Genomorientierte Bioinformatik

Report

Read Simulator

Malte Weyrich

November 2024

In bioinformatics, ***Sequencing*** is a term used to describe the process of gathering genomic data by reading the nucleotides of a DNA molecule. This is done by a *sequencer*. There are different types of sequencing techniques (e.g *Illumina* (next generation sequencing), *Oxford Nanopore*, *Pacbio* (third generation sequencing), etc.) and different variants of *sequencing* (*ATAC-seq*, *scRNA-seq*, *ChIP-seq*, ...). In this report, we will focus on a *Read Simulator*, which is a tool used to simulate the results of a sequencing experiment (in this case, paired-end **Sequencing** using *Illumina*). The program was executed using *"Homo_sapiens.GRCh37.75.dna.toplevel.fa"* as *Reference Genome*, its corresponding *fasta index* and its annotation in form of a *Gene Transfer Format*-File (GTF). The *Read Simulator* was written in *Java* and will be benchmarked and analyzed by its complexity and correctness. The results of the simulator itself will also be discussed in this report.

1 — Introduction

The simplified process of *Illumina* sequencing is as follows: Several DNA target sequences get treated with ultrasound in order to break them down into smaller fragments of a certain length with a certain margin of error (e.g. 200 bp +/- x bp). These fragments are labeled, placed onto a flow cell, amplified, and sequenced. This generates an abundance of short reads, which often overlap with each other and contain mutations. The reads are then aligned to a *Reference Genome* via a *Mapper* like *STAR* in order to determine the original sequence. The *Read Simulator* is a tool that simulates this process of generating fragments and reads for given transcript sequences. In our case, we simulate a paired-end sequencing experiment, where we generate two reads for each fragment, one for each end of the fragment. With a *Read Simulator*, we are able to validate the results of *Mappers* since we know where our reads originated from. In other words, if a *Mapper* was to incorrectly map a read (generated by our *Read Simulator*) to the *Reference Genome*, we could detect this faulty behavior and see how far off the *Mapper* was to the actual coordinates. Like this, we could also detect entire regions inside the *Reference Genome* where the *Mapper* struggles in general (e.g. highly repetitive regions). A downside of the *Read Simulator* is that it assumes a normal distribution of fragment lengths and their starting position inside the transcript and a constant mutation rate across the entire read sequences, which in reality is not the case. However, it is still a valuable tool for testing the performance of other tools.

The *JAR* was executed using the following configuration:

```
java -jar readSimulator.jar
     -length 75
     -frlength 200
     -SD 80
     -mutationrate 1.0
     -gtf "./inputFiles/Homo_sapiens.GRCh37.75.gtf"
     -fasta "./inputFiles/Homo_sapiens.GRCh37.75.dna.toplevel.fa"
     -fidx "./inputFiles/Homo_sapiens.GRCh37.75.dna.toplevel.fa.fai"
     -readcounts "./inputFiles/readcounts.simulation"
     -od "output"
```

2 — Java Implementation

2.1. Logic

The logic of the *Read Simulator* is split up into four main steps:

(A) Read Gene & Transcript IDs to Simulate:

The user specifies a *tsv* file containing the gene and transcript IDs of the sequences and the amount of reads to simulate for each transcript. The file is passed to the *ReadSimulator* via the `-readcounts` argument. The entries are stored inside a nested *HashMap<String, HashMap<String,Integer>> readCounts* object where the first key corresponds to the gene ID and maps to a second map, which maps the transcript ID to the number of reads to simulate. This way, we avoid storing the same gene ID multiple times and can easily access the amount of reads to simulate a given transcript.

