Introduction to genBaRcode

2020-05-08

Table of Contents

5. Generating and plotting Time Series Data

6. genBaRcode Shiny-App

7. Miscellaneous Functions

1. *genBaRcode* Package

The *genBaRcode* package is intended to be a comprehensive toolbox for the analysis of genetic barcode data after next generation sequencing (NGS). It combines the necessary functionalities needed for data extraction, analysis and error-correction, in combination with a variety of visualizations. Furthermore, there is an ancillary shiny-app available which provides a graphical user interface for all the basic functions in order to also make the package usable for less programming experienced users.

Since the CRAN installation routine does not support the automatic installation of Bioconductor packages and if you have not installed the necessary Bioconductor packages already, you have to install the packages *Biostrings*, *ShortRead*, *S4Vectors* and *ggtree* manually.

```
if (!requireNamespace("BiocManager", quietly = TRUE)) {
    install.packages("BiocManager")
}

BiocManager::install(c("Biostrings", "ShortRead", "S4Vectors", "ggtree"))
```

After loading the package with the `require("genBaRcode")` (or `library("genBaRcode")`) command, the entire functionality of the package can be accessed.

2. Barcode Extraction

Normally, every analysis starts with an NGS file, in our case a fasta or a fastq file. In order to extract the genetic barcodes present in the respective file, you have to start with the `processingRawData()` function.

```
require("genBaRcode")

bb <- "ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN"
source_dir <- system.file("extdata", package = "genBaRcode")

BC_data <- processingRawData(file_name = "test_data.fastq.gz",
                             source_dir = source_dir,
                             results_dir = "/my/results/directory/",
                             mismatch = 0,
                             label = "test",
                             bc_backbone = bb,
                             bc_backbone_label = "BC_1",
                             min_score = 30,
                             min_reads = 2,
                             save_it = FALSE,
                             seqLogo = FALSE,
                             cpus = 1,
                             strategy = "sequential",
                             full_output = FALSE,
                             wobble_extraction = TRUE,
                             dist_measure = "hamming")
```

First, you have to choose a NGS file (in our case, as already mentioned, a fasta or fastq file) and assign the name to the parameter `file_name` of the `processingRawData()` function. Two other essential input parameters are called `source_dir` and `results_dir`, here you have to assign a character string describing the path to the chosen file and the path to the directory where you wish to store your results. To make the vignette examples replicable for everybody, we chose the system-specific path to the installation-directory of the package and the included example file, called *test_data.fastq.gz*.
Optionally, you can assign a sample specific label to the parameter named `label`. This character string will internally be used as data-object label, as directory name for all the result-files created and furthermore as discriminator within the generated file names. Depending on the specific parameter choices, the function will first of all read the declared data file. If the `min_score` parameter was assigned a numeric value greater than `0`, all NGS-reads with an average score smaller than this value will be dismissed. The next step is the extraction of all barcode-carrying sequences. In order to do so you have to either provide one or several barcode-backbone structures or the character string `"none"` which will lead to a simple clustering of every sequence present in the respective NGS file. The `mismatch` parameter offers the possibility to define the number of allowed mismatches between the backbone and the matching nucleotides. No atter if only the clustering should be performed or a barcode-backbone was provided, you can also choose between different distance measures (`dist_measure`) on which the clustering or the backbone matching shall be based on and the `mismatch` parameter will then be interpreted as the maximal allowed difference between either two sequences in order to be clustered together or between a sequence and the respective backbone. The default method is the Hamming distance (`dist_measure = "hamming"`), another common choice would be the Levenshtein distance (`dist_measure = "lv"`) but there are also further methods available (type `?'stringdist-metrics'` in the R-console for further details).

Since the package is closely related to several different established genetic barcode constructs, there is also a function which offers all those backbone sequences, it's called `getBackboneSelection()`.

```
  getBackboneSelection()
#>   name
#> 1 BC32-GFP
#> 2 BC32-Venus
#> 3 BC32-eBFP
#> 4 BC32-T-Sapphire
#> 5 BC16-GFP
#> 6 BC16-Venus
#> 7 BC16-mCherry
#> 8 BC16-Cerulean
#>   sequences
#> 1 ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN
#> 2 CGANNAGANNCTTNNCGANNCTANNGGANNCTTNNCGANNAGANNCTTNNCGANNCTANNGGANNCTTNNCGANNAGANN
#> 3 CTANNCAGNNCTTNNCGANNCTANNCTTNNGGANNCTANNCAGNNCTTNNCGANNCTANNCTTNNGGANNCTANNCAGNN
#> 4 CAGNNATCNNCTTNNCGANNGGANNCTANNCTTNNCAGNNATCNNCTTNNCGANNGGANNCTANNCTTNNCAGNNATCNN
#> 5 ATCNNTAGNNTCCNNAAGNNTCGNNAAGNNTCGNNAGTNNTAG
#> 6 CTANNCTANNCAGNNCTTNNCGANNCTANNCTTNNGGANNGAT
#> 7 CTANNCAGNNATCNNCTTNNCGANNGGANNCTANNCTTNNGAT
#> 8 CTANNCACNNAGANNCTTNNCGANNCTANNGGANNCTTNNGAT

  bb <- getBackboneSelection(1)
  show(bb)
#> [1] "ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN"

  bb <- getBackboneSelection("BC32-eBFP")
  show(bb)
#> [1] "CTANNCAGNNCTTNNCGANNCTANNCTTNNGGANNCTANNCAGNNCTTNNCGANNCTANNCTTNNGGANNCTANNCAGNN"
```

The shown example backbones consist of 16 and 32 wobble bases (coded by `Ns`) interspersed by fixed triplets in a defined order (Cornils et al., Thielecke et al. 2017).
As mentioned above, it is also possible to provide more than one backbone at once, therefore we would recommend to also define backbone-specific labels (`bc_backbone_label`), since those labels will also be part of almost everything related to the analysis of those particular backbones, e.g. the names for the created directories and file (if you provide no label at all a numericlabel will be created automatically). After a barcode-extraction based on the hamming distance, the resulting sequences will have exactly the same length as the corresponding backbone pattern(s), since all flanking sequences will be dismissed. Additionally, if you just want to end up only with the so-called wobble bases (coded by `Ns`), you have to set the parameter `wobble_extraction` to `TRUE`. Furthermore, if you are only interested in barcodes with a certain minimum number of read-counts, you can define such a threshold utilizing the `min_reads` parameter and all barcodes with less reads will be dismissed.

Within R, the resulting data object will be a S4 data-object of type *BCdat* which consists of the extracted barcode sequences, their corresponding read counts, the assigned results directory, the used barcode backbone and the assigned label.

Here you can see the resulting data structure, if only one backbone was provided and the `wobble_extraction` parameter was set to `TRUE`.

```
  bb <- "ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN"
  source_dir <- system.file("extdata", package = "genBaRcode")
```

```
# if no results_dir is provided the source_dir automatically also becomes the results_dir
BC_data <- processingRawData(file_name = "test_data.fastq.gz",
                             source_dir = source_dir,
                             mismatch = 0,
                             label = "test",
                             bc_backbone = bb,
                             bc_backbone_label = "BC_1",
                             min_score = 30,
                             min_reads = 2,
                             save_it = FALSE,
                             seqLogo = FALSE,
                             cpus = 1,
                             strategy = "sequential",
                             full_output = FALSE,
                             wobble_extraction = TRUE,
                             dist_measure = "hamming")
```

```
show(BC_data)
#>  class: BCdat
#>
#>   number of barcode sequences: 81
#>   read count distribution: min 3  mean 110.36  median 4  max 2405
#>   barcode sequence length: 32
#>
#>  barcode read counts:
#>  read_count                          barcode
#>        2405 CACGATCCGCTTCTATCGCGTGCACTACATGT
#>        1504 GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT
#>        1296 GCTAAGGGCGATCACATCCACAAGCTTCTTTG
#>        1274 CAGAATCGAATGATGTCTTCATCTCAACGCCG
#>        1058 AGTATTCCGACAGTGACATGTCTTCCCGCGTT
#>         968 GGCCTAGAGTAGTTGTCGCGAAAGTCGGCCTC
#>          45 AGTATTCCGACCGTGACATGTCTTCCCGCGTT
#>          37 CAGAATCGCATGATGTCTTCATCTCAACGCCG
#>          22 GGCCTAGCGTAGTTGTCGCGAAAGTCGGCCTC
#>          19 GCCATCTATATTTGTTCCAAGACTCTACTATT
#>                        ...
#>  results dir:
#>        /my/results/dir/
#>  barcode backbone:
#>        ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN
#>  label:
#>        test
```

