Universität Greifswald

Bachelor thesis

An Online Algorithm for Drawing Optimal Samples from RNA-Seq Libraries

Lukas Willy Bruhn

First Supervisor
Prof. Dr. Stanke
Second Supervisor:
Prof. Dr. Hellmuth

March 20, 2017

Contents

Abstract

Current gene-finders such as AUGUSTUS [13] often take RNA-Seq data as additional evidence. RNA-Seq databases such as the SRA [7] of the NCBI are growing rapidly. However many of the runs submitted to the NCBI are very similar and hence tend to result in only little additional benefit for the application of gene-finding.

With this thesis I offer a software-tool called VARUS that aims to download only a portion of all the available runs in a stepwise manner. At each step VARUS makes estimations about how beneficial further downloads of the available runs are, and then chooses the run that is expected to yield the most improvement, downloads reads from that run (with fastq-dump [3]) and evaluates the resulting alignments (made with STAR [8]). I present a mathematical model to describe this process and present the different estimation strategies implemented in VARUS. VARUS comes with an option to simulate this process as well. It is possible to create completely arbitrary test-scenarios as well as creating simulations that mirror runs of real downloads. This option is used for testing purposes since the download step and alignment step can be very time-consuming. VARUS was tested on different test scenarios with the focus on the different estimation strategies. VARUS is also stable for downloading real runs. However it appears that many more tests need to be done in order to get a feel for how the parameters need to be tweaked.

VARUS is freely available under the GNU-license at `https://github.com/WillyBruhn/VARUS`. A tutorial can also be found there.

1 Introduction

The *genome* is the genetic material of an organism. It consists of DNA, which is present as a *double helix* in its natural form. The two strands that form the double-helix consist of two *complementary* sequences of the bases adenine(A), thymine(T), cytosine(C) and guanine(G). A complementary strand has (A) replaced by (T) and (C) replaced by (G) and vice versa.

A *gene* is a section of the DNA-sequence. As an important example a gene can hold the information necessary for the synthesis of a protein. For the synthesis of a protein the first step is the transcription of the DNA into the pre-mRNA, by making a complementary copy of one strand of the DNA. After transcription the pre-mRNA is spliced: Some parts, called introns, are cut out of the pre-mRNA, the parts that are left, called exons, are then stitched together to form the processed mRNA. The mRNA is then translated into proteins. The whole process of transcription, splicing and translation is called *gene-expression.*

1.1 Genome Analysis

The analysis of the genome of a specific organism consists of multiple steps including DNA sequencing, assembly, gene annotation and analysis. Since current sequencing techniques are only capable of sequencing sequences of up to about 1000 bases, the whole sequence is randomly split into smaller parts. This is called shotgun-sequencing. These smaller parts, called reads, are then sequenced. The resulting reads are then assembled, that means aligned and merged to reconstruct the original sequence. Since shotgun-sequencing is a random sampling process, oversampling is necessary to ensure a given nucleotide is represented in the reconstructed sequence. The result of *DNA sequencing* delivers the exact order of the bases for a given DNA sequence. Since the two strands of the double helix are complementary, it is sufficient to only determine the order for one strand. In bioinformatics a DNA sequence is represented by a sequence consisting of the letters A,C,T and G. In the last two decades the genomes of many species have been completely sequenced.

Gene annotation is the process of attaching biological information to the assembled sequence and includes: Finding portions of the genome that do not code for proteins, identifying specific elements on the genome, a process called *gene prediction* and attaching biological information to these elements.

This thesis was developed in order to improve the selection of input files for AUGUSTUS [13] [12], a gene finder developed and maintained by the workgroup of Prof. Dr. Mario Stanke, who is my supervisor.

1.2 Gene Prediction

Gene prediction is the process of identifying regions in the genome that encode genes. It is the first and most important step in genome annotation after the DNA has been fully sequenced. With *ab initio* approaches the intrinsic information of the genome is used, that means it is searched for signs of protein-coding genes. *Homology-based* approaches make use of the knowledge of the genes of ancestors of the target species. This information is exploited in mainly two ways, through protein spliced alignments and through comparative gene prediction [9]. Protein spliced alignments use either a single protein sequence or a representation of a protein family whereas in comparative gene prediction a genome sequence is used as input.

Additional evidence comes from RNA-Seq. In an *RNA-Seq experiment* the *transcriptome*, meaning all mRNA in a cell currently present, is sequenced with shotgun sequencing. The resulting reads are then mapped to the genome for the purpose of gene-prediction. Only regions of the genome that are capable to be transcribed into mRNA have reads mapping to them. This way one is able to distinguish coding from non-coding regions. However often a small amount of uncertainty about the correctness of the read mappings remains. The aligned RNA might possibly originate from another part of the genome. The uncertainty in the correctness of the mappings can be outweighed by multiple RNA-sequences mapping to a certain region. If multiple RNA-sequences are mapping to a certain region, one can more confidentially assume that this region is an exon. On the other hand the additional benefit of RNA-sequences mapping to that region, decreases as more reads map to that region. A RNA-sequence mapping to a region with only very few aligned RNA-sequences yet is more informative towards finding a correct gene structure, than a RNA-sequence mapping to a region with a lot of RNA-sequences.

In this thesis this problem is addressed by introducing a score function (subsection 2.6), that assigns a score to each run, based on how informative the reads in the run are.

1.3 Drawing Optimal Samples from RNA-Seq Libraries

Which genes are expressed in a specific cell depends on the tissue type and on the outer circumstances. Usually only a fraction of all genes is expressed. Additionally different *alternative splicing* forms can occur in eukaryotes. Alternative splicing describes a set of mechanisms that lets the cell produce multiple different mRNAs from just one gene during splicing. During this process exons can be skipped or rearranged. This makes gene-finding additionally challenging.

A *RNA-Seq-run* is a collection of reads derived in an RNA-Seq experiment. There are several large online-databases of RNA-Seq runs which can be downloaded freely including the *NCBI(National Center for Biotechnology Information)* [7] and

ENA(European Nucleotide Archive) [2]. However the distribution of the reads among the genome, when aligning all reads in a run, is often very similar to other runs. We will say that runs that have a similar distribution form a *library*. The benefit of downloading runs from the same library decreases as more runs are downloaded as described in subsection 1.2.

This circumstance and the fact that these databases are quite large and keep growing, suggests that downloading all reads from all runs is not always an option, or will not be an option in the future.

In this thesis a software-tool called *VARUS* is presented, that aims to download only a fraction of all the available reads of all runs in such a way, that ideally reads from all available libraries are downloaded. These reads are then mapped to the genome using STAR [8].

1.4 An Online Algorithm

An *online algorithm* is an algorithm, that solves a task for which not all input information is available from the start, but the input is continuously added during execution. With each download the information about the runs increases and at each step the algorithm chooses the best available run based on the current information status.

A typical run holds a couple of million reads. These runs are downloaded stepwise in smaller parts, which we will call *batches*. VARUS uses a fixed `batchSize`, although dynamically changing this parameter over time could also be reasonable, because each download comes with a base cost. Therefore downloading the same reads split among multiple downloads can be more expensive than downloading a single larger batch.

At each step the run which maximizes the expected benefit is chosen to be downloaded from. Then a batch of reads is downloaded and the reads are mapped to the genome. Since the chromosomes within a genome can vary drastically in length, it is naturally to be expected that a larger chromosome has more reads mapping to it. By dividing the chromosomes into *blocks* of equal size, we hope to reduce this effect. In VARUS the chromosomes are divided into blocks of a size specified with the parameter `blockSize`. The last block has the length of the rest of the division of the length of the chromosome divided by the `blockSize`. After reads are aligned to the genome, it is then possible to determine a specific block for each read mapping to the genome. For each block of the genome the number of reads mapping to it can then be written in a vector $c = (c_1, \cdots, c_T)^\intercal$. We will call c the observations of a run.

Based on the observations for each run, we can formulate estimations on how further reads from this run will be mapping to the genome. Based on these estimations the next run is then chosen. It should be noted that VARUS should be easily adjustable to use a transcriptome instead of a genome. VARUS is designed for downloading the runs available at the NCBI [7]. The adjustments necessary for downloads from other databases should however not be too difficult. VARUS is implemented in C++ and is freely available at git-hub `https://github.com/WillyBruhn/VARUS`. The input is a list of available Runs at the NCBI for a given species. VARUS uses the sra-toolkit fastq-dump to control the download and the alignment-tool STAR [8] to align the reads. Theoretically the program can be run until all reads have been downloaded and aligned. However, this extreme case would defeat the purpose of VARUS. VARUS was tested on small simulated examples and on the runs of Drosophila *melanogaster* available at the NCBI [7].

2 Model

2.1 The Casino Example

The download problem can be imagined as the following game in a casino. Suppose the casino has a number N of T-sided dice. The numbers that face up when throwing a die are not necessarily uniformly distributed. Each die has it's own distribution, which is unknown to the player. We refer to the distributions as *hidden distribution.* Each turn the player has the option to pay a fixed price and choose a die. The bank then throws this die b times. The player receives money based on the observations produced by these throws and the observations he had before. The less often an outcome had been observed in previous turns, the more money the player receives. At the beginning of a turn the player may choose to end the game. It is a reasonable choice to stop buying dice, if the player expects to lose money, meaning that the price for one buy exceeds the expected win.

Consider the following example: Here we have two possible dice each with two sides:

$$p_1 = \big(0.5, 0.5\big)\,,$$
$$p_2 = \big(0, 1.0\big)\,.$$

The first die is a "fair coin" the second die is a completely "unfair coin" producing only one side. In this scenario choosing only the first die is superior to a strategy, where the dice are chosen at random. This example is covered in more detail in subsection 5.1.

How well a player does in this game, depends on how accurate he estimates the outcomes of further throws with the dice. The best estimation strategy heavily depends on the prior knowledge of the dice. Assumptions could be: all dice follow the same distribution, all dice produce only one of the T numbers, all dice are completely fair. Each of this assumptions would yield a completely different strategy. In this chapter, we will first present different estimation strategies suitable for different scenarios which we think might model the runs from the NCBI.

2.2 Multinomial Distribution

Let $b, T \in \mathbb{N}_0$ and let $p_1, \cdots, p_T \in [0,1]$ with $\sum_{j=1}^{T} p_j = 1$. The probability mass function $f : {\mathbb{N}_0}^T \longrightarrow [0,1]$ of the multinomial distribution $M(b, (p_1, \cdots, p_T)^\intercal\ )$ is then given as

$$f(x_1,\dots,x_T) = \begin{cases} \dfrac{b!}{\prod_{j=1}^{T} x_j!} \prod_{j=1}^{T} p_j^{x_j}, & \text{when } \sum_{j=1}^{T} x_j = b \\ \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

2.3 Parameters

Let

N $\in \mathbb{N}$ be the number of available runs,

T $\in \mathbb{N}$ be the number of blocks that we divided the genome into,

b $\in \mathbb{N}$ be the batch-size, meaning the number of reads that are downloaded at once.

Downloading a batch from a run implies, that this batch has not been downloaded before.

2.4 Observations

For $k = 1,\dots,N$ let $p^k \in [0,1]^T$ with $\sum_{j=1}^{T} p_j^k = 1$ and $(X_n^k)_{n\in\mathbb{N}}$ be a sequence of independently identically $M(b,p^k)$ distributed random-variables. Further, for $n \in \mathbb{N}$ let $C_n^k := \sum_{i=1}^{n} X_i^k$, then $(C_n^k)_{n\in\mathbb{N}}$ are sequences of $M(b\cdot n, p^k)$ distributed random-variables.

Let $n_1,\dots,n_N \in \mathbb{N}$, for $k = 1,\dots,N$ let $(x_1^k,\dots,x_{n_k}^k)$ be realizations of $(X_1^k,\dots, X_{n_k}^k)$ and let $c_{n_k}^k$ be the resulting realizations of $C_{n_k}^k$. It follows

$$c_{n_k}^k = \sum_{n=1}^{n_k} x_n^k = \sum_{n=1}^{n_k} \begin{pmatrix} x_{n,1}^k \\ \vdots \\ x_{n,T}^k \end{pmatrix} = \begin{pmatrix} x_{1,1}^k \\ \vdots \\ x_{1,T}^k \end{pmatrix} + \dots + \begin{pmatrix} x_{n_k,1}^k \\ \vdots \\ x_{n_k,T}^k \end{pmatrix} = \begin{pmatrix} c_{n_k,1}^k \\ \vdots \\ c_{n_k,T}^k \end{pmatrix} \in \mathbb{N}_0^T.$$

Let ${c_{n_k}^k}^* := \sum_{j=1}^{T} c_{n_k,j}^k = b \cdot n_k$.