(B) Initialize Genome using a GTF-File:

The *Genome* class is initialized by passing the path of the *Reference Genome* and the *Fasta Index* file to the constructor. The path of the *Reference Genome* is used to create a *RandomAccessFile* object, which is used to access the large fasta file in a more efficient way by utilizing the indices stored in the *Fasta Index*. This will later be used to extract the sequences of genes containing the transcripts we want to simulate reads for. The *GTF* file and *readCounts* object are then passed to the *Genome* object to initialize the gene and transcript coordinates. Each line of the *GTF-File* is filtered using *GenomeUtils.filterLine(line, readCounts)*, which checks if the gene ID of the line is present in the *readCounts* object. The method works by counting the number of seen *<tabs>* in the current line and then extracting the gene ID, which is located in between the 8th and 9th *<tab>* of the line. This way, we don't call expensive *split()* operations on each line of the *GTF-File*. If a valid line was found, we only need to check if it is a *"gene"*/*"transcript"*/*"exon"* entry and either create a new *Gene*/*Transcript*/*Exon* object. A *Gene* can have several *Transcript*'s and a *Transcript* can have several *Exon*'s. Due to the filtering of *GenomeUtils.filterLine*, our *Genome* object will only contain *Genes* and *Transcripts* we want to simulate reads for.

(C) Initialize Gene Sequences of Interest:

In order to simulate reads, we first need to extract the exonic sequences of the *Transcripts*. This is done by calling *Genome.initTargetSequences(readCounts)*, which iterates over all *Genes* and *Transcripts* in the *readCounts* object. For each *Gene*, we extract its sequence using the *RandomAccessFile* object and store it in a *String seq* object. This is done by utilizing the start/end coordinates of the *Gene* together with the *Fasta Index* to can calculate the byte offset of the *Gene* in the *Reference Genome* file and read in the sequence. The *seq* object can now be used to concatenate the *Transcript* sequences by cutting out the exonic sequences of the corresponding *Transcript* based on their start and end coordinates. This way, we avoid repeatedly accessing the *Reference Genome* file, which is time inefficient due to the large size of the file and the vast amount of *Exons* we need to extract. If a *Gene* is located on the reverse strand, we go through the *Exons* in reverse order.

(D) Generate Reads and Write to File:

For each *Transcript* we want to simulate reads for, we sample a random fragment length and starting position from a normal distribution. These two values and the specified *read length* are used to extract two substrings of the *Transcript* sequence:

```
do {
    fragmentLength = (int) Math.round(normalDist.sample());
} while (fragmentLength < length || fragmentLength > transcriptSeq.length());
int maxStartPos = transcriptSeq.length() - fragmentLength;
int randomStartPos = splittableRandom.nextInt(maxStartPos + 1);
String fwSeqRead = transcriptSeq.substring(
                        randomStartPos,
                        randomStartPos + length
                    );
String rwSeqRead = GenomeUtils.revComplement(
                        transcriptSeq.substring(
                            randomStartPos + fragmentLength - length,
                            randomStartPos + fragmentLength
                        ));
```

These two *Strings* resemble the unmutated forward (*fw*) and reverse (*rw*) sequence of the *Read*. The *start/end* positions inside the *Transcript*, the *Read* sequence, the *Read ID* and a *boolean* indicating if the *Read* is the forward or reverse *Read*, are then collected in a *Read* object (*fwRead* or *rwRead*). We iterate over each nucleotide of the *Read* and use the *mutationRate* to determine if we should mutate the nucleotide of the current position. Mutations to the original nucleotide are not considered, so we only mutate to one of the other three nucleotides. The next step is to derive the *Genomic Region Vector* of both *Reads*. In Figure 1, we can see the mapping of the *Reads* to the *Reference Genome* using the prior knowledge of the *Exon* coordinates.