Here, two backbones are provided and `wobble_extraction` was set to FALSE. Besides the longer sequences you can see that it will result in a list of *BCdat* objects.

```
# if no results_dir is provided the source_dir automatically also becomes the results_dir
BC_data_multiple <- processingRawData(file_name = "test_data.fastq.gz",
                             source_dir = source_dir,
                             mismatch = 0,
```

```
                              label = "test",
                              bc_backbone = getBackboneSelection(1:2),
                              bc_backbone_label = c("BC_1", "BC_2"),
                              min_score = 30,
                              min_reads = 2,
                              save_it = FALSE,
                              seqLogo = FALSE,
                              cpus = 1,
                              strategy = "sequential",
                              full_output = FALSE,
                              wobble_extraction = FALSE,
                              dist_measure = "hamming")
#> Warning in prepareDatObject(dat[[b]], results_dir, label =
#> paste(label, : # There were barcodes detected for backbone
#> 'CGANNAGANNCTTNNCGANNCTANNGGANNCTTNNCGANNAGANNCTTNNCGANNCTANNGGANNCTTNNCGANNAGANN'
#> but all of them with less than 2 reads.
```

```
  show(BC_data_multiple)
#> [[1]]
#>  class: BCdat
#>
#>   number of barcode sequences: 81
#>   read count distribution: min 3  mean 110.36  median 4  max 2405
#>   barcode sequence length: 80
#>
#>  barcode read counts:
#>  read_count                                                barcode
#>        2405 ACTCACGACGCTTATCGACCCT...TTCAGGACTCTAACACTATCGAGT
#>        1504 ACTGGCGATCCTTGACGAAGCT...TTCAGGACGCTAGCACTTGCGACT
#>        1296 ACTGCCGATACTTAGCGAGGCT...TTAGGGACTCTATCACTTTCGATG
#>        1274 ACTCACGAGACTTATCGACGCT...TTCTGGACACTAACACTGCCGACG
#>        1058 ACTAGCGATACTTTTCGACCCT...TTTTGGACCCTACGACTCGCGATT
#>         968 ACTGGCGACCCTTTACGAGACT...TTAGGGATCCTAGGACTCCCGATC
#>          45 ACTAGCGATACTTTTCGACCCT...TTTTGGACCCTACGACTCGCGATT
#>          37 ACTCACGAGACTTATCGACGCT...TTCTGGACACTAACACTGCCGACG
#>          22 ACTGGCGACCCTTTACGAGCCT...TTAGGGATCCTAGGACTCCCGATC
#>          19 ACTGCCGACACTTTCCGATACT...TTCTGGACTCTAACACTTACGATT
#>                        ...
#>  results dir:
#>        /my/results/dir/
#>  barcode backbone:
#>        ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN
#>  label:
#>        test_BC_1
#>
#> [[2]]
#>  class: BCdat
#>
#>   number of barcode sequences: 1
#>   read count distribution: min NA  mean NA  median NA  max NA
#>
#>  barcode read counts:
#>  read_count barcode
```

```
#>          NA    <NA>
#>
#>  results dir:
#>        /my/results/dir/
#>  barcode backbone:
#>        CGANNAGANNCTTNNCGANNCTANNGGANNCTTNNCGANNAGANNCTTNNCGANNCTANNGGANNCTTNNCGANNAGANN
#>  label:
#>        test_BC_2
```

Now, you can see that if no backbone but the phrase `"none"` is provided the `wobble_extraction` parameter will have no effect. And the `mismatch` parameter was increased to `4`, since this will now define the maximal number of nucleotide differences for the clustering of the raw sequencing reads.

```
  # if no results_dir is provided the source_dir automatically also becomes the results_dir
  BC_data_2 <- processingRawData(file_name = "test_data.fastq.gz",
                                 source_dir = source_dir,
                                 mismatch = 4,
                                 label = "test",
                                 bc_backbone = "none",
                                 min_score = 30,
                                 min_reads = 2,
                                 save_it = FALSE,
                                 seqLogo = FALSE,
                                 cpus = 1,
                                 strategy = "sequential",
                                 full_output = FALSE,
                                 wobble_extraction = FALSE,
                                 dist_measure = "hamming")
```

```
  show(BC_data_2)
#>  class: BCdat
#>
#>   number of barcode sequences: 191
#>   read count distribution: min 3  mean 104.06  median 18  max 2821
#>   barcode sequence length: 90
#>
#>  barcode read counts:
#>  read_count                                              barcode
#>        2821 TCCACTCACGACGCTTATCGAC...CTCTAACACTATCGAGTCTCGAGA
#>        1726 ATAACTGGCGATCCTTGACGAA...CGCTAGCACTTGCGACTCTCGAGA
#>        1539 CCCACTGCCGATACTTAGCGAG...CTCTATCACTTTCGATGCTCGAGA
#>        1539 GACACTCACGAGACTTATCGAC...CACTAACACTGCCGACGCTCGAGA
#>        1416 ACCACTTGCGACGCTTACCGAA...AAACTAGAACTACCGATCCTCGAG
#>        1306 TGCACTAGCGATACTTTTCGAC...CCCTACGACTCGCGATTCTCGAGA
#>        1163 CCTACTGGCGACCCTTTACGAG...TCCTAGGACTCCCGATCCTCGAGA
#>         396 CAAGGAATCGGAACTCCAGTCA...CTCTAACACTATCGAGTCTCGAGA
#>         304 CAAGGAATCGGAACTCCAGTCA...AAACTAGAACTACCGATCCTCGAG
#>         300 TATACGACGGAACTCCAGTCAC...CTCTAACACTATCGAGTCTCGAGA
#>                        ...
#>  results dir:
#>        /my/results/dir/
#>  barcode backbone:
```

```
#>        none
#>  label:
#>        test
```

If you assign a `TRUE` to the `save_it` parameter, the barcode-list will be saved as a *csv*-file within the stated results directory (`results_dir`). Additionally, by setting the `seqLogo` parameter to `TRUE`, a sequence logo of the entire input file will be generated and also stored in the provided results directory.

2.1 Parallelization

Finally, the function offers a possibility to make the necessary calculations in a parallel fashion. The entire process is based on the R-package *future*, therefore you have to state the number of available CPUs (`cpus`) and the appropriate strategy (`strategy`). The default strategy will be `sequential` (meaning only one cpu will be used) but also `multisession` can be chosen (meaning more than one cpu will be used). You can either follow the link or consult the package internal help pages of the *future* package (by just type `?future::plan()` into the R-console) for more detailed informations.

2.2 Manipulating the *BCdat* data type

In order to allow for an easy workflow and to reduce the likelihood for ambiguity errors, the data type contains all important sample-specific data. The specifically adjusted `show()` function will provide you with a quick overview of the data set including metadata (number of barcode reads, read count distribution and sequence lengths, an excerpt of the most abundant barcodes, the path to the results directory, the used barcode backbone and the associated label).

```
  show(BC_data)
#>  class: BCdat
#>
#>   number of barcode sequences: 81
#>   read count distribution: min 3  mean 110.36  median 4  max 2405
#>   barcode sequence length: 32
#>
#>  barcode read counts:
#>  read_count                          barcode
#>        2405 CACGATCCGCTTCTATCGCGTGCACTACATGT
#>        1504 GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT
#>        1296 GCTAAGGGCGATCACATCCACAAGCTTCTTTG
#>        1274 CAGAATCGAATGATGTCTTCATCTCAACGCCG
#>        1058 AGTATTCCGACAGTGACATGTCTTCCCGCGTT
#>         968 GGCCTAGAGTAGTTGTCGCGAAAGTCGGCCTC
#>          45 AGTATTCCGACCGTGACATGTCTTCCCGCGTT
#>          37 CAGAATCGCATGATGTCTTCATCTCAACGCCG
#>          22 GGCCTAGCGTAGTTGTCGCGAAAGTCGGCCTC
#>          19 GCCATCTATATTTGTTCCAAGACTCTACTATT
#>                         ...
#>  results dir:
#>        /my/results/dir/
#>  barcode backbone:
#>        ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN
#>  label:
#>        test
```

The different elements of the data format can easily be accessed via the names of the particular slots.

```
  head(getReads(BC_data))
#>   read_count                          barcode
#> 1       2405 CACGATCCGCTTCTATCGCGTGCACTACATGT
#> 2       1504 GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT
#> 3       1296 GCTAAGGGCGATCACATCCACAAGCTTCTTTG
#> 4       1274 CAGAATCGAATGATGTCTTCATCTCAACGCCG
#> 5       1058 AGTATTCCGACAGTGACATGTCTTCCCGCGTT
#> 6        968 GGCCTAGAGTAGTTGTCGCGAAAGTCGGCCTC

  show(getResultsDir(BC_data))
#> [1] "/my/results/dir/"

  show(getBackbone(BC_data))
#> [1] "ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN"

  show(getLabel(BC_data))
#> [1] "test"
```