$n_1,\dots,n_N$ are the number of batches that have been downloaded from each run respectively. The realizations $(x_1^k,\dots,x_{n_k}^k)$ are the first n_k downloaded batches of run k. $c_{n_k}^k$ is the sum of the first n_k downloaded batches in run k. ${c_{n_k}^k}^*$ denotes the number of the downloaded reads in run k. Note that in this model the finiteness of the runs is not modelled. This however is no problem as we can just remove a run from the available runs in our implementation, if it is completely downloaded at any point of execution.

2.5 Total Observations

Let

$$c := \sum_{k=1}^{N} c_{n_k}^k = \begin{pmatrix} c_{n_1,1}^1 \\ \vdots \\ c_{n_1,T}^1 \end{pmatrix} + \ldots + \begin{pmatrix} c_{n_N,1}^N \\ \vdots \\ c_{n_N,T}^N \end{pmatrix} = \begin{pmatrix} c_1 \\ \vdots \\ c_T \end{pmatrix} \in \mathbb{N}_0^T.$$

Let $c^* := \sum_{j=1}^{T} c_j = \sum_{k=1}^{N} {c_{n_k}^k}^* = b \cdot \sum_{k=1}^{N} n_k$.

c is the vector that contains the number of reads that have been observed in each block in all runs combined. c^* is the total number of reads which have been downloaded.

2.6 Score Function

Let $S\colon \mathbb{R}_0^T \to \mathbb{R}$ be a positive function, called the score-function S. With a given total observation c we calculate scores for observations x and x' by calculating $S(x+c)$ and $S(x'+c)$ respectively. This way we are able to compare the benefit of adding any observations x or x' to the total observations.

For the application of gene-finding a higher coverage increases the accuracy of the gene prediction. However the benefit of additional coverage decreases with increasing coverage. In practice the point where adding reads to a certain block yields no further improvement is often reached, when runs are large or numerous. We therefore choose a monotonically increasing function S with negative second derivative. That means that runs that have a high chance of obtaining reads, that are mapping to blocks, which have a otherwise low coverage, are preferred in this scoring-scheme. Note that it is relatively easy to implement another such scoring function in VARUS. We define

$$S(x) := \sum_{j=1}^{T} \log(x_j + 1). \tag{2}$$

The logarithm has this above described behaviour. We take $\log(x_j + 1)$ instead of $\log(x_j)$ because $\log(0) = -\infty$ is bad to handle and $\log(1) = 0$, which means $S\big((0, \ldots, 0)\big) = 0$.

The process of downloading batches from the runs is a stepwise online procedure. It should also be remarked that the outcome of X_{n_k+1}, meaning the next batch of run k is unknown at the time when making the decision whether batch n_{k+1} should be downloaded. In each step we determine a run whose next batch is expected to yield the greatest score. That means that we are interested in downloading a batch from a run that maximizes the expected value of S after sampling one batch from the k-th run:

$$\underset{1 \leq k \leq N}{\arg\max} \quad E(S(X_{n_k+1} + c)). \tag{3}$$

Since calculating $E(S(X_{n_k+1} + c))$ with the multinomial distribution (1) is very complex, we instead maximize (4) as an approximation to (3)

$$\begin{aligned} &\underset{1\leq k\leq N}{\arg\max} \quad S(E(X_{n_k+1} + c)) \\ &= \underset{1\leq k\leq N}{\arg\max} \quad S(p^k \cdot b + c). \end{aligned} \tag{4}$$

In our case the p^k are unknown parameters. In order to determine the run that maximizes (4), we need to estimate p^k. We will denote the estimations of p^k as $\hat{p}^k$, and use this estimation to calculate the score as

$$k' := \underset{1\leq k\leq N}{\arg\max} \quad S(\hat{p}^k \cdot b + c). \tag{5}$$

k' is the index of the run that is expected to yield the greatest improvements to the gene prediction at a certain step in the online algorithm.

3 Estimators

In order to calculate k' we need to define and calculate $\hat{p}^k$. In this section we present different estimators that make different assumptions on how the reads are distributed in the runs.

3.1 Maximum Likelihood Estimator

Suppose the runs are unrelated to each other. This would be the simplest assumption. Suppose ${c^k_{n_k}}^* > 0$, then the maximum likelihood estimator for $j \in 1, \cdots, T$ can be derived as:

$$\hat{p}^k_j := \frac{c^k_{n_k,j}}{{c^k_{n_k}}^*}.$$

We complement this estimator for the case of a run with no observations as follows:

$$\hat{p}^k_j := \begin{cases} \frac{c^k_{n_k,j}}{{c^k_{n_k}}^*} & \text{if } {c^k_{n_k}}^* > 0 \\ \frac{1}{T} & \text{else} \end{cases}, \quad \hat{p}^k = \begin{pmatrix} \hat{p}^k_1 \\ \vdots \\ \hat{p}^k_T \end{pmatrix}.$$

A run with no observations will be estimated to follow a uniform distribution. The estimation is inaccurate for small batch sizes, as the probability for a block with no reads mapping to it, is estimated as 0. This estimator can be used in VARUS by setting the parameters `estimator = 1` and `pseudoCount = 0`.

3.2 Simple Estimator

In order to compensate the disadvantage of the maximum-likelihood-estimator to estimate inaccurately for small batch sizes, we add a pseudo-count to each observation. Let $a \in \mathbb{R}_+$, we define $\hat{p}_j^k$ as:

$$\hat{p}_j^k := \frac{c_{n_k,j}^k + a}{{c_{n_k}^k}^* + a \cdot T}.$$

This estimator can be seen as a mixture between the actual observations of a run and a uniform distribution. Since a is a constant, this estimator approaches the maximum likelihood estimator with a growing number of observations:

$$\lim_{n_k \to \infty} \frac{c_{n_k,j}^k + a}{{c_{n_k}^k}^* + a \cdot T} - \frac{c_{n_k,j}^k}{{c_{n_k}^k}^*} = 0.$$

That means for small numbers of reads the estimation will lean more towards the uniform distribution and with a growing number of reads the estimation approaches the maximum likelihood estimator. This estimator can be used in VARUS by setting the parameters `estimator = 1` and `pseudoCount` accordingly.

3.3 Advanced Estimator

In the first estimator we made the assumption that the runs are independently distributed, and hence estimated the runs based on their observations independently of the observations of other runs. With the next estimators we want to model our knowledge of the libraries. The libraries submitted to the NCBI are often very similar and many experiments are done under similar conditions. Suppose the number of libraries is relatively small, meaning that many runs come from the same library and hence are similarly distributed. Let $m < N$ with unknown $(X'^1, \ldots, X'^m)$, such that for all $v \in 1, \cdots, N$ there exists $u \in 1, \cdots, m$ with X^v equally distributed as X'^u. These further assumptions motivate the following estimators.

Let $\hat{p}$ be the estimator based on c (total-observations) defined as:

$$\hat{p}_j := \begin{cases} \frac{c_j}{c^*} & \text{if } c^* > 0, \\ \frac{1}{T} & \text{else} \end{cases}, \quad \hat{p} = \begin{pmatrix} \hat{p}_1 \\ \vdots \\ \hat{p}_T \end{pmatrix}.$$

Let $\lambda \in \mathbb{R}_+$, we define $\hat{p}^k$ as follows:

$$\hat{p_j}^k := \frac{c_{n_k,j}^k + a + \lambda \cdot \hat{p}_j \cdot T}{{c_{n_k}^k}^* + a \cdot T + \lambda \cdot T}.$$

This estimator takes also into account the observations of all other runs. Runs with no reads yet or only few reads are then estimated to be somewhat similar to the mixture of all other runs.

Again since a and λ are constants, this estimator approaches the maximum likelihood estimator with a growing number of observations:

$$\lim_{n_k \to \infty} \frac{c^k_{n_k,j} + a + \lambda \cdot \hat{p}_j \cdot T}{{c^k_{n_k}}^* + a \cdot T + \lambda \cdot T} - \frac{c^k_{n_k,j}}{{c^k_{n_k}}^*} = 0 \quad .$$

This estimator can be used in VARUS by setting the parameters `estimator = 2` and `pseudoCount`, `lambda` accordingly.

3.4 Dirichlet Mixture

In the advanced estimator we assumed that the runs are similarly distributed. However the influence of each library was mixed together in only one parameter λ. With this estimator we hope to model multiple libraries.

Most of the mathematical equations in this section are adapted from the two articles [14] and [11], where the dirichlet mixture was used in the context of protein-alignments. The parameters in VARUS that refer to this estimator are:

```
estimator = 3, components, exportNewtons, newtonPrecision,
newtonIterations, trainingsIterations, simpleDM.
```

3.4.1 Gamma Function

For complex numbers with positive real part the *gamma function* is defined as:

$$\Gamma(z) := \int_0^\infty x^{z-1} e^{-x} \, \mathrm{d}x.$$

For positive integers z this leads to:

$$\Gamma(z) = (z-1)!.$$

3.4.2 Dirichlet Distribution

Let $q := \left(q_1, \cdots, q_T\right) \in [0,1]^T$ be a vector with $1 = \sum_{j=1}^T q_j$. Let $\alpha^* > 0 \in \mathbb{R}$. The density $\rho : \{y \in \mathbb{R}^T | 0 \leq y_j \leq 1 \wedge \sum_{j=1}^T y_j = 1\} \to \mathbb{R}_{\geq 0}$ of the *dirichlet distribution* is then defined as

$$\rho(y) := \frac{\Gamma(\alpha^*)}{\prod_{j=1}^T \Gamma(\alpha^* q_j)} \prod_{j=1}^T y_j^{\alpha^* q_j - 1} \tag{6}$$

q is called the *location parameter*, α^* is called the *concentration parameter*. The probability mass function of the multinomial distribution $M(b, (p_1, \cdots, p_T)^\intercal)$ can be expressed using the gamma function as:

$$\rho(x_1, \ldots, x_T) = \frac{b!}{\prod_{j=1}^T x_j!} \prod_{j=1}^T p_j^{x_j} = \frac{\Gamma(\sum_{j=1}^T x_j + 1)}{\prod_{j=1}^T \Gamma(x_j + 1)} \prod_{j=1}^T p_j^{x_j}.$$

This form shows its resemblance to the density of the dirichlet distribution (6).

3.4.3 Dirichlet Mixture

Let $\ell \in \mathbb{N}$ and let $\rho_1, \cdots, \rho_\ell$ be dirichlet distributions with parameters $((q^1, \alpha_1^*), \cdots, (q^\ell, \alpha_\ell^*))$. Let $m_1, \cdots, m_\ell \in [0, 1]^\ell$ with $1 = \sum_{i=1}^\ell m_i$. The density of the *dirichlet mixture* is then defined as

$$\rho = m_1\rho_1 + \cdots + m_\ell\rho_\ell.$$

Each density in the mixture is called a component of the mixture. $m_1, \cdots, m_\ell$ are called the weights of the components or mixture coefficients.

We define $\theta := \big((m_1, q^1, \alpha_1^*), \cdots, (m_\ell, q^\ell, \alpha_\ell^*)\big)$ as the set of parameters that define a Dirichlet-mixture.

3.4.4 Interpretation of the Dirichlet Distribution and Mixture

The dirichlet distribution can be seen as a run factory that produces runs with a certain multinomial distribution. However the actual distribution of the runs varies around this distribution. The higher the concentration parameter α^* the smaller the variance. For α^* approaching ∞ the dirichlet distribution approaches the multinomial distribution $M(b, (q_1, \cdots, q_T)^\intercal)$.

If we were to cluster the runs based on their observations in such a way, that after clustering similarly distributed runs were in the same cluster, we could estimate p for a given run based on the observations of all runs in the same cluster. For each distinct cluster we could use one distinct dirichlet distribution as it is done in subsubsection 3.4.10.

With a dirichlet mixture it might be possible to refine this estimation. Instead of letting each component of the mixture represent exactly one library, we model each library by a mixture of the components. By using a mixture we are able to model many more libraries with a smaller amount of components. This is comparable to taking the elements in the powerset of a set, instead of only taking the elements in the set.

3.4.5 Basic Procedure

This estimation process consists of multiple parts. The first part is *training* the dirichlet mixture through *Gibbs-Sampling.* Training hereby means that we determine θ, which defines the dirichlet mixture. For the gibbs-sampling in particular we need to estimate the concentrations parameters α_i^* at each iteration of the training process. For calculating α^* that maximizes the corresponding likelihood-function, we use *Newton's method.* In the second part the observations for further downloads are estimated with this mixture.