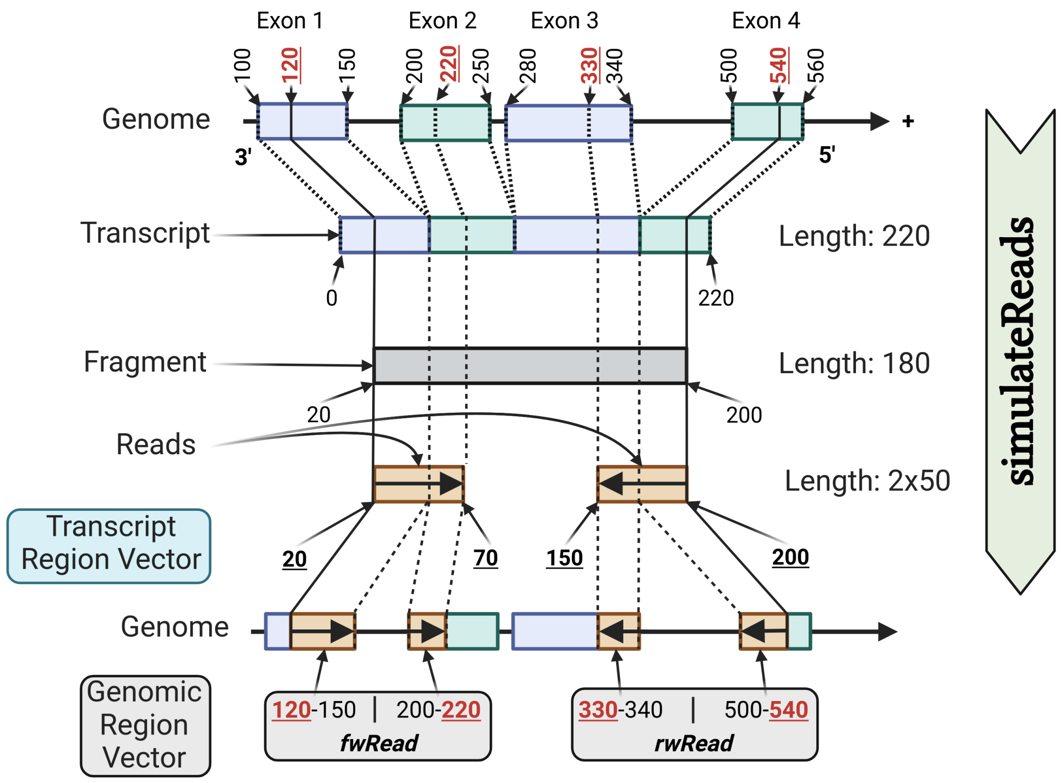

Figure 1 — Mapping of Reads (of length 50) to the Reference Genome using prior knowledge. This figure was created using BioRender.com 2024.

The method *getGenomicRegion* is used to annotate the generated *Reads* with the corresponding *Genomic Region Vector*. We start by extracting the list of *Exons* from the provided *Transcript* object and getting the *Read*'s *start* and *end* positions inside the *Transcript*. For each *Exon*, we calculate its genomic *start*, *end*, and *length*. By checking if the *Read* overlaps with the current *Exon* in *Transcript* coordinates, we identify the overlapping region. This region is then mapped to genomic coordinates, accounting for the strand direction. Finally, the genomic regions are formatted and stored in a list, ensuring proper coordinate ordering for the reverse strand, before proceeding to the next *Exon*.

After both *Reads* have been generated and annotated with their *Genomic Region Vectors*, we write them to two separate *fastq* files (*"fw.fastq"* and *"rw.fastq"*) in the following format:

```
@0
CTAAAAGGCGCAAGAGAATGGATGATAGTAGTGTCCTCGAGGCCACACGGGTT...
+0
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII...
```

The *Read ID* is followed by the *Read* sequence and a *+* sign. The quality scores are represented by the *ASCII* characters *I* in this case. The quality scores are not considered in the *Read Simulator* and are set to the same value for all *Reads*. A summary of the generated *Reads* is also created and written to a *tsv* file (*"read.mappinginfo"*) in the following format (table 1):

Table 1—Format of the Summary File created by the Read Simulator. Each row corresponds to a Read Pair (*fwRead* and *rwRead*).

id	chr	gene_id	transcript_id	t_fw_regvec	t_rw_regvec	fw_regvec	rw_regvec	fw_mut	rw_mut
…	…	…	…	…	…	…	…	…	…

The *Read* objects are discarded after writing them to the files in order to save memory. Writing is done by three separate *BufferedWriter* objects, one for each file. The entries are constructed using three different *StringBuilder* objects, which are then written to the files and reset afterward.

