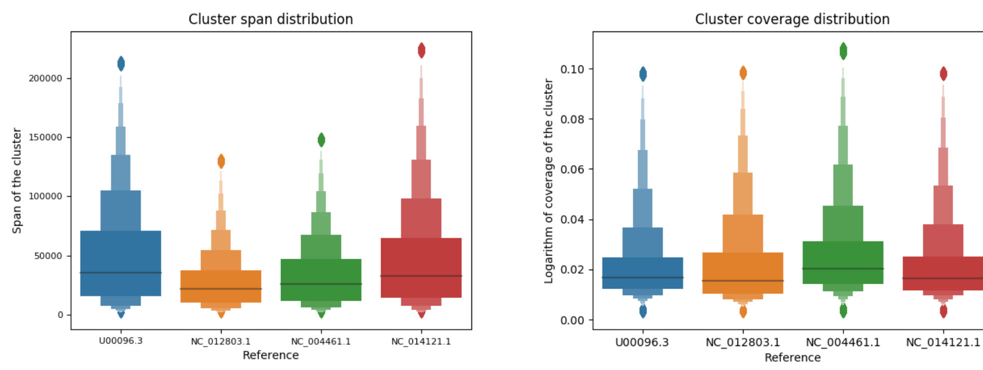


Fig. 16.7: **Distribution of cluster spans (left) and the coverage of clusters (right) for four reference genomes in the MOCK5 dataset.** Reference genomes are represented by their RefSeq IDs.

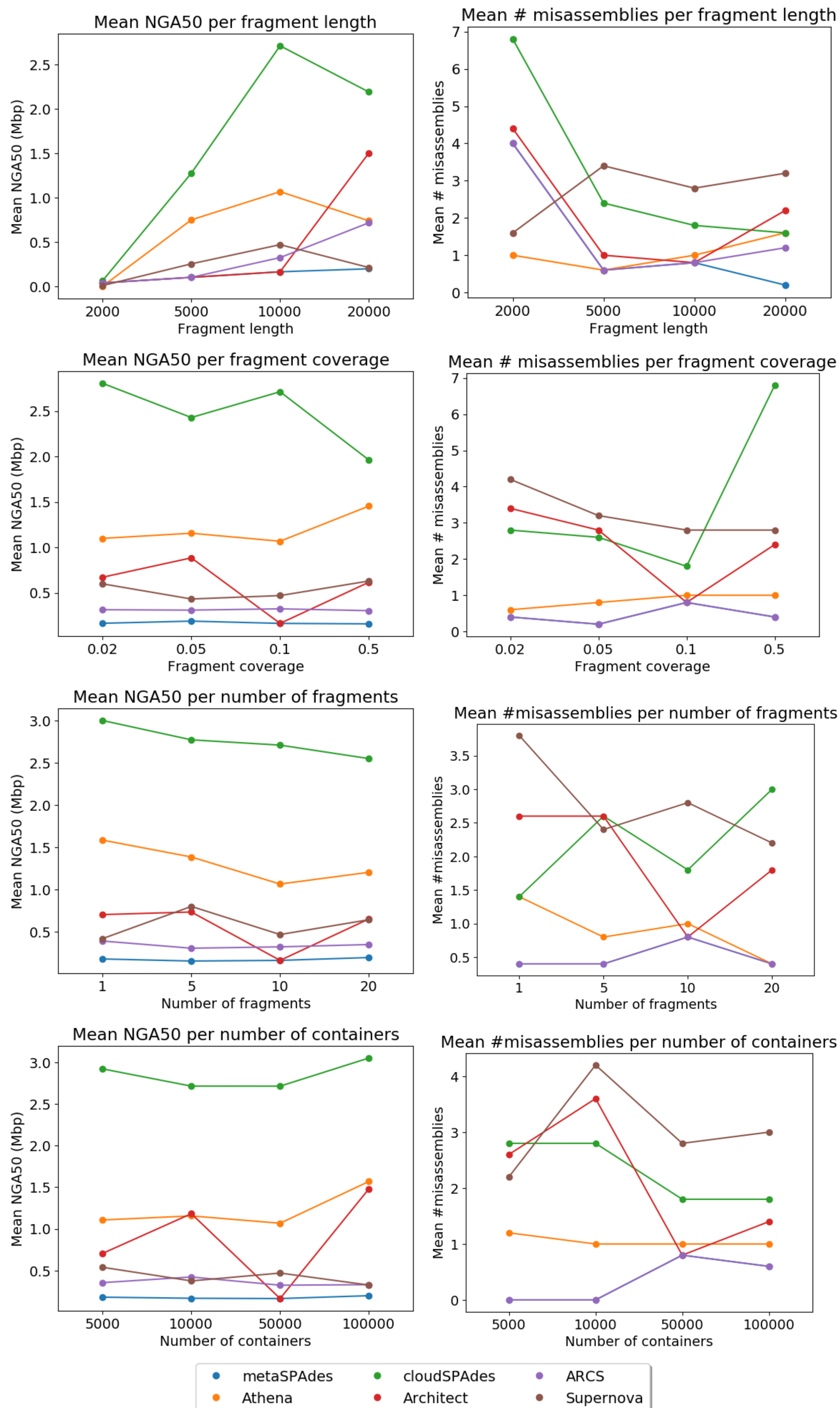


Fig. 17.1: **Assembly statistics for the simulated datasets. (First layer)** Mean NGA50 **(left)** and mean number of misassemblies **(right)** for given fragment length. **(Second layer)** Mean NGA50 **(left)** and mean number of misassemblies **(right)** for given fragment coverage. **(Third layer)** Mean NGA50 **(left)** and mean number of misassemblies **(right)** for given number of fragments. **(Fourth layer)** Mean NGA50 **(left)** and mean number of misassemblies **(right)** for given number of containers.