3.4.6 Training The Dirichlet Mixture through Gibbs-Sampling

For VARUS the algorithm presented in *On the Inference of Dirichlet Mixture Priors for Protein Sequence Comparison*(4.1 A Gibbs-sampling-strategy) [14] was taken. We take a fixed number l of components, specified with the parameter `components`, and initialize each component with a uniform distribution. We calculate for each component the likelihood for the observations in a given run. The run is then randomly assigned to one of the components based on the likelihoods. That means the run is most likely sampled into the bin with the highest likelihood, however it can also be sampled into another component. After all runs are assigned to components, the parameters for each component is calculated based on the runs being sampled into the component in question. The runs are then released and again sampled, based on the likelihoods with now different parameters. This process is repeated until convergence.

In [14] the number of components is also variable and a parameter to be trained. As the training of the correct number of components is very complex, for this thesis only a fixed number l of components was taken. That leads to a slight modification to the algorithm. Transferred to the download-problem and with the notation adapted to this thesis, the pseudocode from [14] looks like this:

1. Create a vector $\vec{m} \in \mathbb{R}^\ell$ with $\vec{m}_i = \frac{1}{\ell}$ for $i \in 1, \cdots, \ell$ and create ℓ empty components, and then for each run k:
   (a) Use $\vec{m}$ and (7) to calculate a likelihood $m_i \phi_i^k$ for each constituent component of θ.
   (b) Normalize these likelihoods, and use them to randomly sample run k into one of the ℓ components. That leads to a partition $M_1, \cdots, M_l$ of the indices $\{1, \cdots, N\}$ of the runs.
2. For each component, calculate parameters for a new dirichlet mixture θ' as follows:

(a) Calculate a new mixture parameter as $m'_i := \frac{|M_i|}{N}$, where $|M_i|$ is the number of runs that have been sampled into component i.

(b) Calculate new location parameters as

$$q'^i_j := \frac{\sum_{k \in M_i} c^k_{n_k,j}}{\sum_{k \in M_i} c^{k\ *}_{n_k}} = \frac{\sum_{k \in M_i} c^k_{n_k,j}}{b \cdot \sum_{k \in M_i} n_k}.$$

The counter is the aggregate count of reads in block j among all runs assigned to component i, and the denominator is the aggregate count of all reads in all runs assigned to component i.

(c) Calculate a new concentration parameter $\alpha^{*\prime}_i$ using the maximum-likelihood procedure described below and return to step 1 a.

The variables marked with a prime symbol $'$ are the variables that form the new values for each iteration.

3.4.7 Sampling

The likelihoods in the Gibbs-sampling-step (1a) are calculated as in [14] as:

$$\phi^k_i = \frac{\Gamma(\alpha^*_i)}{\Gamma(\alpha^*_i + c^{*k})} \cdot \prod_{j=1}^{T} \frac{\Gamma(\alpha^*_i q^i_j + c^k_j)}{\Gamma(\alpha^*_i q^i_j)} \tag{7}$$

3.4.8 Estimation of the Concentration Parameter α^*

α^*_i describes the variance in the i-th component. The following procedure is also taken from [14]. Let ${c^k}_i$ with $k \in 1, \cdots, n_i$ be the observations of the runs sampled into component i. c^{*k}_i is the sum of all observations in run k. According to [14] (7) the log-likelihood $\mathcal{L}$ of the data is then given by:

$$\mathcal{L} = \sum_{k \in M_i} \log[\frac{\Gamma(\alpha^*_i)}{\Gamma(\alpha^*_i + c^{*k})} \cdot \prod_{j=1}^{T} \frac{\Gamma(\alpha^*_i q^i_j + c^k_j)}{\Gamma(\alpha^*_i q^i_j)}] \tag{8}$$

$$= \sum_{k \in M_i} \{\log(\Gamma(\alpha^*_i)) - \log(\Gamma(\alpha^*_i + c^{*k})) + \sum_{j=1}^{T} [\log(\Gamma(\alpha^*_i q^i_j + c^k_j)) - \log(\Gamma(\alpha^*_i q^i_j))]\} \tag{9}$$

According to [14] q^i_j can then be estimated by

$$\hat{q}^i_j = \frac{\sum_{k \in M_i} c^k_j}{\sum_{k \in M_i} c^{*k}_j} \tag{10}$$

and q_j^i can be replaced in (9) by $\hat{q}_j^i$. This leaves the single parameter α_i^* still to estimate. We want to take the parameter α_i^* that maximizes (9). As in [14] we use *Newton's method* to find the parameter. To apply Newton's method, we need the first and second derivatives of $\mathcal{L}$ with respect to α_i^*. According to [14] the first and second derivatives can be written as:

$$\frac{\partial \mathcal{L}}{\partial \alpha_i^*} = \sum_{k \in M_i} \{\psi(\alpha_i^*) - \psi(\alpha_i^* + c^{*k}) + \sum_{j=1}^{T} \hat{q}_j^i [\psi(\alpha_i^* \cdot \hat{q}_j^i + c_j^k) - \psi(\alpha_i^* \cdot \hat{q}_j^i)]\} \tag{11}$$

$$\frac{\partial^2 \mathcal{L}}{\partial^2 \alpha_i^*} = \sum_{k \in M_i} \{\psi'(\alpha_i^*) - \psi'(\alpha_i^* + c^{*k}) + \sum_{j=1}^{T} (\hat{q}_j^i)^2 [\psi'(\alpha_i^* \cdot \hat{q}_j^i + c_j^k) - \psi'(\alpha_i^* \cdot \hat{q}_j^i)]\} \tag{12}$$

where ψ and ψ' are the digamma and the trigamma functions. According to [14] a good start-value is critical. With too small or big values Newton's method does not converge against the null point. According to [14] a start value for Newton's method can be calculated as:

$$c_i^* := \sum_{k \in M_i} c^k, \quad c_i^{*2} := \left((c_{i\,1}^*)^2, \cdots, (c_{i\,T}^*)^2\right),$$

$$d_j := \left(c_j^1, \cdots, c_j^N\right), \quad v_j^i := \frac{\hat{\text{Var}}(d_j)}{\hat{E}^2(c_i^*)} - \frac{\hat{\text{Var}}(c_i^*)}{\hat{E}^2(c_i^*)} \cdot (\hat{q}_j^i)^2,$$

$$(\hat{\alpha}^*)_j^i := \frac{\hat{q}_j^i (1 - \hat{q}_j^i) \cdot \frac{\hat{E}(c_i^{*2})}{\hat{E}^2(c_i^*)} - v_j^i}{v_j^i - \frac{\hat{q}_j^i \cdot (1 - \hat{q}_j^i)}{\hat{E}(c_i^*)}}, \tag{13}$$

$$\hat{\alpha}_i^* = \sum_{j=1}^{T} \hat{q}_j^i (\hat{\alpha}^*)_j^i \quad \text{for } j \in 1, \cdots, T \text{ and for } i \in 1, \cdots, \ell. \tag{14}$$

Newton's method is initialized with $\hat{\alpha_i^*}$.

3.4.9 Mixture Estimation

After training, meaning after θ is determined the run k is estimated with the mixture after [11] as follows:

$$\hat{p}_j^k := \frac{Z_j}{\sum_{j=1}^{T} Z_j} \tag{15}$$

with

$$Z_j := \sum_{i=1}^{l} m_i \cdot \exp(\log(B(\alpha_i^* q^i + c^k)) - \log(B(\alpha_i^* q^i))) \cdot \frac{\alpha_i^* q_j^i + c_j^k}{\alpha_i^* + c^{*k}} \tag{16}$$

with

$$\log(B(x)) := \sum_{j=1}^{T} \Gamma(x_j - \log(\Gamma(x^*))), \quad \text{where } x^* := \sum_{j=1}^{T} x_j. \tag{17}$$

That means on the estimation for each run all components of the mixture have an influence. Components that are more similar to the observations of the runs have more influence.

3.4.10 Single Component Estimation

First tests with the estimation-procedure as described above suggested that the calculation times might be too long. A simpler and less expensive approach for the estimation is to understand each component in the mixture as a cluster. The runs are then only estimated based on the observations in the same component. According to [11] in the case of a single dirichlet distribution the estimations can be calculated as:

$$\hat{p}_j^k := \frac{c_{n_k,j}^k + \alpha_i^* q_j^i}{|c^{*k}| + |\alpha_i^*|}, \quad \text{with } k \in M_i. \tag{18}$$

That means for each run we calculate the estimation based on the component it was sampled into. That means here only the runs in the same component are used for the estimation of a specific run. For a given run, runs in other components have, unlike as in the mixture estimation, no influence on the estimations of the run. This estimation strategy can be used in VARUS by using the option `simpleDM`.

3.5 Cluster Estimator

In contrast to the dirichlet mixture the expensive process of finding a θ is skipped in the cluster estimator. Instead this estimator clusters the runs with observations based on their observations with the *k-means-algorithm.* After similarly distributed runs are in the same cluster, for each cluster an estimation is done in the same way as it was done in the advanced estimator.

This estimator can be used in VARUS by setting the parameters `estimator = 4` and `components`, `trainingsIterations` accordingly.

3.5.1 Clustering

Given a number ℓ of clusters with clustering centers K^i with $i \in 1, \cdots, \ell$ and $K^{*i} = \sum_{j=1}^{T} K_j^i$, the clustering-process consists of the following steps:

1. Calculate $q^k := \left(\frac{c_1^k + a}{c^k * + a \cdot T}, \cdots, \frac{c_T^k + a}{c^k * + a \cdot T} \right)$ for all runs.
2. For each cluster take a randomly chosen k and assign $K^i = q^k$, to initialize the cluster center.
3. For each run calculate the distance to each cluster-center according to (19) and assign each run to the cluster with the closest distance. That leads to a partition $M_1, \cdots, M_l$ of the indices $1, \cdots, N$.
4. For each cluster calculate a new cluster center by averaging over the q-vectors of all runs assigned to that cluster: $K^i = \frac{1}{|M_i|} \sum_{k \in M_i} q^k$.
5. Return to step 2. until K^i doesn't change.

3.5.2 Distance Measure

We modify the *Kullback-Leibler divergence* to make it symmetric as follows:

$$\text{dist}(\text{run}, \text{cluster}) := \sum_{j=1}^{T} \log(\frac{q_j}{K_j}) \cdot q_j + \log(\frac{K_j}{q_j}) \cdot K_j. \quad (19)$$

Note that $q_j, K_j > 0$ for all $j \in 1, \cdots, T$ because of the pseudo-counts we used in step 1.

3.5.3 Scaling λ_i

In resemblance to the concentration-parameter of the dirichlet mixture we want to have a similar effect for this estimator. Therefore λ_i should be greater in a component with less variance and smaller in a component with more variance. Therefore λ_i is calculated as the average of the standard deviations:

$$\lambda_i = \frac{T}{\sum_{j=1}^{T} \sqrt{\frac{1}{|M_i|} \cdot \sum_{k \in |M_i|} (q_j^k - K_j)^2}}. \quad (20)$$

3.5.4 Estimation

The estimation is exactly the same as in the advanced estimator for each cluster. We define $\hat{p}^k$, the estimation of runs in a given cluster i as follows:

$$\hat{p}_j^k := \frac{c_{n_k,j}^k + a + \lambda_i \cdot T \cdot K_j^i}{{c_{n_k}^k}^* + a \cdot T + \lambda_i \cdot T}.$$

Runs with no observations are estimated in the same way as in the advanced estimator. Let $\lambda \in \mathbb{R}_+$, we define $\hat{p}^k$, the estimation of runs with no observations as follows:

$$\hat{p}_j^k := \frac{a + \lambda \cdot T \cdot \hat{p}_j}{a \cdot T + \lambda \cdot T}$$

where

$$\hat{p}_j := \begin{cases} \frac{c_j}{c^*} & \text{if } c^* > 0, \\ \frac{1}{T} & \text{else} \end{cases}, \quad \hat{p} = \begin{pmatrix} \hat{p}_1 \\ \vdots \\ \hat{p}_T \end{pmatrix}$$

is the estimation for all runs combined. Note that the parameter `lambda` is only used for the estimation of the runs with no observations.

4 Online Algorithm

With the model as described in section 2 we can now formulate the pseudocode for the online algorithm. We are interested in downloading reads until we expect that the costs for a further download outweigh the added benefit. Therefore we want to assure that

$$S(c + Run.p \cdot b) - S(c) - cost > 0 \tag{21}$$

at all time, where c refers to the total observations and *Run.p* is the estimator for the run that we want to download next and *cost* is a fixed value, determining the cost for downloading one batch. We refer to the left side of (21) as the *expected profit* of a run. The download continues with a run that maximizes the expected profit. The expected profit of all runs is then updated. Then the run with the highest expected profit is chosen. The chosen run is then downloaded and its reads are aligned to the genome. The alignment is evaluated and the observations of the runs are formed. Then the estimation for all runs is done. This process continues until the expected profit is not greater zero anymore.