2.2. Complexity

The complexity of the *Read Simulator* can be broken down into the parts described above:

(A) Read Gene & Transcript IDs to Simulate:

Section (A) of the *ReadSimulator* scales linearly with the amount of lines ($|L_{sim}|$) in the *tsv* file, since we create an entry of each line in the *readCounts* object:

$$\mathcal{O}(|L_{sim}|).$$

(B) Initialize Genome using a GTF-File:

Let L_{gtf} be a set of all lines in the provided *GTF-File*. Each line $l \in L_{gtf}$ has to be checked by the *filterLine* method (F) of *FileUtils*:

$$\sum_{i=1}^{|L_{gtf}|} F(l_i).$$

F iterates over all chars in l_i and extracts a substring (*relevantCol*) based on the number of observed <*tabs*> and <*spaces*>. All these operations are constant in $\mathcal{O}(1)$. Let $\bar{l}$ be the upper limit for the length of a *GTF* entry. In the worst case, all lines have the maximum length $\bar{l}$. Then each call of F has less than $\bar{l}$ operations, since we stop processing the line l_i after constructing the relevant substring. In $\mathcal{O}$-notation, $\mathcal{O}(F(l_i)) \in \mathcal{O}(1)$, since $\bar{l}$ is a constant and $\#$ operations of $F < \bar{l}$. For the relevant lines l_{rel} (lines that survived filtering), we perform several *split()* operations on the entry, and, based on the entry type, parse the *gene ID* or *transcript ID*. Let $P(l_{rel})$ describe the remaining amount of work needed for parsing the entry. In the worst case, all lines are relevant and need to be processed with P. Since all these operations are sequential and tied to the number of lines in the *GTF-File*, the overall complexity of **(B)** can be described with:

$$\sum_{i=1}^{|L_{gtf}|} F(l_i) + P(l_{rel}) < \sum_{i=1}^{|L_{gtf}|} F(l_i) + P(l_i) = \sum_{i=1}^{|L_{gtf}|} c_1 + c_2 = |L_{gtf}| \cdot (c_1 + c_2) \in \mathcal{O}(|L_{gtf}|).$$

(C) Initialize Gene Sequences of Interest:

Let G and T be the set of genes and transcript we want to simulate (stored in *readCounts*) and E_{max} be the largest set of exons of a $t \in T$. Let's also assume that each gene has exactly one transcript (worst case, because we need to call *GeneSequenceExtractor.getSequence* for each $t \in T$).

For each entry $(g, t, counts | g \in G \wedge t \in T) \in readCounts$, we need to extract the gene sequence of g. This happens by calculating the *offset* ($\in \mathcal{O}(1)$) and reading in the sequence of length n ($\in \mathcal{O}(n)$). Let's define n_{max} as the longest gene sequence. After extracting the sequence, the transcript sequence needs to be initialized by iterating over its exons. We can use E_{max} as an upper limit for this. Using these variables, the complexity can be described with:

$$\sum_{i=1}^{|readConts|} n_i + |E|_i < \sum_{i=1}^{|readConts|} n_{max} + |E_{max}| \in \mathcal{O}\Big(|readCounts| \cdot (n_{max} + |E_{max}|)\Big).$$

(D) Generate Reads and Write to File:

At this stage of the program, all transcript sequences have been initialized, and all that's left is to simulate the reads. Let R be the sum of all *count entries* of the *readCounts* object, meaning R is the total number of reads to simulate. For each read, we need to:

(CUT) Cut out a random fragment of the transcript

(GEN) Generate two substrings stemming from that fragment (*fwSeqRead, rwSeqRead*)

(MUT) Mutate these substrings

(WRT) Write reads to result files

Let e be the summary content per line. We can formulate the above into the following expression:

$$\sum_{i=1}^{R} \mathbf{CUT}(frlength) + 2 \cdot \mathbf{GEN}(length) + 2 \cdot \mathbf{MUT}(length) + 2 \cdot \mathbf{WRT}(length) + \mathbf{WRT}(e).$$

Creating a substring in *Java* with *String.substring()* has a complexity of the length of the substring. This means that

$$\mathbf{CUT}(frlength) \in \mathcal{O}(frlength) \wedge \mathbf{GEN}(length) \in \mathcal{O}(length).$$

Mutating the reads is also a linear task since we iterate over $length$, and for each iteration, we perform constant operations of $\mathcal{O}(1)$:

$$\mathbf{MUT}(length) \in \mathcal{O}(length).$$

The two *fastq* files generated both contain four lines per read, which is a total of

$$2 \cdot \big(2 \cdot length + 2 + |id|\big)$$

chars. So

$$\mathbf{WRT}\big(2 \cdot (2 \cdot length + 2 + |id|)\big) \in \mathcal{O}(length).$$