Also modifying or swapping the slot-specific data is easily possible.

```
  BC_data <- setReads(BC_data, data.frame(read_count = c(1:5), barcode = letters[1:5]))
  BC_data <- setResultsDir(BC_data, "/my/test/folder/")
  BC_data <- setBackbone(BC_data, "AAANNNNGGG")
  BC_data <- setLabel(BC_data, "new label")
```

2.3 Reading and Converting other data formats

There is also the possibility to re-analyze already stored barcode-lists via the function `readBCdat()`. You just have to define the path to the particular file of interest and the corresponding file name. And of course you can also provide a label and a barcode-backbone to complete the stored metadata. And if, for some reason, the stored data file is no standard *csv*-file with a semi-colon as separator, there is another parameter `s` which allows to specify the actual field separator.

```
  BC_data <- readBCdat(path = "/my/test/folder/",
                       label = "test",
                       BC_backbone = "AAANNNNCCCC",
                       file_name = "test.csv",
                       s = ";")
```

3. Error Correction

After extracting the barcode-carrying sequences, you can apply the list of detected barcodes to one of the available error-correction approaches. Since all the necessary analysis steps beforehand (like PCR and NGS) are naturally error-prone and therefore susceptible to small changes within the original nucleotide sequences, we recommend to cluster highly similar sequences together. Conveniently, you can use the *BCdat* object which was already created by the `processingRawData()` function as input for the `errorCorrection()` function.

```
BC_data_EC <- errorCorrection(BC_dat = BC_data,
                              maxDist = 4,
                              save_it = FALSE,
                              cpus = 1,
                              strategy = "sequential",
                              m = "hamming",
                              type = "standard",
                              only_EC_BCs = TRUE,
                              EC_analysis = FALSE,
                              start_small = TRUE)
```

Therefore, you have to provide a *BCdat* data-object (`BC_data`) and a numeric value (`maxDist`) specifying the amount of differences tolerated in order to cluster two barcode sequences together. Depending on the choice for `m`, which defines the distance measure (per default it's the Hamming distance), the `maxDist` value defines the maximum number of deviating nucleotides allowed. After we did a sensitivity analysis of the effect of the chosen Hamming-distance threshold and given the initial Hamming distance differences of the type of barcodes we are working with, we decided to use one fourth of the total barcode-length as a conservative Hamming-distance limit. The definition of such a threshold accounts for a successively increasing amount of single-nucleotide mutations due to several applied PCR-cycles, a potential loss of sequences due to the fact that only a part of the PCR-material will finally be sequenced and errors finally introduced by NGS. The exact value of this threshold should be adapted according to the used barcode construct, the number of initially marked cells and the particular complexity of the used barcode-library.
Another reasonable choice could be the Levenstein distance (`m = "lv"`) in which case the `maxDist` value will be interpreted as the maximum number of allowed edit operations. There are also further methods available, since this piece of the function is based on the *stringdist* R-package. A documentation can easily be found via `?'stringdist-metrics'`. All of the available error-correction approaches are based on an interative algorithm, therefore they are hardly parallelizable. Parallel computations are only feasible, if a list of *BCdat* objects is supplied. The function will then automatically initiate a parallel execution depending on the number of data-äobjects and the cpus available. The resulting data-object will then be a list of *BCdat* objects (in the same order as the backbone-sequences). Optionally, the final list of corrected barcode sequences can also be saved as a common *csv*-file (`save_it = TRUE`) in the previously specified results directory. The parameter `type` identifies the chosen error-correction approach (see the following section).

3.1 Error Correction Approaches

3.1.1 Standard

The `standard` error correction refers to, per default, a Hamming-distance based approach. Starting with the least abundant barcode *BC_small*, the Hamming distances to all other barcodes, in an increasing order of read counts, are calculated. If there is a highly-similar barcode (*BC_similar*, according to the chosen threshold `maxDist` and the distance measure `m`) the read counts of *BC_small* are added to those of *BC_similar* and the sequence of *BC_small* is dismissed. The list of barcodes will be sorted again for read counts after every "merging-event". The error-correction will always be applied starting from the least to the most abundant barcode. If there is more than one highly similar barcode present, the one with the smallest amount of read counts will always be selected first.

```
BC_data_EC <- errorCorrection(BC_dat = BC_data,
                              maxDist = 4,
                              save_it = FALSE,
                              cpus = 1,
                              strategy = "sequential",
                              m = "hamming",
                              type = "standard",
                              only_EC_BCs = TRUE,
                              EC_analysis = FALSE,
                              start_small = TRUE)
```

```
show(BC_data_EC)
#>  class: BCdat
#>
#>   number of barcode sequences: 9
#>   read count distribution: min 3  mean 993.22  median 1138  max 2503
#>   barcode sequence length: 32
#>
#>  barcode read counts:
#>  read_count                          barcode
#>        2503 CACGATCCGCTTCTATCGCGTGCACTACATGT
#>        1555 GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT
#>        1353 GCTAAGGGCGATCACATCCACAAGCTTCTTTG
#>        1352 CAGAATCGAATGATGTCTTCATCTCAACGCCG
#>        1138 AGTATTCCGACAGTGACATGTCTTCCCGCGTT
#>        1011 GGCCTAGAGTAGTTGTCGCGAAAGTCGGCCTC
#>          19 GCCATCTATATTTGTTCCAAGACTCTACTATT
#>           5 ATTGGGTCCGTCTGAGGGCGTTTCTGCGCCTT
#>           3 CCGCATCTTCTTATCCTGTAGTTCACTATCTC
#>
#>  results dir:
#>        /my/results/dir/
#>  barcode backbone:
#>        ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN
#>  label:
#>        test_EC
```

If, in combination with `type = "standard"` also the parameter *start_small* was set to `FALSE`, the algorithm will now in case of more than one highly similar barcode, select the most abundant one first.

```
BC_data_EC <- errorCorrection(BC_dat = BC_data,
                              maxDist = 4,
                              save_it = FALSE,
                              cpus = 1,
                              strategy = "sequential",
                              m = "hamming",
                              type = "standard",
                              only_EC_BCs = TRUE,
                              EC_analysis = FALSE,
                              start_small = FALSE)
```

```
  show(BC_data_EC)
#>  class: BCdat
#>
#>   number of barcode sequences: 9
#>   read count distribution: min 3  mean 993.22  median 1138  max 2503
#>   barcode sequence length: 32
#>
#>  barcode read counts:
#>  read_count                              barcode
#>        2503 CACGATCCGCTTCTATCGCGTGCACTACATGT
#>        1555 GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT
#>        1353 GCTAAGGGCGATCACATCCACAAGCTTCTTTG
#>        1352 CAGAATCGAATGATGTCTTCATCTCAACGCCG
#>        1138 AGTATTCCGACAGTGACATGTCTTCCCGCGTT
#>        1011 GGCCTAGAGTAGTTGTCGCGAAAGTCGGCCTC
#>          19 GCCATCTATATTTGTTCCAAGACTCTACTATT
#>           5 ATTGGGTCCGTCTGAGGGCGTTTCTGCGCCTT
#>           3 CCGCATCTTCTTATCCTGTAGTTCACTATCTC
#>
#>  results dir:
#>        /my/results/dir/
#>  barcode backbone:
#>        ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN
#>  label:
#>        test_EC
```

3.1.2 Graph based

The `graph based` error-correction is based on a graph-theoretic approach. Firstly, for each and every barcode the distances to all the other barcodes will be calculated, all distances `> maxDist` will then be set to zero and all distances `=< maxDist` will be set to one. Secondly, the resulting matrix serves as an adjacency matrix for which existing connected components can be identified. And finally, all of the *member-barcodes* of each of those components will then be clustered together, with the most abundant barcode as the respective cluster-label.

```
  BC_data_EC <- errorCorrection(BC_dat = BC_data,
                                maxDist = 4,
                                save_it = FALSE,
                                cpus = 1,
                                strategy = "sequential",
                                m = "hamming",
                                type = "graph based",
                                only_EC_BCs = TRUE,
                                EC_analysis = FALSE,
                                start_small = FALSE)
```

```
  show(BC_data_EC)
#>  class: BCdat
#>
#>   number of barcode sequences: 9
#>   read count distribution: min 3  mean 993.22  median 1138  max 2503
#>   barcode sequence length: 32
```

```
#>
#>  barcode read counts:
#>  read_count                          barcode
#>        2503 CACGATCCGCTTCTATCGCGTGCACTACATGT
#>        1555 GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT
#>        1353 GCTAAGGGCGATCACATCCACAAGCTTCTTTG
#>        1352 CAGAATCGAATGATGTCTTCATCTCAACGCCG
#>        1138 AGTATTCCGACAGTGACATGTCTTCCCGCGTT
#>        1011 GGCCTAGAGTAGTTGTCGCGAAAGTCGGCCTC
#>          19 GCCATCTATATTTGTTCCAAGACTCTACTATT
#>           5 ATTGGGTCCGTCTGAGGGCGTTTCTGCGCCTT
#>           3 CCGCATCTTCTTATCCTGTAGTTCACTATCTC
#>
#>  results dir:
#>        /my/results/dir/
#>  barcode backbone:
#>        ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN
#>  label:
#>        test_EC
```