The function *chooseNextBatchIndices* sets the indices that are given to fastq-dump in the download step. The reads in a run can be downloaded by setting a lower bound N and an upper bound X. All reads between N and X are then

Algorithm 1 online-algorithm

```
1:  bestRun.profit ← 1
2:  while bestRun.profit > 0 do
3:      for all Runs do
4:          Run.profit ← Score(c + Run.p · b) - Score(c) - cost
5:      bestRun ← chooseNextRun()
6:      if bestRun.profit ≤ 0 then
7:          break
8:      bestRun.batchIndices ← chooseNextBatchIndices(bestRun)
9:      fasta-file ← downloadWithFastqDump(bestRun)
10:     SAM-file ← alignWithSTAR(fasta-file,genome-file)
11:     uniquelyMappingReads ← retrieveResultsFromAlignment(SAM-file)
12:     bestRun.c ← updateObservations(uniquelyMappingReads)
13:     c ← updateObservations(uniquelyMappingReads)
14:     for all Runs do
15:         Run.p ← estimateP(Runs)
16:     CSV-file ← exportToCSV(Runs)
```

Algorithm 2 chooseNextBatchIndices

```
1:  procedure CHOOSENEXTBATCHINDICES(bestRun)
2:      bestRun.N ← bestRun.sigma[bestRun.sigmaIndex] · bestRun.b
3:      bestRun.X ← bestRun.N + bestRun.b - 1
4:      bestRun.sigmaIndex++
```

downloaded. It is not guaranteed that the reads are not ordered in a certain way, e.g. alphabetically. Since we want a good representation of the reads of a run, we therefore do not download all runs beginning with N = 0 and going through them linearly, but instead randomize which parts are downloaded. We do this by dividing the total number of reads in a run by b, this gives us the number of batches that we can download from this run. *sigma* is an array that holds which batch is to be downloaded for each iteration.

The functions *downloadWithFastqDump* and *alignWithSTAR* in line 9 and 10 call fastq-dump and STAR respectively with the needed parameters. The result of a download with fastq-dump is a file in either fasta or fastq-format. STAR also offers to make output-files in BAM-format or SAM-format. In VARUS the SAM-file-format was chosen. Switching to BAM-format would require adaptations to the function *retrieveResultsFromAlignment*.

The function *retrieveResultsFromAlignment* reads in all the read mappings noted in the SAM-file. The reads that map to *one* block exactly *once* are returned as the *uniquelyMappingReads (UMR).* Note that STAR uses the term UMR in a different way. In the log-file output of STAR the UMR refer to reads that map to a *chromosome* exactly once.

The UMR form the observations for each download. With the function *updateObservations* the observations of the bestRun and the total observations c are updated in the following way: for each block the number of reads that mapped to that block is added to the number of reads that mapped to that block in previous downloads.

In function *estimateP* the expected observations for further downloads are estimated for all runs. Here one of the estimators presented in section 3 is called, depending on what the user specified. With the function *exportToCSV* informations about the observations and expected observations are exported into a CSV-file. The level of detail for this output can be adjusted and it should be noted that a high level of detail can be a performance-issue, depending on the number of blocks and the number of runs.

5 Results

In this section we discuss the results achieved with VARUS. Since the downloading and aligning of the runs can be relatively costly VARUS has an option `simulation` to simulate these steps. Also for testing-purposes the possibility of defining arbitrary test-scenarios with an arbitrary number of blocks for the runs is implemented in VARUS. In a simulation for each run a hidden distribution is specified by the user. With the simulation the implementation and the behaviour of the different estimators, described in section 3, are tested. In order to become confident about the implementation and understand how the estimators work, we first study simplified versions of the download-problem. Later we look at real runs of Drosophila *melanogaster*. In the context of simulations unrelated to real runs we use the term 'die' instead of 'run'. A die's size can range from just two fields to an arbitrary size and is specified with the parameter `numOfBlocks`. In the case of Drosophila *melanogaster* with a `blockSize` of 5000 base pairs the number of fields of a die is 29918.

When we use the term 'fair die' we refer to a die with a uniform distribution. In general a *random seed* is used to define the start for a random sequence produced by a random engine. For different tests we use different random seeds. However in order to make the results more comparable, we use the same random seed for each estimator in the same test. Therefore the run choices can be very similar in the first steps of a test and only differ in later iterations. One thing to be remarked is that these tests are of only prototypical nature. In how far the estimators are of any use for real data, is only superficially examined for the case of Drosophila *melanogaster*. Further tests with also other species might reveal which of the estimators is most suitable, if downloading randomly is not sufficient.

5.1 Introductory Example

In this first test we use two dice with the following hidden distributions and parameters:

Test 1:

$$\begin{aligned}
&p_1 = \big(0.5, 0.5\big)\,,\\
&p_2 = \big(0, 1.0\big)\,,\\
&b = 10, cost = 0, maxBatches = 30\\
&a = 1.0, \lambda = 1.0.
\end{aligned}$$

The parameter `maxBatches` specifies the number of times a die can be thrown. In order to maximize the score-function it is obvious that an estimator should only go for die 1. The result is shown in Figure 1. Each column corresponds to a specific estimator. The first column shows the results achieved by picking dice randomly, meaning without actually estimating. The second column shows the results achieved by the simple estimator subsection 3.2. The third column shows the results achieved by the advanced estimator subsection 3.3.

The first line of plots shows the time-line, that means which die was chosen at each iteration. The x-axis is the iteration, the y-axis shows the chosen die. The second line shows how many times each die was chosen. The third line is a histogram of the outcomes. A uniform distribution is the goal here and a sign for a good estimator.

The plot in the bottom of the figure shows how the *profit* developed. The profit is calculated as $Profit(iteration) = S(c) - cost \cdot iteration$.

As one can see, the simple estimator and the advanced estimator yield similar results in this example, which are better than picking runs completely random. Both the simple and the advanced estimator assign a very poor score to die2 after once choosing it and then never choosing it again. One thing to be remarked is: even though the plot for the advanced and the simple estimator is way better, that does not reflect so heavily in the scores in the second line. This is due to the nature of the logarithm. That however means that even small differences in scores result in great differences in the histograms.

5.2 Advanced Estimator Is Better than the Simple Estimator

The previous test showed us that the simple and advanced estimator are better than picking dice randomly. In order to show the advantage of the advanced estimator over the simple estimator, we make a slight modification. We increase the number of unfair dice to a total of twenty-nine dice and keep the one fair die. We use the same parameters:

Test 2:

$$\begin{aligned}
&p_1 = \big(0.5, 0.5\big)\,, \\
&p_i = \big(0, 1.0\big) \quad i \in 2, \cdots, 30 \\
&b = 10, cost = 0, maxBatches = 60 \\
&a = 1.0, \lambda = 0.5.
\end{aligned}$$

The result is shown in Figure 2. The simple estimator keeps choosing the dice he didn't observe yet. That is due to the fact that the estimation for the dice with no observations is $p = \big(0.5, 0.5\big)$, which is approximately the same for the fair die.

The advanced estimator almost always chooses the fair die, once he chooses it once. The reason for this behaviour is that the advanced estimator uses the information from all dice in order to estimate a die with no observations. Therefore the estimator for a die with no observations will differ from $p = \big(0.5, 0.5\big)$ towards an estimator that has a greater value in the second entry. That makes it less profitable since we already have a lot of observations on the second side.

When looking at the plots two things seem remarkable. Why does it take the simple estimator thirty-eight iterations until it identifies die1 as the good die? We would expect that it would take thirty iterations since we have thirty dice. The other question is, why does the advanced estimator choose a bad die in iteration thirty-one, after it chose the fair die for a long time? In order to answer these questions, we will take a closer look at the two estimators in this experiment.

5.2.1 Simple Estimator in More Detail

The first four choices are unfair dice. In the fifth step c_1 is chosen and the observations are $c_1(5) = \big(7, 3\big)$. The total observations therefore are $c(5) = \big(7, 43\big)$ The estimation for the fair die is then calculated as:

$$\hat{p}^1_j := \frac{c^1_{5,j} + a}{c^{1*}_5 + a \cdot 2} \quad j \in 1, 2$$
$$\hat{p}^1 = \big(\tfrac{7+1}{10+1\cdot 2}, \tfrac{3+1}{10+1\cdot 2}\big) = \big(\tfrac{2}{3}, \tfrac{1}{3}\big)\,.$$

And the score for c_1 is calculated as:

$$S(p^1 \cdot b + c) = S(\big(\tfrac{2}{3}, \tfrac{1}{3}\big) \cdot 10 + \big(7, 43\big)) = \log(\frac{20}{3} + 7 + 1) + \log(\frac{10}{3} + 43 + 1) = 2.84.$$

The score for one of the dice with no observations is calculated as:

$$S(p \cdot b + c) = S(\big(\tfrac{1}{2}, \tfrac{1}{2}\big) \cdot 10 + \big(7, 43\big)) = \log(5 + 7 + 1) + \log(5 + 43 + 1) = 2.8.$$

The score for one of the unfair dice which were chosen once is calculated as:

$$S(p \cdot b + c) = S(\big(\tfrac{1}{12}, \tfrac{11}{12}\big) \cdot 10 + \big(7, 43\big)) = \log(\frac{10}{12} + 7 + 1) + \log(\frac{110}{12} + 43 + 1) = 2.67.$$

Because c_1 yields the highest score it is chosen in step six. The next observations are $c_1(6) = \big(4, 6\big)$. With these observations the estimator for c_1 is

$$\hat{p}^1(6) := \big(\tfrac{11+1}{20+2}, \tfrac{9+1}{20+2}\big) = \big(\tfrac{6}{11}, \tfrac{5}{11}\big)$$

Since we still have more observations on the second side of the total observations $(c(6) = \big(11, 49\big))$ it is obvious that dice that are expected to produce more observations on the first side of the the die get a higher score. That is the reason why

c_1 is chosen in step seven as well. In step seven the observation is $c_1(7) = \big(4, 6\big)$. That leads to an estimation of $\hat{p}^1(7) := \big(0.5, 0.5\big)$. That means now the estimation for c_1 is the same as the estimation for any other die that was not chosen yet. That means until all the dice are downloaded once, the simple estimator will go for the good dice equally as likely as for the bad dice. In this example the last unfair die is chosen in step thirty-eight. After this step the estimator will only go for c_1.

5.2.2 Advanced Estimator in More Detail

The first four choices are unfair dice. In the fifth step c_1 is chosen and the observations are $c_1(5) = \big(7, 3\big)$. The total observations therefore are $c(5) = \big(7, 43\big)$ The estimation for the fair die is then calculated as:

$$\hat{p_j}^1 := \frac{c_j^1 + a + \lambda \cdot \hat{p}_j \cdot T}{c_5^{1*} + a \cdot 2 + \lambda \cdot T} \quad j \in 1, 2.$$

$$\hat{p}^1 = \Big(\tfrac{7+1+\frac{7}{50}}{10+1\cdot 2+1}, \tfrac{3+1+\frac{43}{50}}{10+1\cdot 2+1}\Big) = \big(\tfrac{407}{650}, \tfrac{243}{650}\big) \; = \big(0.626, 0.374\big)\,.$$

And the score for c_1 is calculated as:

$$S(p^1 \cdot b + c(5)) = S(\big(\tfrac{407}{650}, \tfrac{243}{650}\big) \cdot 10 + \big(7, 43\big)) = \log(\frac{407}{65} + 7 + 1) + \log(\frac{243}{65} + 43 + 1) = 2.83.$$

As you can see this score is a little bit lower than the score that was assigned by the simple estimator in the fifth step. That is due to the fact that this estimator is leaning a little bit more to the second side because of λ.