The summary file generated contains one entry e for each read pair. Let e_{max} be the entire with the most chars: $\forall_{e \in entries} : |e| \leq |e_{max}|$. This means

$$\mathbf{WRT}(e) \in \mathcal{O}(|e_{max}|).$$

Since $frlength$ and $length$ are fixed constants provided by the user and because e_{max} is also a finite upper bound, the overall runtime complexity is:

$$\sum_{i=1}^{R} \overbrace{\overbrace{\mathbf{CUT}(frlength)}^{c_1} + 2 \cdot \overbrace{\mathbf{GEN}(length)}^{c_2} + 2 \cdot \overbrace{\mathbf{MUT}(length)}^{c_3} + 2 \cdot \overbrace{\mathbf{WRT}(length)}^{c_4} + \overbrace{\mathbf{WRT}(e_{max})}^{c_5}}^{C}$$

$$\iff \sum_{i=1}^{R} C = R \cdot C \implies \mathcal{O}(R \cdot C) \in \mathcal{O}(R)$$

2.3. Correctness

The user can provide two additional flags to the *JAR*:

I. *-debug*

   If *-debug* is provided, the *JAR* will try to validate all *GenomicRegionVectors* by comparing the current (non-mutated) read sequence to a reference sequence. The reference sequence is created by iterating over all coordinates of the corresponding *GenomicRegionVector*. This is done for both the *fwRead* and *rwRead* objects. If a single read sequence does not match the reference sequence, the *JAR* throws a *RuntimeException* and provides the user with the relevant debug information. Additionally, it will also check if all reads are generated. If not, the *JAR* will throw a *RuntimeException*.

II. *-transcriptome <pathToTranscriptome>* (in combination with *-debug*)

   If the user additionally provides a path to a transcriptome corresponding to the input files (*GTF-File*, *fasta*, *fidx*), the *JAR* will test all generated transcript sequences (Step **(C)**) by comparing them to the correct sequences of the transcriptome. If a mismatch occurs, the *JAR* will also throw a *RuntimeException*, otherwise a debug statement will be printed. This debug case ensures that our *GenomicSequenceExtractor* is working correctly and that the sequences of the transcripts are extracted correctly.

If these two debug tests pass, we ensure that the generated reads and their *GenomicRegionVectors* are correct (based on the provided input files). Using the plots in Section 3, we can also see that the fragment length and starting position of the reads are distributed as expected. Conclusively, we can say that the *Read Simulator* is correct.

2.4. Benchmarking

Steps **(B) - (D)** were benchmarked by executing the *readSimulator JAR* 30 times using the parameters defined in Section 1.