3.1.3 Connectivity based

The `connectivity based` error-correction is very similar to the `standard` approach but instead of ordering the available barcodes by their read count values and starting the clustering with the least frequent one, it now will be ordered by the amount of highly-similar analogues for each barcode (again depending on the chosen distance measure `m` and the threshold `maxDist`). Consequently, the clustering will now start with the barcode possessing the lowest number of highly-similar counterparts.

```
BC_data_EC <- errorCorrection(BC_dat = BC_data,
                              maxDist = 4,
                              save_it = FALSE,
                              cpus = 1,
                              strategy = "sequential",
                              m = "hamming",
                              type = "connectivity based",
                              only_EC_BCs = TRUE,
                              EC_analysis = FALSE,
                              start_small = FALSE)
```

```
show(BC_data_EC)
#>  class: BCdat
#>
#>   number of barcode sequences: 9
#>   read count distribution: min 3  mean 993.22  median 1138  max 2503
#>   barcode sequence length: 32
#>
#>  barcode read counts:
#>  read_count                          barcode
#>        2503 CACGATCCGCTTCTATCGCGTGCACTACATGT
#>        1555 GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT
#>        1353 GCTAAGGGCGATCACATCCACAAGCTTCTTTG
#>        1352 CAGAATCGAATGATGTCTTCATCTCAACGCCG
```

```
#>        1138 AGTATTCCGACAGTGACATGTCTTCCCGCGTT
#>        1011 GGCCTAGAGTAGTTGTCGCGAAAGTCGGCCTC
#>          19 GCCATCTATATTTGTTCCAAGACTCTACTATT
#>           5 ATTGGGTCCGTCTGAGGGCGTTTCTGCGCCTT
#>           3 CCGCATCTTCTTATCCTGTAGTTCACTATCTC
#>
#>  results dir:
#>        /my/results/dir/
#>  barcode backbone:
#>        ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN
#>  label:
#>        test_EC
```

3.1.4 Clustering

The error-correction type called `clustering` takes the most abundant barcode, then identifies all highly-similar counterparts (again based on the method `m` and the threshold `maxDist`), adds up all corresponding read-counts to the most-abundant one and finally dismisses all of those added up barcode sequences. Then, the procedure continues with the second most-abundant barcode until all barcodes are processed.

```
BC_data_EC <- errorCorrection(BC_dat = BC_data,
                              maxDist = 4,
                              save_it = FALSE,
                              cpus = 1,
                              strategy = "sequential",
                              m = "hamming",
                              type = "clustering",
                              only_EC_BCs = TRUE,
                              EC_analysis = FALSE,
                              start_small = FALSE)
```

```
show(BC_data_EC)
#>  class: BCdat
#>
#>   number of barcode sequences: 9
#>   read count distribution: min 3  mean 993.22  median 1138  max 2503
#>   barcode sequence length: 32
#>
#>  barcode read counts:
#>  read_count                          barcode
#>        2503 CACGATCCGCTTCTATCGCGTGCACTACATGT
#>        1555 GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT
#>        1353 GCTAAGGGCGATCACATCCACAAGCTTCTTTG
#>        1352 CAGAATCGAATGATGTCTTCATCTCAACGCCG
#>        1138 AGTATTCCGACAGTGACATGTCTTCCCGCGTT
#>        1011 GGCCTAGAGTAGTTGTCGCGAAAGTCGGCCTC
#>          19 GCCATCTATATTTGTTCCAAGACTCTACTATT
#>           5 ATTGGGTCCGTCTGAGGGCGTTTCTGCGCCTT
#>           3 CCGCATCTTCTTATCCTGTAGTTCACTATCTC
#>
#>  results dir:
#>        /my/results/dir/
```

```
#>  barcode backbone:
#>        ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN
#>  label:
#>        test_EC
```

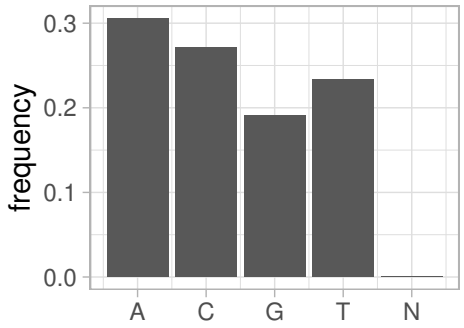
4. Visualizations

After extracting and preprocessing the barcode sequences, there are a variety of visualizations available.

4.1 Fastq Quality Plots

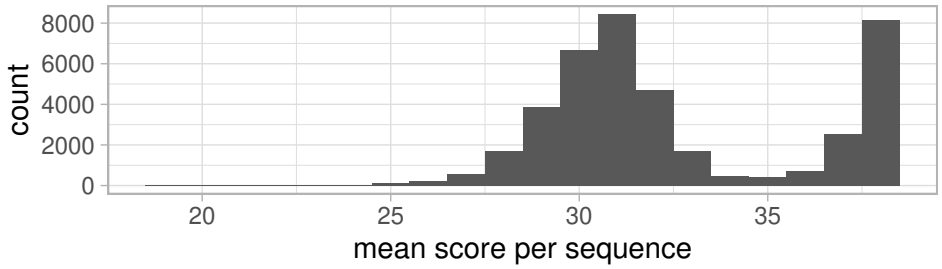
Besides visualizing the extracted barcodes, their frequencies and sequence similarities, there are also plot-functions for checking the quality of the initial fastq-file. The function `plotNucFrequency()` allows an overview of the nucleotide frequencies over the entire raw data file.

```
s_dir <- system.file("extdata", package = "genBaRcode")

plotNucFrequency(source_dir = s_dir, file_name = "test_data.fastq.gz")
```

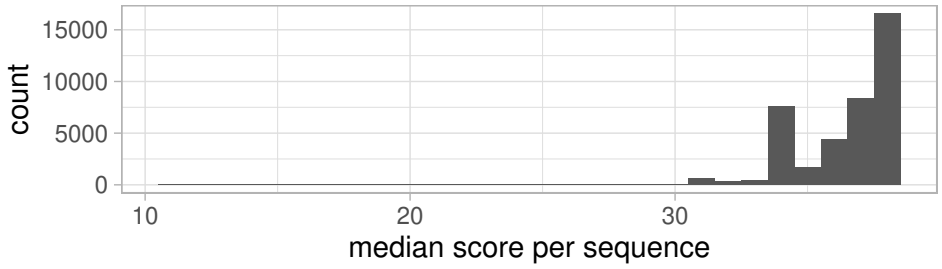


It is also possible to create a histogram of the *mean* or *median* quality-score distribution over all reads within the *fastq*-file, using the function `plotQualityScoreDis()`.

```
plotQualityScoreDis(source_dir = s_dir, file_name = "test_data.fastq.gz", type = "mean")
```

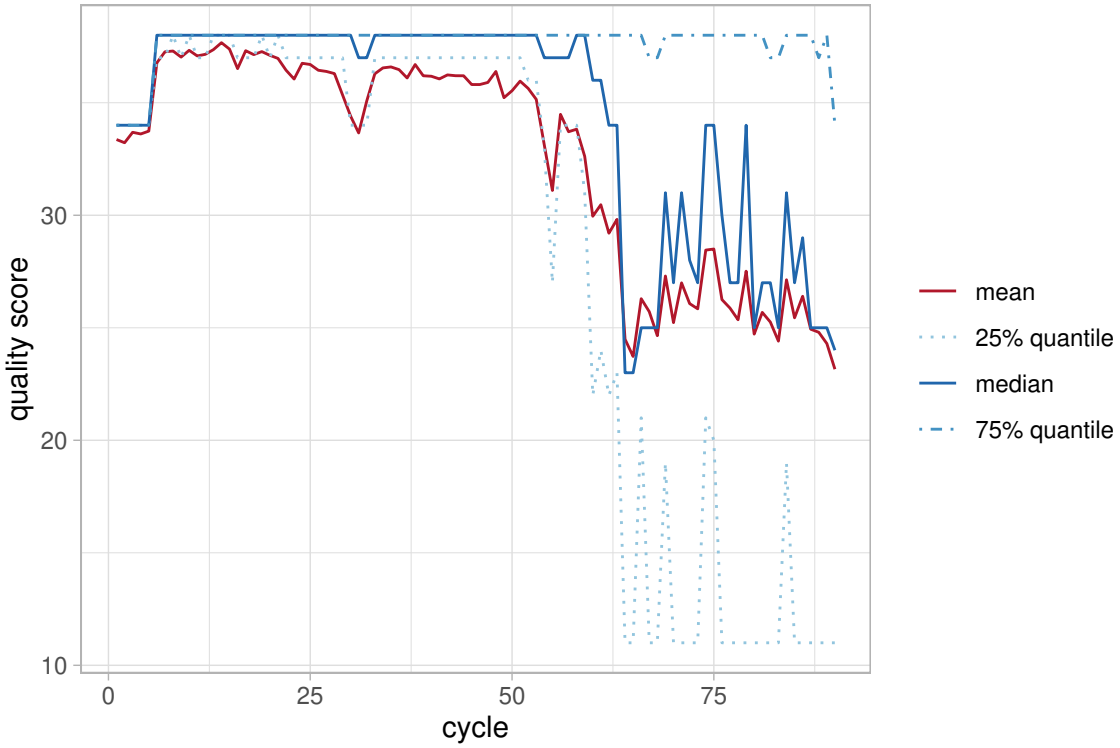



```
plotQualityScoreDis(source_dir = s_dir, file_name = "test_data.fastq.gz", type = "median")
```