The estimation for one of the dice with no observations is calculated as:

$$\hat{p^i} := \Big(\tfrac{0+1+\frac{7}{50}}{10+1\cdot 2+1}, \tfrac{0+1+\frac{43}{50}}{0+1\cdot 2+1}\Big) = \big(\tfrac{19}{50}, \tfrac{31}{50}\big) = \big(0.38, 0.62\big)\; i \in 2, ..., 30.$$

The score for one of the dice with no observations is calculated as:

$$S(p \cdot b + c) = S(\big(\tfrac{19}{50}, \tfrac{31}{50}\big) \cdot 10 + \big(7, 43\big)) = \log(\frac{19}{5} + 7 + 1) + \log(\frac{31}{5} + 43 + 1) = 2.77.$$

The estimation for one of the unfair dice with one observation is calculated as:

$$\hat{p^i} := \Big(\tfrac{0+1+\frac{7}{50}}{10+1\cdot 2+1}, \tfrac{10+1+\frac{43}{50}}{10+1\cdot 2+1}\Big) = \big(\tfrac{57}{650}, \tfrac{593}{650}\big) \; = \big(0.088, 0.912\big)\; i \in 2, ..., 30.$$

The score for one of the unfair dice with one observation is calculated as:

$$S(p \cdot b + c) = S(\big(\tfrac{57}{650}, \tfrac{593}{650}\big) \cdot 10 + \big(7, 43\big)) = \log(\frac{57}{65} + 7 + 1) + \log(\frac{593}{65} + 43 + 1) = 2.67.$$

To sum this up one can say that the simple estimator will first choose all dice once before it chooses the good ones, whereas the advanced estimator identifies the good dice when seeing them for the first time.

5.3 Creating Less Trivial Tests

The scenarios in the first tests are rather extreme. The dice are very easily distinguished by their outcomes. We now want to try to create an example, where picking the good dice is a little bit more difficult. When creating the dice for this test we want to fulfil two criteria: firstly choosing dice randomly should yield a poor result. If choosing dice randomly yielded a good result, we wouldn't need to make expensive estimations but instead could just download randomly. The expected frequencies of observations are equivalent to adding up the probability distributions of all dice and normalizing. Secondly it should be possible to choose dice in such a way that the expected frequencies of observations are close to uniform distributed data. These demands can be formulated in a linear equation:

Let $T > N \in \mathbb{N}$ and $p^1 = (p^1_1, \cdots, p^1_T), \cdots, p^N = (p^N_1, \cdots, p^N_T)$ be the probability distributions of the dice. We write the probability distributions as a matrix:

$$A := \begin{pmatrix} p^1_1, \cdots, p^N_1 \\ \vdots \\ p^1_T, \cdots, p^N_T \end{pmatrix}$$

Let $(m_1, \cdots, m_N) \in [0,1]^N$ with $\sum_{k=1}^{N} m_k = 1$. Let $u = (\frac{1}{T}, \cdots, \frac{1}{T})^\intercal$. We then minimize the error:

$$\text{Minimize} \|A \cdot m - u\|^2 = \sum_{j=1}^{T} \left(\sum_{k=1}^{N} a_{j,k} \cdot m_k - u_j \right)^2$$

For test 3 probability distributions were used in such a way that $m_k = \frac{1}{N}$ for all $k \in 1, \cdots, N$ would be a very bad solution and secondly that the solution to the corresponding linear equation has positive values.

5.4 Dirichlet Mixture

In this test we want to describe a situation in which the dirichlet mixture and the cluster estimator produce better results than the advanced estimator. For this test we use two types of dice with the following parameters:

Test 3:

$p_i := \left(0, 0, 0, 0, 0, 0, 0, 0, \frac{1}{2}, \frac{1}{2}\right), \text{readNum}_i = 100 \text{ for } i \in 1, \cdots, 64$

$p_i := \left(\frac{1}{100}, \frac{1}{100}, \frac{1}{100}, \frac{1}{100}, \frac{1}{100}, \frac{1}{100}, \frac{1}{100}, \frac{1}{100}, 0, \frac{92}{100}\right),$

$\text{readNum}_i = 30 \text{ for } i \in 65, \cdots, 128,$

$b = 10, \text{cost} = 0.005, \text{loadAllOnce } = 1$

$a = 1.0, \lambda = 180.$

The parameter `loadAllOnce` implies that all dice are chosen once and their observations are added, before the estimator begins choosing dice. The `readNum` is the number of throws a die can be thrown. After the number is reached the die can't be chosen in further iterations.

Motivation for this test The second type of die is more valuable because the probability-distribution is much closer to a uniform distribution. We can view the first eight outcomes of the die as a success and the last outcome as a loss. That leads to the following binomial-distribution $p := \left(\frac{8}{100}, \frac{92}{100}\right)$. The expected number of throws to get a success is then: $\frac{100}{8} = 12.5$. The batch size `b = 10` is the number that defines how many times a die is thrown each time it is chosen. That means on average a die of type two needs to be chosen two times to produce an outcome on one of the first eight sides of the die. Suppose the dirichlet mixture is able to cluster the two types of dice correctly. If then only one of the dice of type two produced a desirable outcome (one of the first eight sides on the die), the dirichlet mixture might be able to improve the estimation for all other dice of type two. The advanced estimator would only value dice of type two as high if it produced at least one desirable outcome. The advanced estimator however would not value the dice with no desirable outcome as high. An exemplary situation might look like this:

$$\begin{aligned}
\text{Cluster1}:& \\
c_1 &= \big(0,0,1,0,0,0,0,0,0,9\big) \\
c_2 &= \big(0,0,0,0,0,0,0,0,0,10\big) \\
c_3 &= \big(0,0,0,0,0,0,0,0,0,10\big) \\
c_4 &= \big(0,0,0,0,0,0,0,0,0,10\big) \\
c_5 &= \big(1,0,0,0,0,0,0,0,0,9\big)\,.
\end{aligned}$$

Suppose the dice $c_1, \cdots, c_5$ are all of the second type as described in test 3, and correctly clustered into the same component. In this situation the advanced estimator would not expect c_2, c_3, c_4 to produce desirable outcomes, because the observations for each die are examined separately. The cluster estimator and the dirichlet mixture however take into consideration also the observations of the other runs in the same cluster. Therefore the potential of the dice c_2, c_3, c_4 is not overlooked.

The results are shown in Figure 3.

5.5 Tests on Simulated Runs

For the simulations of real runs, all reads of 373 randomly chosen runs of Drosophila *melanogaster* were downloaded. For each run all reads were aligned to the genome.

The observations made with this alignment were then saved in a csv-file. These csv-files were then read in, the observations were normalized and then used as probability-distributions.

5.5.1 104 Runs

In this test 104 randomly chosen runs and their observations as probability-density were simulated. We used the following parameters:

Test 4:

$$b = 1000000, \text{cost} = 0.001,$$
$$a = 1.0, \lambda = 1000000.$$

The result is shown in Figure 4. Both estimators reached their maximum profit at around iteration twenty-five, with the advanced estimator being a bit better. Both estimators continued to chose dice and their profit dropped. The advanced estimator stopped at iteration 121. The simple estimator stopped at iteration 185. The profit achieved by the advanced estimator is around twelve times better.

5.5.2 373 Runs

In this test we used 373 randomly chosen runs and used their observations as probability-density for our dice. We used the following parameters:

Test 5:

$$b = 1000000, \text{cost} = 0.0005,$$
$$a = 1.0, \lambda = 60000.$$

The result is shown in Figure 5.

5.6 Test on Real Runs of Drosophila *Melanogaster*

In this test VARUS is tested on real downloads. With this test only the procedure of downloading and aligning was tested and no special focus was put on the results delivered by the estimators. VARUS has an option `profitCondition = 0`, which allows for downloading a fixed number of times specified with the parameter `maxBatches`, ignoring the expected profit as an exit condition. In this test a fixed number of 400 downloads was done with the simple estimator and the advanced estimator. The simple and advanced estimator each took roughly 26 hours to do the 400 iterations of downloads on my PC(8 GB RAM, 4 cores 2.4 GHZ).

Test 6:

$$b = 1000000,$$
$$a = 1.0, \lambda = 3.0,$$
$$profitCondition = 0, maxBatches = 400,$$

The plots are shown in Figure 6. The `cost` were set to a value other than zero which results in the total profit decreasing over time. Both estimators only choose the runs for further downloads out of a handful of runs after roughly downloading 100 iterations, that means roughly by the time each run was downloaded once. This might indicate a very desirable attribute of the estimators, that is after some training the best runs are chosen over and over again.

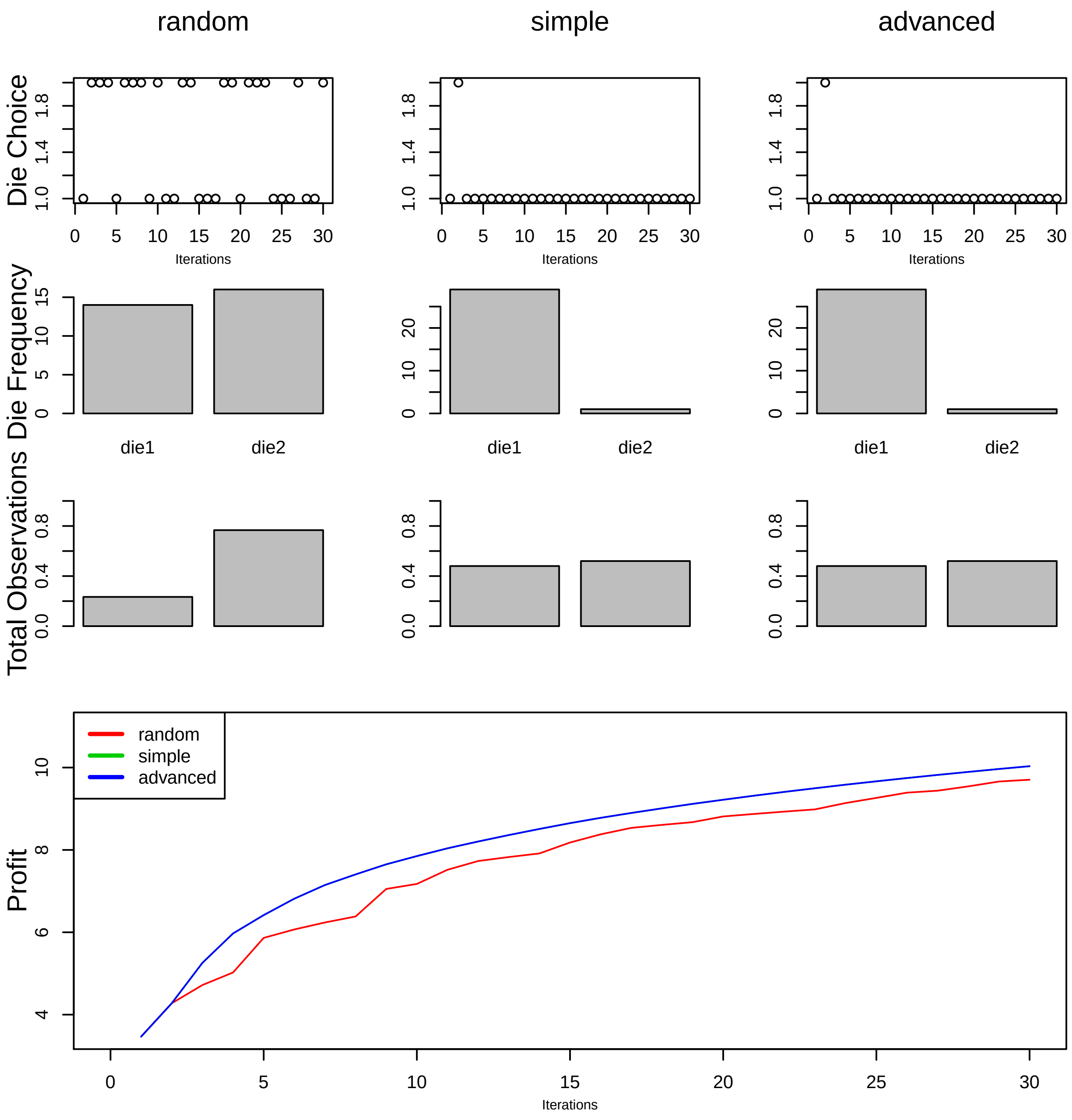

Figure 1: fair die vs unfair die (green and blue are identical) (Test 1)