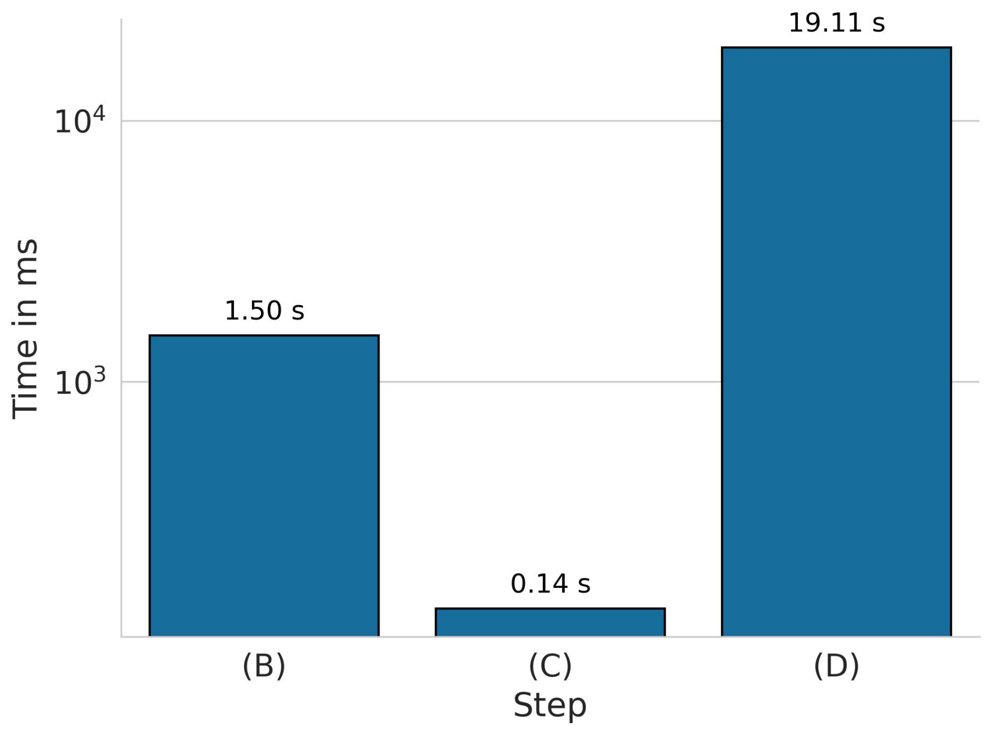

Figure 2 — Mean *JAR* performance in ms after 30 executions measured on *AMD Ryzen 7 PRO 4750U with Radeon Graphics (16) @ 1.700GHz.*

Figure 2 shows us that generating the reads takes significantly longer than the other steps. This was expected since the *JAR* had to simulate 7,777,500 reads per run. As described in Section 2.1, the *generateReads* **(D)** method does several jobs at once (generating reads, mutating them, and writing them). Usually one would prefer to split these tasks into several smaller methods, but due to the vast amount of reads generated, it is really memory intensive to do so. Splitting up *generateReads* would require storing all *Read* objects in a data structure and lead to excessive heap usage.

In Figure 3 and 4 we can see method **(D)** split up into its components based on *CPU time* and *memory allocations*. Note that the component *unknown* in Figure 3 was the fraction of *generateReads*, which was not displayed/identified by the *IntelliJ Profiler*. It is most likely the time needed for garbage collection by *JVM* since each *Read* object and its *readSeq* object are

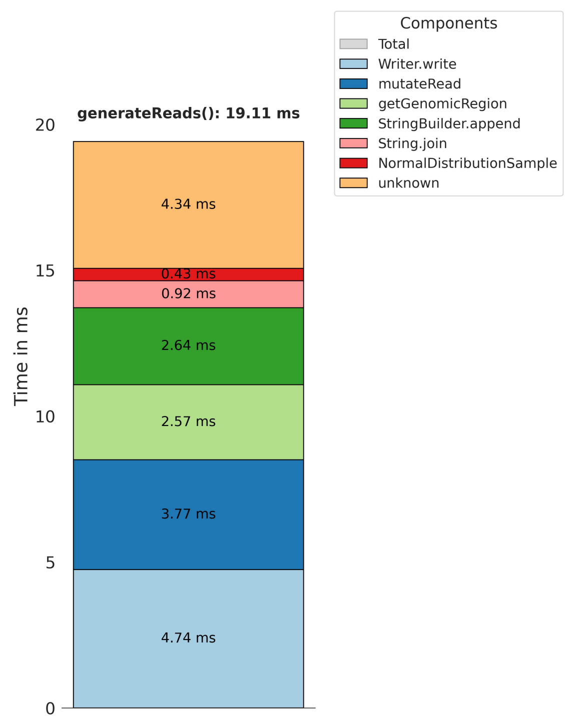

Figure 3 — *generateReads* method analyzed by CPU time using *IntelliJ Profiler*.