Furthermore, the function `plotQualityScorePerCycle()` allows for a read-position specific visualization of the *mean* or *median* sequence quality.

```
plotQualityScorePerCycle(source_dir = s_dir, file_name = "test_data.fastq.gz")
```



Additionally, there is a sequence logo function available providing the opportunity to visually inspect all NGS reads at once.

```
show(BC_data)
#>  class: BCdat
#>
#>   number of barcode sequences: 81
#>   read count distribution: min 3  mean 110.36  median 4  max 2405
#>   barcode sequence length: 32
#>
#>  barcode read counts:
#>  read_count                              barcode
#>        2405 CACGATCCGCTTCTATCGCGTGCACTACATGT
#>        1504 GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT
#>        1296 GCTAAGGGCGATCACATCCACAAGCTTCTTTG
#>        1274 CAGAATCGAATGATGTCTTCATCTCAACGCCG
#>        1058 AGTATTCCGACAGTGACATGTCTTCCCGCGTT
#>         968 GGCCTAGAGTAGTTGTCGCGAAAGTCGGCCTC
#>          45 AGTATTCCGACCGTGACATGTCTTCCCGCGTT
#>          37 CAGAATCGCATGATGTCTTCATCTCAACGCCG
#>          22 GGCCTAGCGTAGTTGTCGCGAAAGTCGGCCTC
#>          19 GCCATCTATATTTGTTCCAAGACTCTACTATT
#>                        ...
```

```
#>  results dir:
#>        /my/results/dir/
#>  barcode backbone:
#>        ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN
#>  label:
#>        test
```

```
plotSeqLogo(BC_dat = BC_data, colrs = NULL)
```

It is also possible to adjust the color selection for each nucleotide.

```
# color order correlates to the following nucleotide order A, T, C, G, N
col_vec <- c("#000000",
             "#000000",
             RColorBrewer::brewer.pal(6, "Paired")[c(5, 6)],
             "#000000")
show(col_vec)
#> [1] "#000000" "#000000" "#FB9A99" "#E31A1C" "#000000"

plotSeqLogo(BC_dat = BC_data, colrs = col_vec)
```

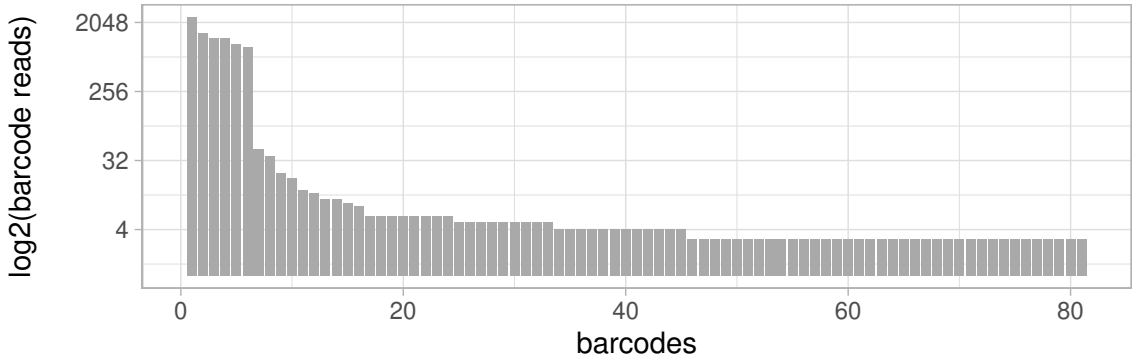
4.2 Plotting Barcode Frequencies

The function `generateKirchenplot()` offers the possibility to explore the barcode frequencies of the analyzed sample.

```
show(BC_data)
#>  class: BCdat
#>
#>   number of barcode sequences: 81
#>   read count distribution: min 3  mean 110.36  median 4  max 2405
#>   barcode sequence length: 32
#>
#>  barcode read counts:
#>  read_count                              barcode
#>        2405 CACGATCCGCTTCTATCGCGTGCACTACATGT
#>        1504 GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT
```

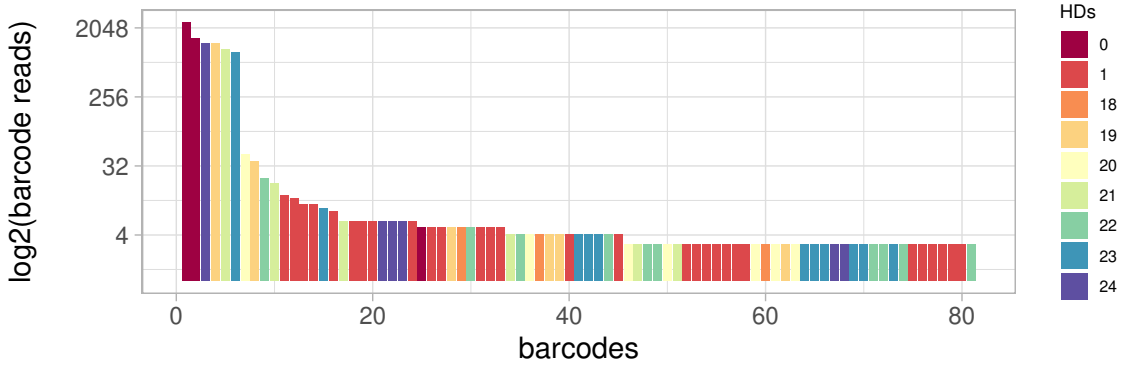
```
#>        1296 GCTAAGGGCGATCACATCCACAAGCTTCTTTG
#>        1274 CAGAATCGAATGATGTCTTCATCTCAACGCCG
#>        1058 AGTATTCCGACAGTGACATGTCTTCCCGCGTT
#>         968 GGCCTAGAGTAGTTGTCGCGAAAGTCGGCCTC
#>          45 AGTATTCCGACCGTGACATGTCTTCCCGCGTT
#>          37 CAGAATCGCATGATGTCTTCATCTCAACGCCG
#>          22 GGCCTAGCGTAGTTGTCGCGAAAGTCGGCCTC
#>          19 GCCATCTATATTTGTTCCAAGACTCTACTATT
#>                        ...
#>  results dir:
#>        /my/results/dir/
#>  barcode backbone:
#>        ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN
#>  label:
#>        test
```

```
generateKirchenplot(BC_dat = BC_data)
```