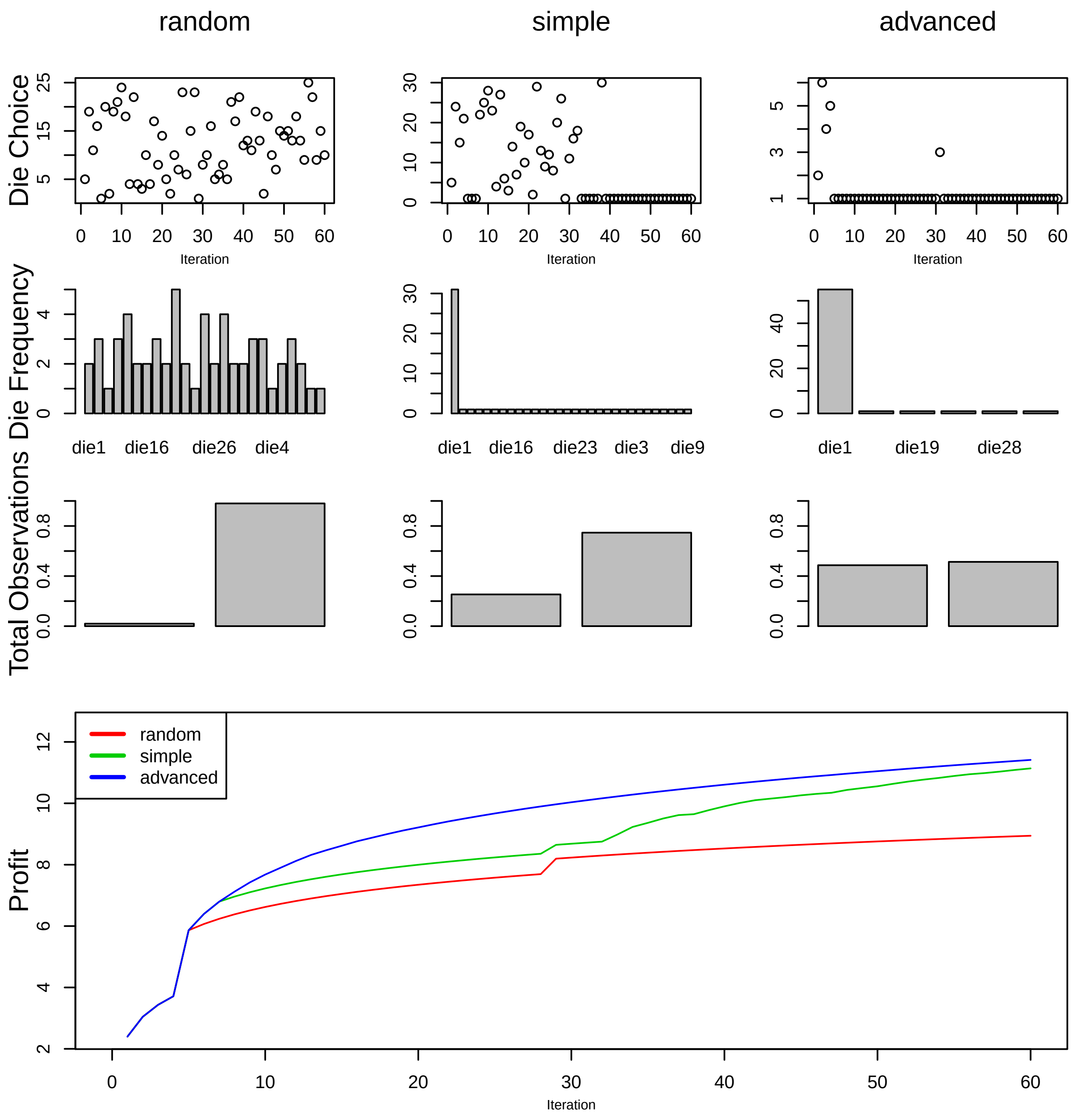

Figure 2: 5.2 1 fair die vs 29 unfair dice (Test 2)

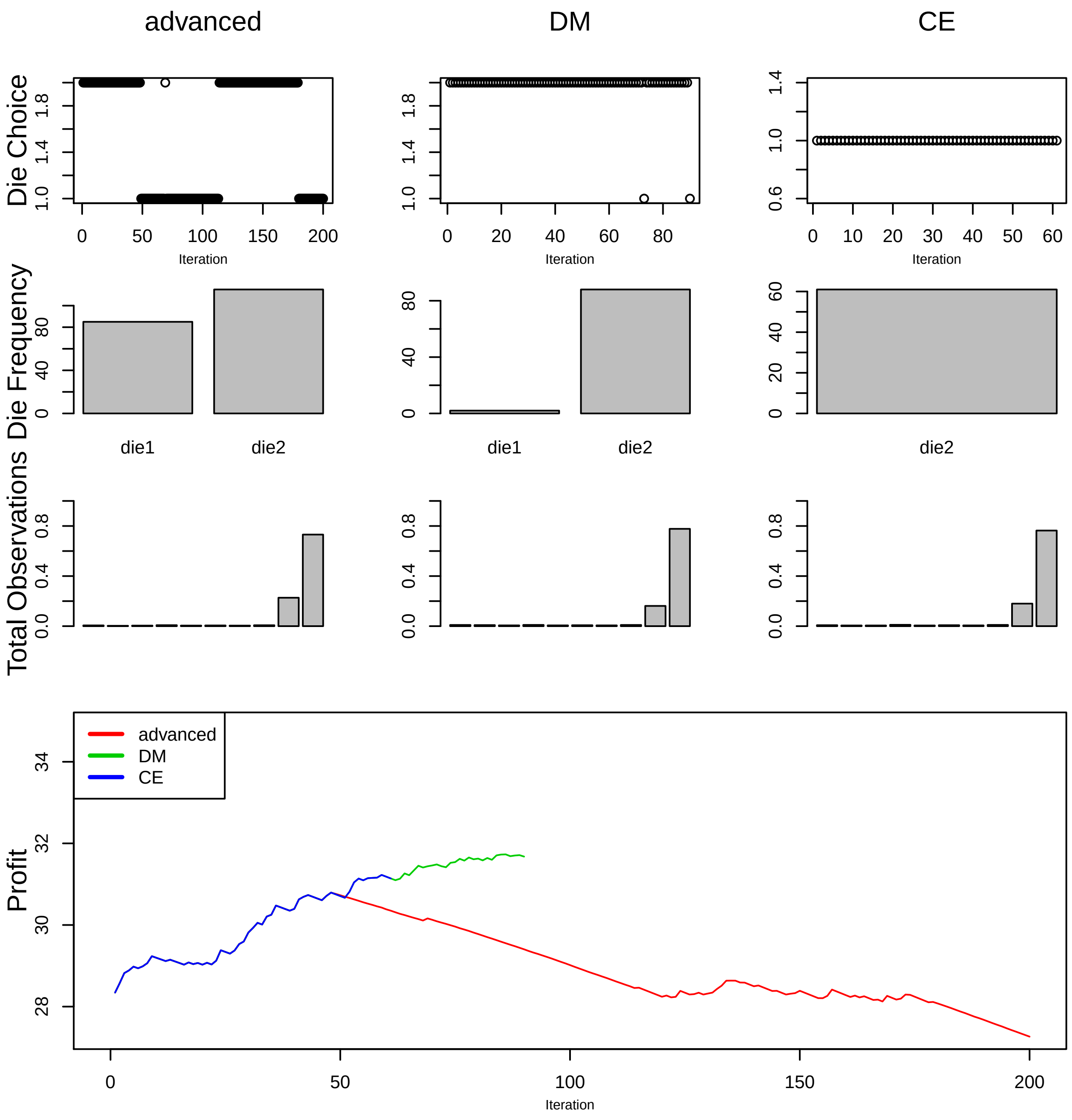

Figure 3: advanced estimator, dirichlet mixture and cluster estimator (Test 3)

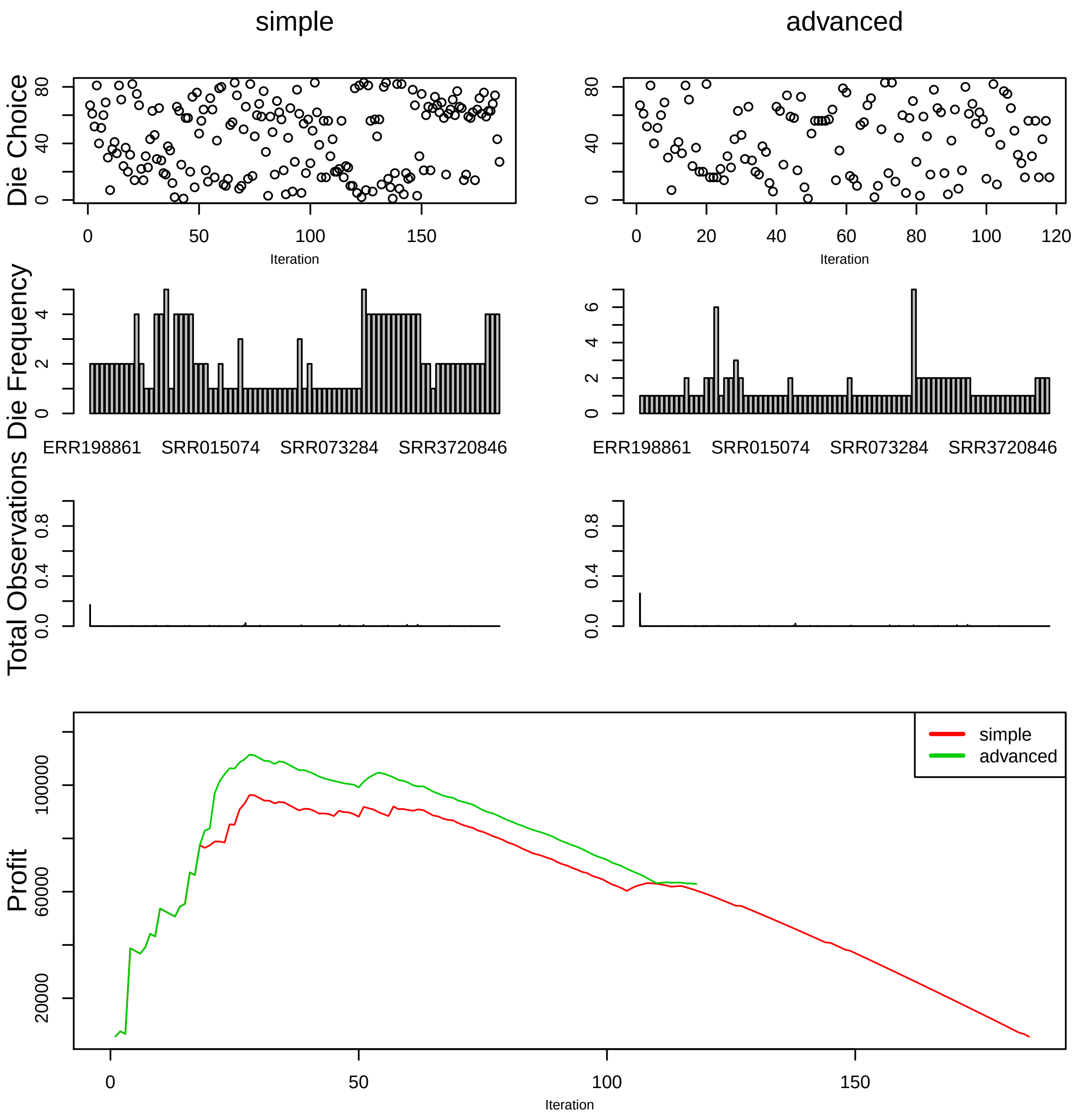

Figure 4: 104 simulated runs (Test 4)