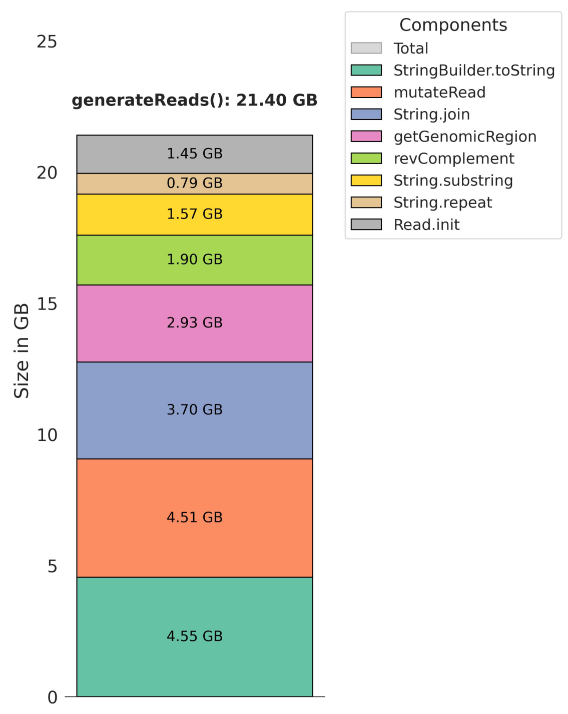

Figure 4 — *generateReads* method analyzed by memory allocations using *IntelliJ Profiler*.

temporary objects that only exist once per iteration. The process of writing takes up the second most considerable fraction in this plot, which is due to the high amount of information being written (around 3.4GB). Mutating the read sequences is unsurprisingly also quite intensive in its time usage. A total of $7,777,500 \cdot 2 \cdot length$ positions need to be mutated ($length ==$ read length), each time generating a new random *double* and potentially needing to draw several bases per position (since mutations like 'A' $\mapsto$ 'A' are not allowed). Because each output entry is built via *StringBuilders*, we call *StringBuilder.toString()* three times per read ($3 \cdot 7,777,500$) and each time this happens, new memory is allocated. A total of $4.55GB$ of allocations can be traced back to this cause. $4.51GB$ is needed to mutate the reads. The method *mutateRead* creates a new *StringBuilder* each time its called and converts said *StringBuilder* to a *String* at the end of the method. Additionally, *mutateRead* also saves all positions mutated as a *String* inside an *ArrayList<String>* of the current *Read* object. Again, since we call this method $2 \cdot 7,777,500$ times and the *StringBuilders* are temporary, the memory allocations accumulate significantly, resulting in the observed memory usage of $4.51GB$.

3 — Results

Figure 5 plots the length of all generated fragments against their total amount. Since a fragment can't be shorter than our specified read length, we have a cutoff at a length of 75. As expected, the fragment length follows a normal distribution with the parameters.

$$\mathcal{N}(frlength, \sigma).$$

The *standard deviation* σ is also provided by the user (see Section 1) and in this case, $\sigma = 80$.

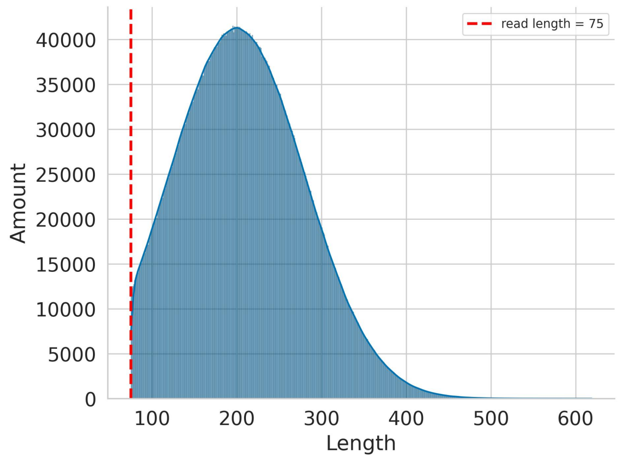

Figure 5 — Fragment length (*frlength*) distribution across all fragments.

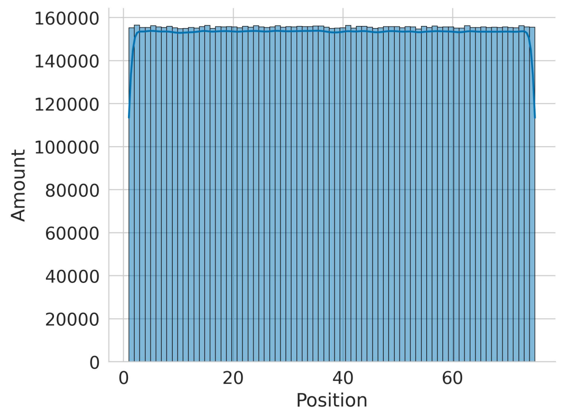

Figure 6 — Distribution of mutation positions inside all reads.