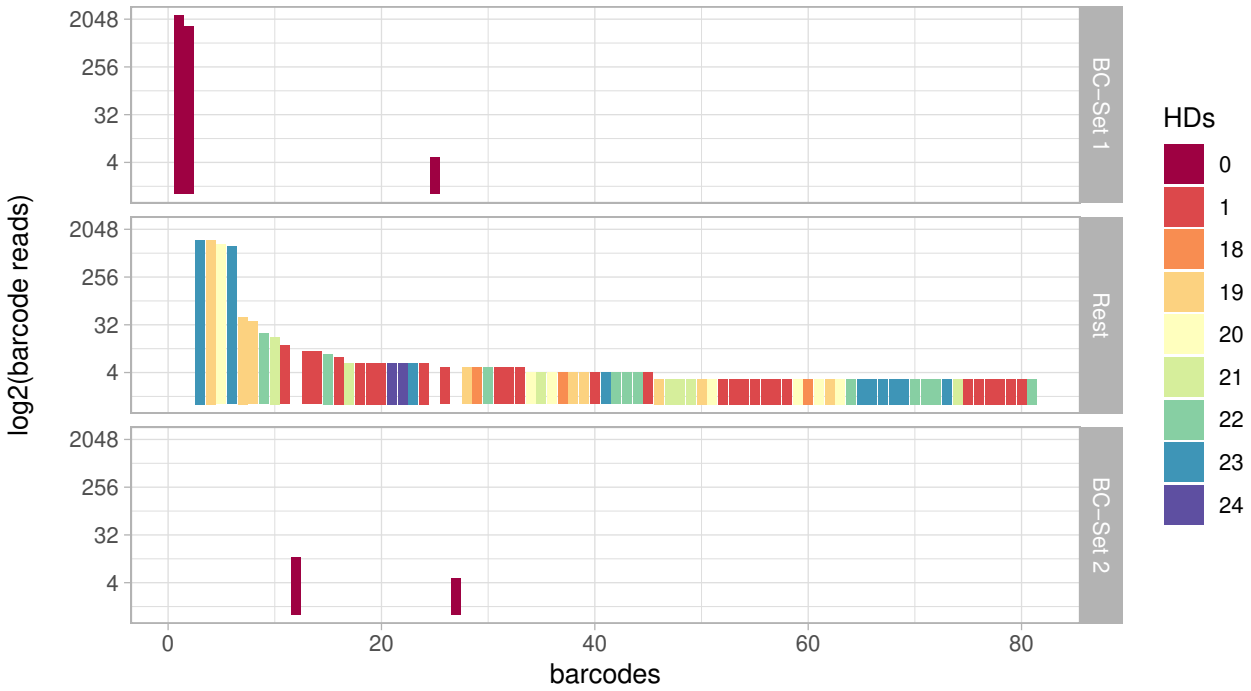
In certain cases, if you are particularly interested in previously known barcode sequences, you can provide those sequences woth the function call and therefore introducing an additional color-coding visualizing sequence (dis-)similarities.

```
known_BCs <- c("GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT",
               "CACGATCCGCTTCTATCGCGTGCACTACATGT",
               "ATTGGGTCCGTCTGAGGGCGTTTCTGCGCCTT")

generateKirchenplot(BC_dat = BC_data, ori_BCs = known_BCs)
```



Furthermore, if you have two different barcode sets, e.g. a barcode white-list and known contaminations, both of those sets can be provided and will then automatically lead to separate plots.

```
known_BCs <- c("GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT",
               "CACGATCCGCTTCTATCGCGTGCACTACATGT",
               "ATTGGGTCCGTCTGAGGGCGTTTCTGCGCCTT")
contaminations <- c("CACGATCCGCTTCTATCGCGTGCACTACATGC",
                    "ATTGGGTCCGTCTGAGGGCGTCTCTGCGCCTT",
                    "CACGATCCGCTTCTATCGCGTGCGCTACATGT",
                    "TACGATCCGCTTCTATCGCGTGCACTACATGT")

generateKirchenplot(BC_dat = BC_data, ori_BCs = known_BCs, ori_BCs2 = contaminations)
```

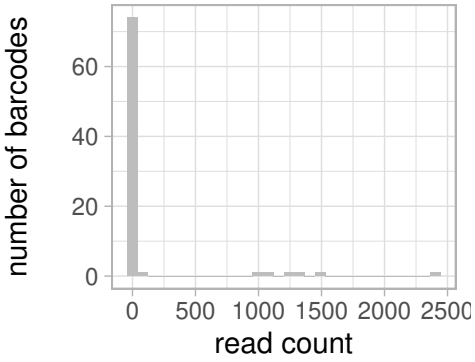


This function also offers the possibilities to change the scale of the y-axis (`loga`), the distance measure (`m`), the color palettes (`col_type`) and of course the corresponding labels (`setLabels`).

```
known_BCs <- c("GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT",
               "CACGATCCGCTTCTATCGCGTGCACTACATGT",
               "ATTGGGTCCGTCTGAGGGCGTTTCTGCGCCTT")
contaminations <- c("CACGATCCGCTTCTATCGCGTGCACTACATGC",
                    "ATTGGGTCCGTCTGAGGGCGTCTCTGCGCCTT",
                    "CACGATCCGCTTCTATCGCGTGCGCTACATGT",
                    "TACGATCCGCTTCTATCGCGTGCACTACATGT")

generateKirchenplot(BC_dat = BC_data,
                    ori_BCs = known_BCs, ori_BCs2 = contaminations,
                    setLabels = c("known BCs", "stuff", "contaminations"),
                    loga = TRUE, col_type = "wild", m = "lv")
```

Another interesting feature of a data set concerns the read-count frequencies. The following function offers the possibility to plot read-counts as absolute numbers.

```
plotReadFrequencies(BC_dat = BC_data)
```



. . . or as log-values or the corresponding density.

```
plotReadFrequencies(BC_dat = BC_data, log = TRUE)
plotReadFrequencies(BC_dat = BC_data, dens = TRUE)
```

Also the width of the bins (`bw`) or the number of the bins (`b`) are adjustable.

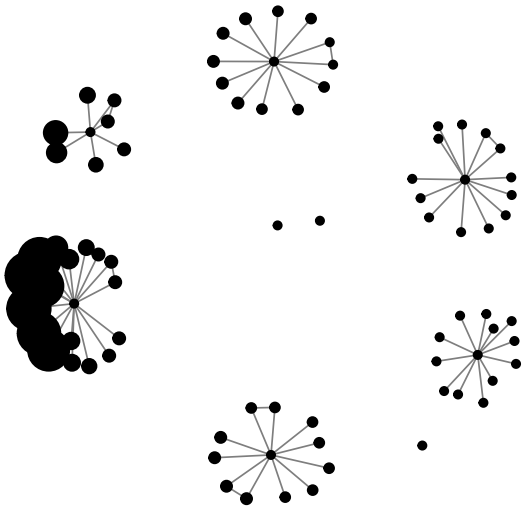
```
plotReadFrequencies(BC_dat = BC_data, bw = 30)
plotReadFrequencies(BC_dat = BC_data, b = 30)
```

4.3 Plotting Barcode Relations

You can also have a look at the sequence (dis-)similarities of the detected barcode sequences in a graph-like plot. Those plots can be used to identify false-positive barcodes and also to help visualizing the actions of the chosen error-correction approach. The underlying idea is again based on some kind of similarity/distance measure. Therefore, every barcode sequence will be compared to all other sequences and based on the chosen similarity measure the distances will be calculated, per default the Hamming distance will be predefined. The nodes within those network-plots represent the detected barcode sequences and (per default) every edge symbolizes a distance between two barcodes of exactly one nucleotide difference (adjustable via the parameter `minDist`). There are different functions available, which basically do the same but are based on different R-packages. The `plotDistanceIgraph()` function is based on the *igraph* package and therefore will also return an *igraph*-object whereas the `ggplotDistanceGraph()` function offers the possibility to create an *ggplot2*-object which than can be further customized. The `plotDistanceVisNetwork()` function is based in the *visNetwork* package and will consequently return a *visNetwork*-object which allows for an interactive exploration of the resulting plot. For instance, you can adjust the layout by clicking and dragging specific network nodes or by hovering the mouse-arrow over a node it is also possible to retrieve the underlying barcode metadata. Additionally, it is also possible to zoom-in and out and see all the corresponding barcode sequences at once.

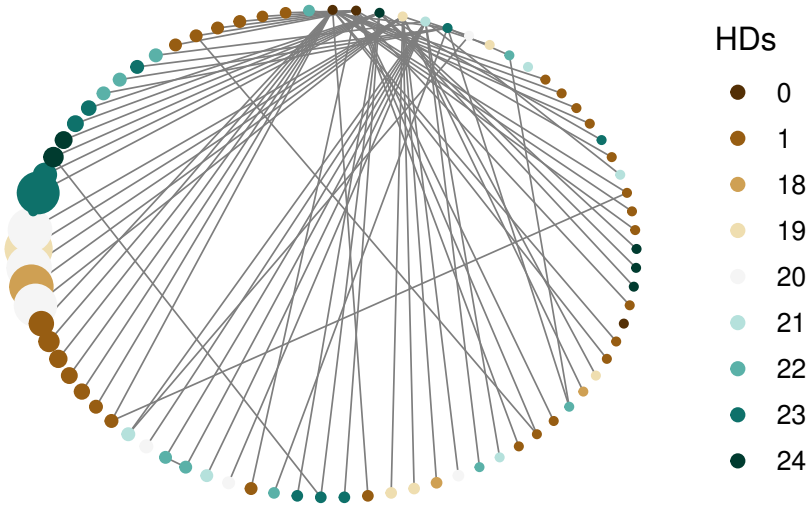
```
plotDistanceVisNetwork(BC_dat = BC_data, minDist = 1, loga = TRUE, m = "hamming")
plotDistanceIgraph(BC_dat = BC_data, minDist = 1, loga = TRUE, m = "hamming")
```

```
ggplotDistanceGraph(BC_dat = BC_data, minDist = 1, loga = TRUE, m = "hamming")
```