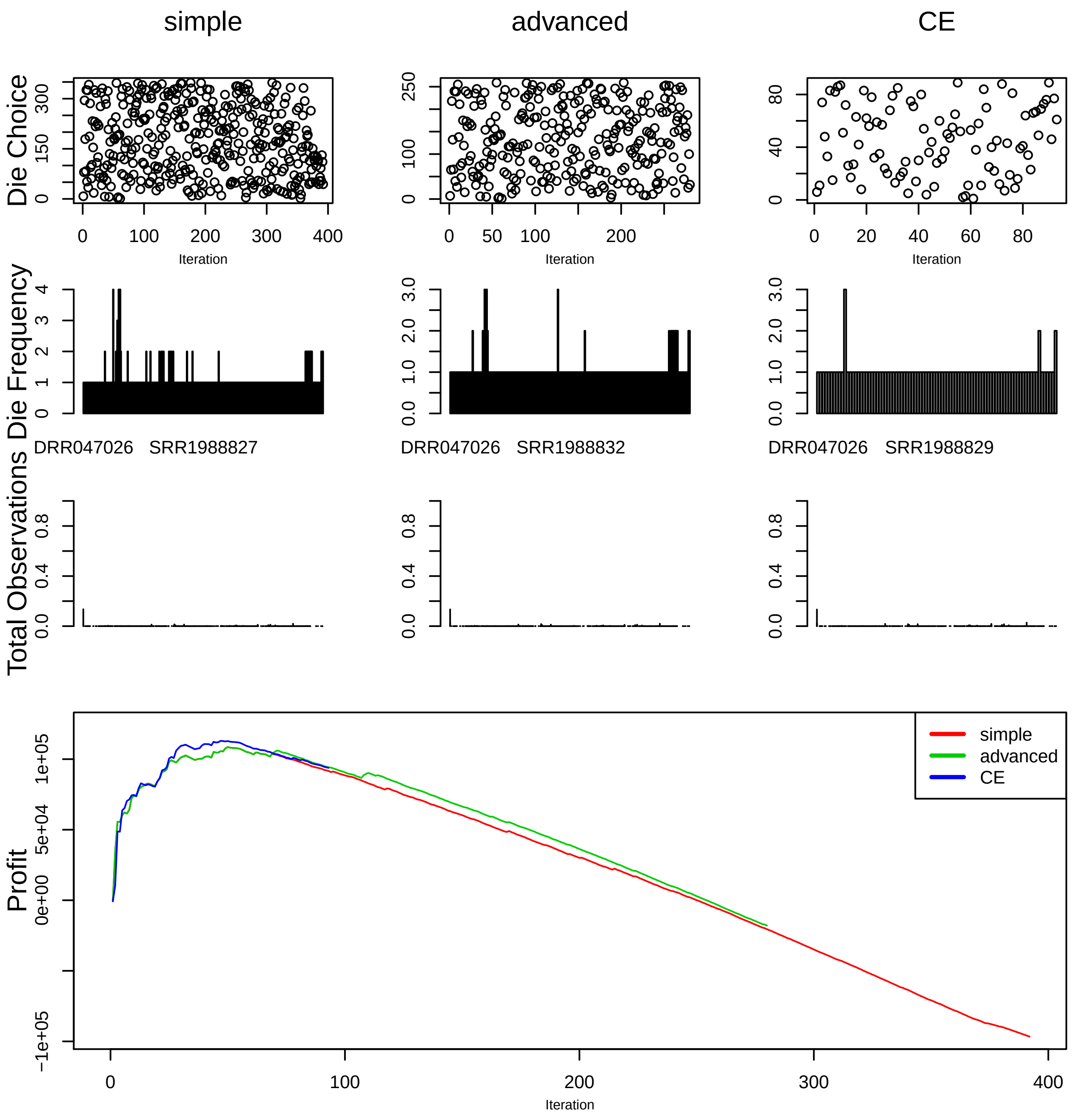

Figure 5: 373 simulated runs with simple, advanced and cluster estimator (Test 5)

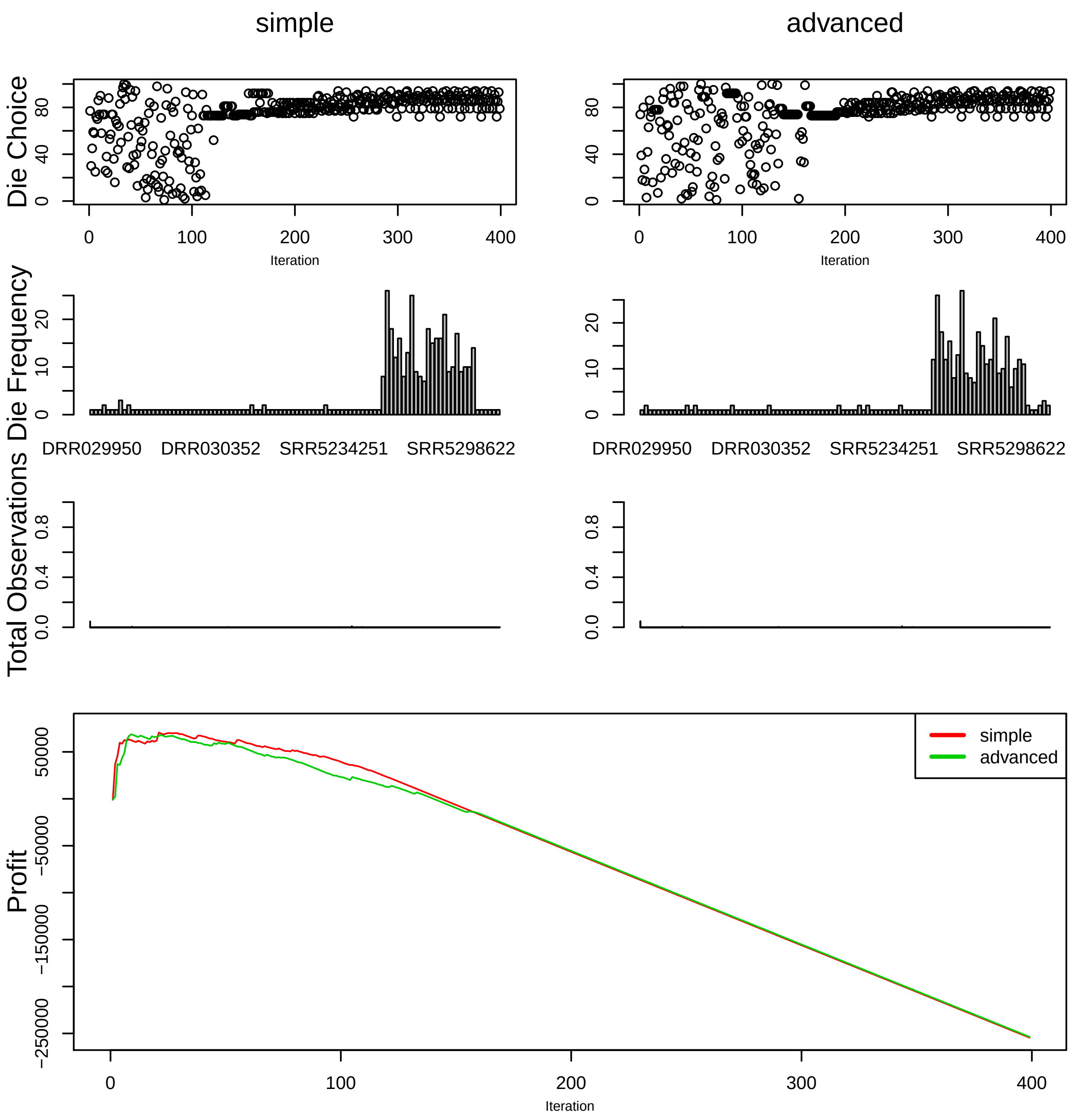

Figure 6: 400 iterations of real downloads (Test 6), cost ≥ 0 which leads to the decreasing total profit

6 Implementation of VARUS

6.1 Workflow

In this section we discuss the basic workflow of VARUS. The different components necessary are briefly described. The different dependencies are illustrated in Figure 7.

6.2 Run List Retriever

The *Run List Retriever* is a perl-script [5] that was developed for this thesis. It retrieves the accession-ids of the available runs from the NCBI. These ids are printed along with the number of reads in the run, the number of bases in the run and a flag indicating whether the reads are paired or not, into a file called "Runlist.txt". These attributes are tabulator-separated. The first line with an "@" is dismissed when read in by VARUS. An example of a run-list with two runs might look like the one shown in Figure 8.

```
@Run_acc        total_spots        total_bases        bool:paired
SRR4116438        31764766        3176476600        0
SRR4116437        31828879        3182887900        0
```

Figure 8: Examplary runlist with two runs

The use of this script is arbitrary and might not be stable for future use, since it depends on the structure of the websites of the NCBI. However the format of the run-list should be kept, since it is the input format expected by VARUS.

6.3 Fastq-dump

Fastq-dump [3] is a tool of the sra-toolkit. It is possible to download batches of runs from the NCBI. For a given run two parameters N and X define the range of the spots to be downloaded. Calling fastq-dump with the parameters

```
  $ fastq-dump SRR001339 -N 10000 -X 20000 --fasta
```

downloads 10000 reads of the run SRR001339 in fasta-format.

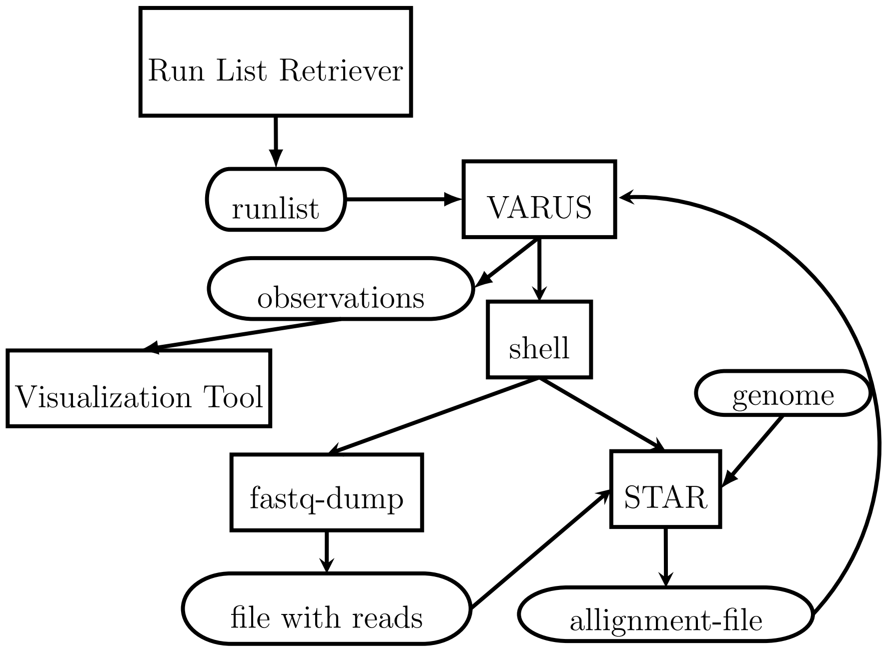

Figure 7: Workflow: the rectangles are programms, the rounded shapes are files

6.4 STAR

Spliced Transcripts Alignment to a Reference (STAR) is a RNA-Seq aligner. It aligns RNA-Seq data to transcriptomes or genomes. In this thesis alignment to a genome was done. The input for this program are the reads downloaded with fastq-dump. The output is an alignment-file in SAM format.

6.5 VARUS

VARUS is the core-program that was implemented in c++ for this thesis. It controls the download-step and the alignment-step. VARUS chooses runs to download reads from, based on the runs appearing in the "Runlist.txt". The command for downloading a run is invoked through the shell and calls fastq-dump with the needed parameters. After successfully downloading a batch from a run, STAR is called through the shell. STAR aligns the reads and then outputs an alignment-file in SAM-format. This file is then read in by VARUS and the uniquely mapping reads are retrieved. The observations of the run are changed accordingly. After this step VARUS decides to exit the program or to download another batch.

6.5.1 Batch Choice

VARUS downloads relatively large batches at once. The maximum number of spots in a run is divided by b the *batch size*, given as a input parameter. This division yields the number of times we can download from that specific run with the given batch size `b`. To diminish a possible bias in the order of the reads, we choose the next batch randomly. The reads might be lexicographically ordered. Loading the first 100000 reads might then not be representative for the whole run.

6.5.2 Choice of the Data Structure for the Observations

unordered_map

A read is assigned to exactly one of the $0, \cdots, T$ blocks in the alignment. The counts of the reads per block form the observations. Typically the number T is quite large. This makes necessary a data structure, that allows for fast access of the block and their assigned read-counts. The key is the name of the block and the value is the count of the reads assigned to this block.

key, value

Since each run has to save his own observations a look at which type to use for the key and value should be taken. VARUS uses a single header-file "TypeConventions.h" in which the type of the key and value for the maps, that should represent observations, are defined. Currently the type for both the key and the value is *unsigned int*. It could be taken into consideration to take the data-type *short* as the key. unsigned *int_32* has a size of 4 bytes. The data-type *char* has a size of 1 byte. A *string* consists of multiple chars. The blocks of the chromosomes are named as chrName:number in VARUS. That makes a size of at least 3 bytes. Since many chromosomes have names with more than one letter, this size will often be much larger. That is why choosing a string as the data-type for the key for each map of each run is more memory-consuming. Instead using an *unsigned int_t32* as the data-type for the key is expected to be less memory consuming. This makes necessary some kind of translation from the actual names of the block names into the numbers $0, \cdots, T$. For this purpose VARUS also uses an *unordered_map* which has the block names as keys and one of the $0, \cdots, T$ numbers as value. The advantage is, that this map needs to be saved only once for all runs.