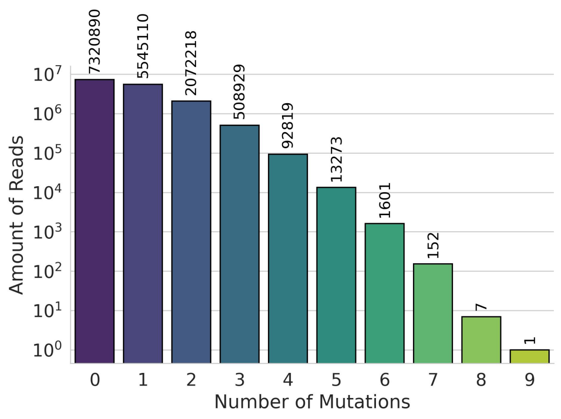

Figure 7 — Number of mutated bases inside read sequences.

The distribution of mutated positions inside the generated reads seems to be uniformly distributed (Figure 6). Each position has the same probability of mutating (in this case: `-mutationrate 1.0` $= 1\%$). Most reads have between zero and three mutated bases (Figure 7); only around 100,000 have more than three mutations, with one outlier having nine mutations.

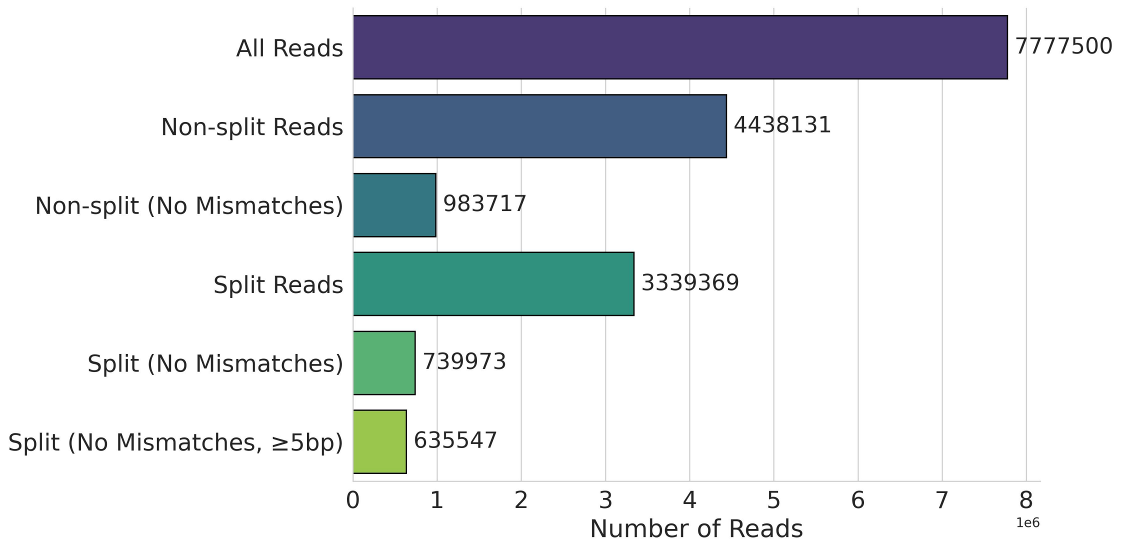

Figure 8 — Breakdown of total reads into split and non-split categories, including subsets with no mismatches or specific thresholds.

Figure 8 shows additional read feature insights. 57% of the generated reads are contained within only one exon (*Non-split Reads*). Since, in our use case, the *frlength* (200) is relatively short, it is a likely scenario. Calculating the mean exon length of the *GTF-File* used in this report also validates this assumption:

```
for(Gene g : genome.getGenes().values()) {
    for (Transcript t: g.getTranscriptList()) {
        for (Exon e: t.getExonList()) {
            totalLength += e.getLength();
            totalExons++;
        }
    }
}
System.out.println(totalLength / totalExons);
>>> 256
```

Because there are less *Split Reads* than *Non-split Reads*, the number of observed *Split (No Mismatches)* is also smaller that *Non-split (No Mismatches)*.