Also, there are additional parameters available. You can provide a list of particular interesting barcodes (`ori_BCs`) which will introduce a similarity based color-coding. The parameter called `lay` allows for the selection of different layout algorithms or, instead of providing a name of the layout algorithm, you can provide a two-column matrix with as many rows as there are nodes in the network, in which case the matrix is used as layout node-coordinates. Default value for `lay` is *fruchtermanreingold*, but possible are also *circle*, *eigen*, *kamadakawai*, *spring* and many more. Additionally, you can choose a custom color palette (*rainbow*, *heat.colors*, *topo.colors*, *greens*, *wild* - see package *grDevices*). And finally, the parameter `legend_size` allows for an adjustment of legend size.

```
known_BCs <- c("GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT",
               "CACGATCCGCTTCTATCGCGTGCACTACATGT",
               "ATTGGGTCCGTCTGAGGGCGTTTCTGCGCCTT")

ggplotDistanceGraph(BC_dat = BC_data,
                    minDist = 1, loga = TRUE, m = "hamming",
                    ori_BCs = known_BCs, lay = "circle", complete = FALSE,
                    col_type = "topo.colors", legend_size = 2)
```

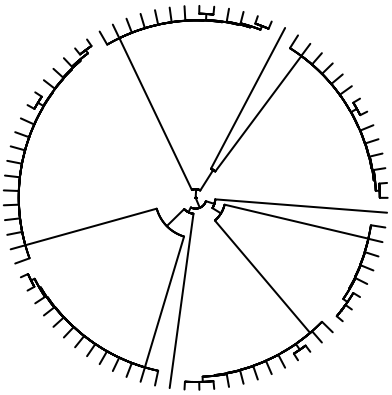


For further customization possibilities the function `createGDF()` will create a *gdf*-file which then can be used as input for an open-source and free software called *Gephi*.

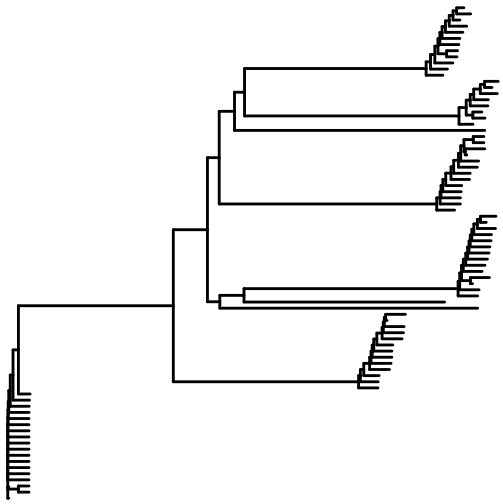
```
createGDF(BC_dat = BC_data, minDist = 1, loga = TRUE, m = "hamming")
```

There is also the possibility to use the same conceptual approach but not visualizing the barcodes as nodes in a network but as branches in a tree.

```
plotClusterTree(BC_dat = BC_data, tree_est = "UPGMA",
                type = "fan", tipLabel = FALSE, m = "hamming")
```



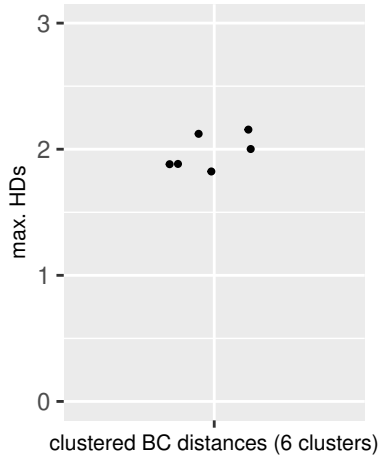
```
plotClusterGgTree(BC_dat = BC_data, tree_est = "NJ",
                  type = "rectangular", m = "hamming")
#> Registered S3 method overwritten by 'treeio':
#>   method     from
#>   root.phylo ape
#> Registered S3 method overwritten by 'ggtree':
#>   method         from
#>   fortify.igraph ggnetwork
#> Warning: `data_frame()` is deprecated as of tibble 1.1.0.
#> Please use `tibble()` instead.
#> This warning is displayed once every 8 hours.
#> Call `lifecycle::last_warnings()` to see where this warning was generated.
```



4.4 Plotting Error Correction Results

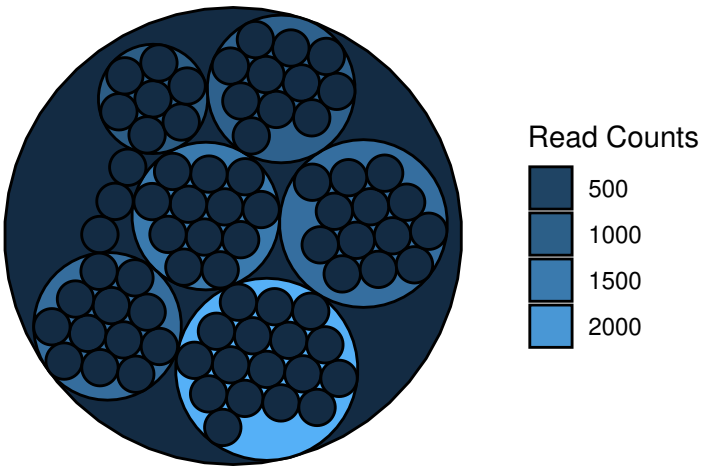
Furthermore, the package comes with a variety of functions solely designed to inspect the "insides"" of the error-correction approaches. Each error-correction basically clusters the input sequences based on slightly different "assumptions". To retrospectively inspect the resulting cluster compositions, e.g. the function `error_correction_clustered_HDs()` will visualize the sequences similarities in the respective clusters. In order to do so, you have to set the `EC_analysis` paramter of the `errorCorrection()`-function to `TRUE`.

```
BC_data_EC <- errorCorrection(BC_dat = BC_data,
                              maxDist = 4,
                              save_it = FALSE,
                              cpus = 1,
                              strategy = "sequential",
                              m = "hamming",
                              type = "standard",
                              only_EC_BCs = FALSE,
                              EC_analysis = TRUE,
                              start_small = FALSE)

error_correction_clustered_HDs(datEC = BC_data_EC, size = 0.75)
```

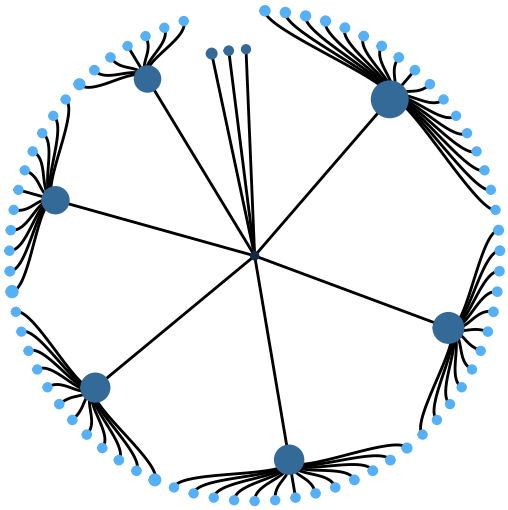


You can also visualize the cluster composition in a more graph-like plot and if you are only interested in the final outcome of the error-correction procedure, it is possible to set the `only_EC_BCs` parameter to `TRUE`. Otherwise, all initial barcodes and the iterative nature of the error-correction procedure will be part of the additional metadata.

```
error_correction_circlePlot(edges = BC_data_EC$edges, vertices = BC_data_EC$vertices)
```

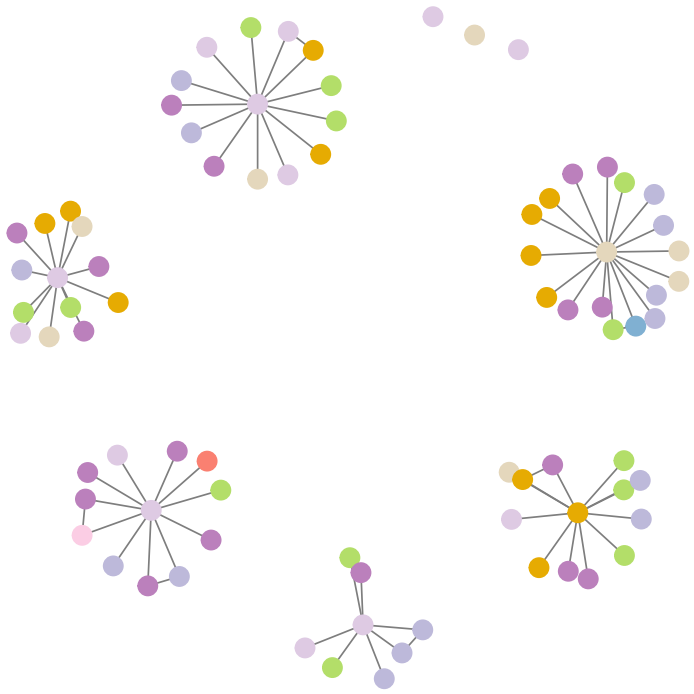



```
error_correction_treePlot(edges = BC_data_EC$edges, vertices = BC_data_EC$vertices)
```