Memory Usage

All runs listed in the "Runlist.txt" are initialized. That means even for blocks with no observations an entry in the *unordered_map* is created right from the beginning.

```
blockName                  Observations
mitochondrion_genome:0     33636
mitochondrion_genome:1     29072
mitochondrion_genome:2     154729
mitochondrion_genome:3     44
211000022278950:0          0
211000022278909:0          0
211000022278309:0          1
211000022278504:0          0
211000022279212:0          0
        .                  .
        .                  .
        .                  .
```

Figure 9: Observations in a run

That includes the observations and estimations for the observations. The observations are initialized with zero observations for each key. For Drosophila *melanogaster* with a `blockSize` of 5000 base pairs this makes 29918 entries. As described in 6.5.2 the sizes of the keys and values of these maps are both four bytes. With Looking only at the leaves of the *unordered_map* the lower bound for the memory-usage of a single run is then at least:

$$(4\,\mathrm{B} + 4\,\mathrm{B}) \cdot 29918 \approx 0.241\,\mathrm{MB} \leq \mathrm{MemoryUsage}$$

6.5.3 Simulation

For the simulation of the download and alignment step VARUS uses the *discrete-distribution* from the c++stl(*Standard Library*). In VARUS with `simulation = 1` for each run a probability-distribution is read in. These probability-distributions are then used as an input for the discrete-distribution from the stl to generate observations.

The option `createDice` can be set to true, in order to create simulation-files out of runs. The runs in the run-list are then read in and downloaded completely and aligned to the genome. The resulting observations are then saved in a CSV-file, which can then be used for simulations. In Figure 9 the first entries from an exemplary run are shown.

The numbers in the column "Observations" are then normalized and used to form a hidden distribution, based on which the simulated observations can be created.

6.5.4 Observations and Hidden Distributions

The output of VARUS are informations about the observations made in the runs. The first entries from an exemplary iteration from a simulation is shown in Figure 10.

```
blockName   totalObs.   totalScore  totalProfit best    die1.obs  die1.p
1           87          15,8256     14,9456     die3    9         0,416
2           35          15,8256     14,9456     die3    2         0,125   ...
3           37          15,8256     14,9456     die3    1         0,083
4           61          15,8256     14,9456     die3    8         0,375
```

Figure 10: Example of the output of VARUS

6.5.5 Dirichlet Mixture

As the other estimators are rather straight forward to implement in comparison to the dirichlet mixture, we will only discuss some of the details of the dirichlet mixture.

As written in 3.4.6 the training is continued until the parameters do not change in between iterations. This is however not directly implemented in VARUS. Instead a single parameter `trainingsIterations` is used to specify the number of iterations that the training should take place. As a consequence it can occur that the runs are only very poorly separated into the components. On the other hand this ensures that the training will always take the same amount of iterations and the program will not be stuck in the training process forever. Finding a good value for the number for the `trainingsIterations` should be investigated in future works. The calculation of the start value for the Newton's method as described in 3.4.8 can only be done if at least two runs are sampled into the component in question. In VARUS this is achieved by adding two artificial runs to each component. The functioning of these runs is comparable to the functioning of the pseudo-counts in the simple estimator. The runs are initialized with a uniform distribution of observations meaning one observation for each block. In the current implementation these runs are also used for the estimations. If these *pseudo-runs* are fruitful for the estimation as well, needs more investigation.

A *profiler* is used to determine exactly how much time a function call within an implementation costs. In Figure 11 the profile of the dirichlet mixture in a typical scenario, made with gprof [4] is shown.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self
 time   seconds   seconds   name
 35.03     14.54    14.54   trigamma
 22.96     24.07     9.53   digamma
 16.48     30.91     6.84   DirichletMixture::ddL
  7.25     33.92     3.01   DirichletMixture::dL
  2.17     34.82     0.90   DirichletDistribution::calculateCj
  2.05     35.67     0.85   DirichletDistribution::calculateLikelihood
  2.00     36.50     0.83   DirichletDistribution::EalphaSum
                            .
                            .
                            .
```

Figure 11: Profile output of the dirichlet mixture using gprof

The functions *digamma* and *trigamma* were both translated by me into c++ from the original code [10] for this implementation. These functions can probably be implemented much more efficiently by using for example the implementations offered by boost [1].

6.5.6 Performance Overview

In order to give a very rough orientation on how much time the different estimators need for the calculation a very rough benchmark test was done on my PC(8 GB RAM, 4 cores 2.4 GHZ). In Figure 12 the calculation times with the different estimators are shown. The corresponding test includes 100, 200 and 300 runs of simulated runs of Drosophila *melanogaster*. The `user` times, derived with the call `$ time ./VARUS`, are shown in the column "time". A single estimation of all runs was done.

`estimator = 3` corresponds to the dirichlet mixture, `estimator = 4` corresponds to the cluster estimator and `estimator = 1` corresponds to the simple estimator. The column "newtons" corresponds to the parameter `newtonIterations` and the column "trainings" corresponds to the parameter `trainingsIterations`. In this example the dirichlet mixture takes roughly six times as long as the cluster estimator to complete a single estimation step. The simple estimator roughly takes two thirds of the time that the cluster estimator takes.

runs	estimator	newtons	training	times
100	3	1	1	18.53
100	3	1	5	45.53
100	3	1	10	81.05
100	3	4	1	24.59
100	3	4	5	85.26
100	3	4	10	164.25
100	3	10	1	34.60
100	3	10	5	131.81
100	3	10	10	260.07
200	3	6	5	196.99
300	3	6	5	301.46
100	4	%	5	13.12
200	4	%	5	27.93
300	4	%	5	47.54
100	1	%	%	8.88
200	1	%	%	17.41
300	1	%	%	26.22

Figure 12: Performance Overview of dirichlet mixture, cluster estimator and simple estimator

6.6 Visualization

For the visualization of the progress in each step of the online algorithm CSV-files can be exported, containing the total observations and other information. A script written in R [6] can be used to read in these CSV-files and plot different things. The plots in section 5 are made with this script.

7 Appendix

7.1 Manual

7.1.1 VARUS

This is the usage of VARUS:

```
Usage:
Online-algorithm to download RNA-seqdata.
--------------------------------------------------------------------------------
--batchSize:                the number of reads to be downloaded at once.

--blockSize:                the number of bases one block will have. This is
                            done in order to be able to compare the coverage of
                            larger chromosomes with smaller ones, since larger
                            chromosomes will naturally have more reads mapped to
                            them than smaller ones.

--components:               sets the number of components of the dirichlet
                            mixture or the cluster estimator. Only relevant for
                            estimators 3 and 4.

--cost:                     sets the cost for downloading one read.

--createDice:               if set to 1 dice will be created from real runs.Only
                            needed for testing/simulation-purposes.

--deleteLater:              if set to 1, the fasta-files and alignment-files
                            will be deleted after they are used to identify the
                            next run to be downloaded from. If you want to use
                            the reads for your genome-annotation you should not
                            use this option.

--estimator:                1 == simple, 2 == advanced, 3 == dirichlet mixture,
                            4 == cluster estimator, else downloads will be done
                            choosing the runs randomly. Note that if you choose
                            to download randomly, you should specify maxBatches
                            in order to let the program end at some point.

--exportNewtons:            exports the steps of the newtons method. Deprecated
                            since it is very expensive. Only relevant for
                            estimator 3.

--exportObservationsToFile: if set to 1 the program will output the observations
                            in all runs in all steps into CSV-files. NOTE: Using
                            this option can lead to performance-issues.

--exportParametersToFile:   if set to 1 the parameters used for this execution
                            of the program will be exported into a
                            parametersfile

--genomeDir:                specifies the path to the genome.

--ignoreReadNum:            Only important for simulation: if ignoreReadNum ==
                            1, it will be ignored in case of a simulation if a
                            run has no reads left. Otherwise the run is not an
                            option to download from after the maximum number of
                            reads is downloaded from this run.

--lambda:                   parameter for estimator 2.

--lessInfo:                 if set to 1, Toy prints less info. Only Relevant for
                            simulation.

--loadAllOnce:              if set to 1, a single batch from each run will be
                            downloaded once, before the estimation process
                            starts. Useful for the expensive estimators 3 and 4.

--maxBatches:               if maxBatches > 0, the program will exit as soon as
                            it loaded maxBatches batches.

--newtonIterations:         number of times the newtons method is done to find
                            the maximum-likelihood for the alpha-sum. Only
                            relevant for estimator 3.

--newtonPrecision:          threshold at which the newtons-method will aboard,
                            and return the value. Only relevant for estimator 3.

--numOfBlocks:              sets the number of blocks into which the genome
                            should be divided.Reading it automatically DOES NOT
                            WORK YET! You must specify the correct number of
                            blocks.

--outFileNamePrefix:        specifies the path in which all output of VARUS
                            should be stored.

--pathToDice:               specifies the path to the dice. Dice are saved in
                            csv-format.

--pathToParameters:         specifies the path and name to the parameters-file
                            that should be read in and written to.

--pathToRuns:               specifies the path to Runlist.txt.
```

```
--pathToSTAR:                specifies the path to the executable of STAR, the
                             alignment-program.

--profitCondition:           if profitConditon == 1, the program will exit if the
                             expected profit falls below 0. Note that the
                             expected profit can lead to the program downloading
                             for a very long time, since some of the estimators
                             tend to be very optimistic if the parameters are not
                             set adequately.

--pseudoCount:               adds a pseudocount to all possible observations.
                             Only relevant for estimators 1, 2.

--randomSeed:                if randomSeed > 0, the program will have
                             deterministic results. Else the seed will be set
                             according to the current time

--readParametersFromFile:    if set to 1, the program will look for a file
                             specified with pathToParameters and interpret its
                             content as command-line arguments. These parameters
                             will then be used to run the program. Note:
                             additional parameters passed with the command line
                             will overwrite the parameters read from the
                             parameters-file.

--runThreadN:                Number of threads to run STAR with. Read STAR-manual
                             for more information.

--simpleDM:                  refers to estimator 3. With this estimation
                             procedure the calculation times are a bit better
                             than with the normal dirichlet mixture. However the
                             estimation is not that accurate.

--simulation:                if set to 1, the program will simulate downloads.

--trainingsIterations:       the number of iterations the dirichlet mixture or
                             cluster estimator will be trained in each step. Only
                             relevant for estimators 3 and 4.

--verbosityDebug:            sets the debug-verbosity-level. There are 4
                             verbosity-levels. Higher values mean more output.
```

7.1.2 Run List Retriever

This is the usage of of the Run List Retriever written in perl:

```
Usage:
  --genus:         default: Drosophila

  --species:         default: melanogaster

  --retmax:         default: 100, number of runs to be downloaded

  --retstart:         default: 1, download from retstart to retstart + retmax-1

  --outFileDir: default: "", the directory where all files will be stored

  --all:         default: 'false', retrieve all available runs

  --paired:         default: 'false', retrieve only paired-seq runs
```

7.1.3 Visualization Tool

The visualization tool used for the plots made in section 5, is written in R. The command line arguments for this program are the names of the folders in which the csv-files produced by VARUS are located. The names of these folders also form the caption of the plots.

References

[1] Boost C++ Libraries. `http://www.boost.org/`, 2017.

[2] European Nucleotide Archive. `http://www.ebi.ac.uk/ena`, 2017.

[3] Fastq-dump. `https://ncbi.github.io/sra-tools/fastq-dump.html`, 2017.

[4] Gprof. `https://sourceware.org/binutils/docs-2.16/gprof/`, 2017.

[5] Perl. `https://www.perl.org/`, 2017.

[6] R. `https://www.r-project.org/`, 2017.

[7] Sequence Read Archive, NCBI. `https://www.ncbi.nlm.nih.gov/sra`, 2017.

[8] Alexander Dobin, Carrie A Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R Gingeras. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

[9] K J Hoff and M Stanke. Current methods for automated annotation of protein-coding genes. *Current Opinion in Insect Science*, 7:8–14, 2015.

[10] B. E. Schneider. Algorithm as 121: trigamma function. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 27(1):97–99, 1978.

[11] K Sjlander, K Karplus, M Brown, R Hughey, Anders Strmose Krogh, IS Mian, and D Haussler. Dirichlet mixtures: a method for improved detection of weak but significant protein sequence homology. *Bioinformatics*, 12(4):327–345, 1996.

[12] Mario Stanke, Mark Diekhans, Robert Baertsch, and David Haussler. Using Native and Syntenically Mapped Cdna Alignments to Improve De Novo Gene Finding. *Bioinformatics*, 24(5):637–644, 2008.

[13] Mario Stanke, Rasmus Steinkamp, Stephan Waack, and Burkhard Morgenstern. Augustus: a Web Server for Gene Finding in Eukaryotes, 2004.

[14] Xugang Ye, Yi-Kuo Yu, and Stephen F Altschul. On the Inference of Dirichlet Mixture Priors for Protein Sequence Comparison. *Journal of Computational Biology*, 18(8):941–954, 2011.