But you can also choose the already presented *ggplot2* and *visNetwork* based plots to have a look at the subsumed barcode sequences during error-correction.

```
ggplotDistanceGraph_EC(BC_dat = BC_data, BC_dat_EC = BC_data_EC,
                       minDist = 1, loga = TRUE, m = "hamming")
```

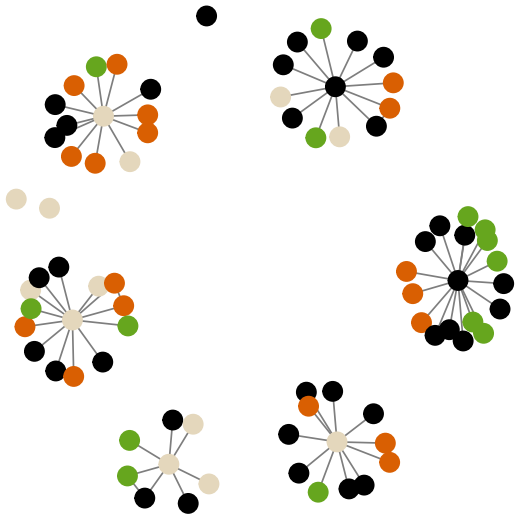


Or.

```
plotDistanceVisNetwork_EC(BC_dat = BC_data, BC_dat_EC = BC_data_EC,
                          minDist = 1, loga = TRUE, m = "hamming")
```

And here again, you can define a barcode list of interest to limit the amount of color-coding.

```
known_BCs <- c("GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT",
               "CACGATCCGCTTCTATCGCGTGCACTACATGT",
               "ATTGGGTCCGTCTGAGGGCGTTTCTGCGCCTT")

ggplotDistanceGraph_EC(BC_dat = BC_data, BC_dat_EC = BC_data_EC,
                       minDist = 1, loga = TRUE, m = "hamming", ori_BCs = known_BCs)
```

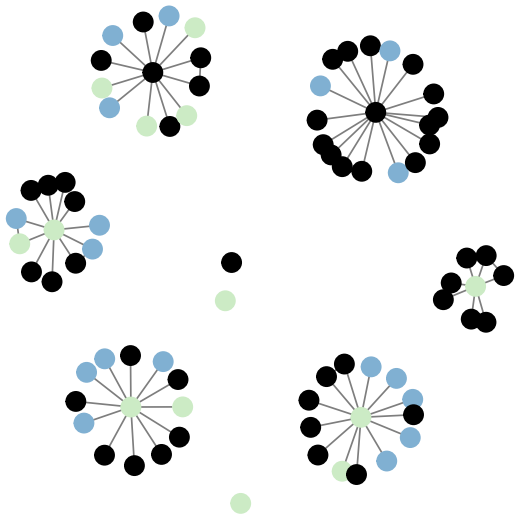


Or.

```
plotDistanceVisNetwork_EC(BC_dat = BC_data, BC_dat_EC = BC_data_EC,
                          minDist = 1, loga = TRUE, m = "hamming", ori_BCs = known_BCs)
```

Or you can just limit the number of regarded barcodes by defining how many of the most abundant barcodes you would like to inspect (in this example, we limited it to the two most-abundant once, `BC_threshold = 2`).

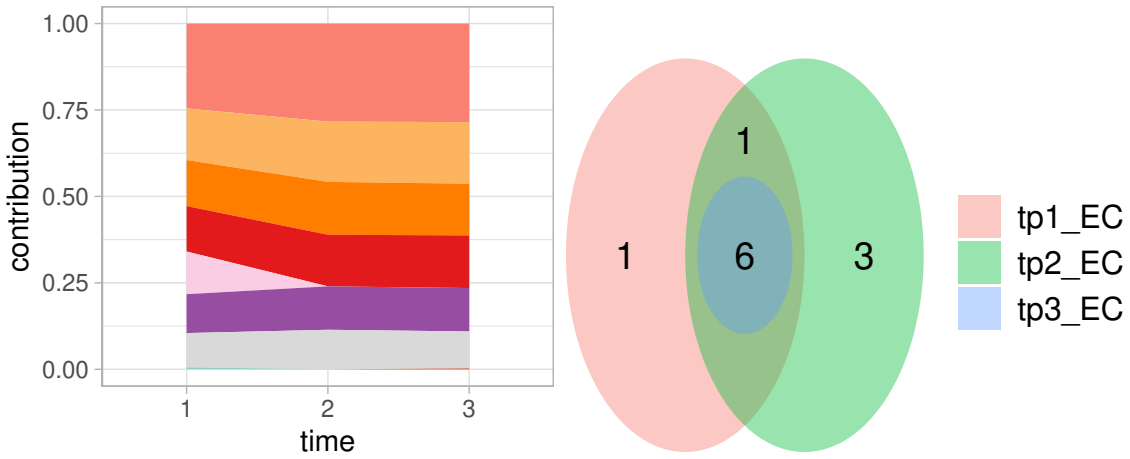
```
known_BCs <- c("GGTCGAAGCTTCTTTCGGGCCGCACGGCTGCT",
               "CACGATCCGCTTCTATCGCGTGCACTACATGT",
               "ATTGGGTCCGTCTGAGGGCGTTTCTGCGCCTT")

ggplotDistanceGraph_EC(BC_dat = BC_data, BC_dat_EC = BC_data_EC,
                       minDist = 1, loga = TRUE, m = "hamming", BC_threshold = 2)
```



5. Generating and plotting Time Series Data

The package also provides the possibility to analyze time series data. Let's assume there are different fastq-files for different points in time, you have to start with the analysis of each file separately. After that, all the separately generated *BCdat* objects have to be arranged in a list in the time-correct order. If you already provided all of the fastq-files to the `processingRawData()` function at once, the resulting *BCdat*-objects will already be arranged in a list. Now, you just have to use that list with the function `generateTimeSeriesData()`. The resulting output can then be visualized utilizing the `plotTimeSeries()` and the `plotVennDiagram()` function.

```
  # path to the package internal data file
  source_dir <- system.file("extdata", package = "genBaRcode")

  BC_data_tp1 <- processingRawData(file_name = "test_data.fastq.gz",
                                   source_dir,
                                   mismatch = 10,
                                   label = "tp1",
                                   bc_backbone = getBackboneSelection(1),
                                   bc_backbone_label = "BC_1",
                                   min_score = 10,
                                   save_it = FALSE)
#> Warning in .local(x, row.names, optional, ...): 'optional' argument was ignored
  BC_data_tp1 <- errorCorrection(BC_data_tp1, maxDist = 2)

  BC_data_tp2 <- processingRawData(file_name = "test_data.fastq.gz",
                                   source_dir,
                                   mismatch = 1,
                                   label = "tp2",
                                   bc_backbone = getBackboneSelection(1),
                                   bc_backbone_label = "BC_1",
                                   min_score = 30,
                                   min_reads = 1000,
                                   save_it = FALSE)
#> Warning in .local(x, row.names, optional, ...): 'optional' argument was ignored
  BC_data_tp2 <- errorCorrection(BC_data_tp2, maxDist = 4, type = "clustering")

  BC_data_tp3 <- processingRawData(file_name = "test_data.fastq.gz",
                                   source_dir,
                                   mismatch = 0,
                                   label = "tp3",
                                   bc_backbone = getBackboneSelection(1),
                                   bc_backbone_label = "BC_1",
                                   min_score = 37,
                                   save_it = FALSE)
  BC_data_tp3 <- errorCorrection(BC_data_tp3, maxDist = 8, type = "graph based")

  BC_list <- list(BC_data_tp1, BC_data_tp2, BC_data_tp3)
  BC_matrix <- generateTimeSeriesData(BC_dat_list = BC_list)
  plotTimeSeries(ov_dat = BC_matrix)
#> # normalization of read count data
  plotVennDiagram(BC_dat = BC_list)
```



As usual, also those two visualization functions come with a variety of layout options.

```
# choose colors
test_colors <- RColorBrewer::brewer.pal(12, "Set3")

plotTimeSeries(ov_dat = BC_matrix[1:12, ],
               colr = test_colors, tp = c(1,3,4),
               x_label = "test data", y_label = "test freqs")

plotVennDiagram(BC_dat = BC_list, alpha_value = 0.25,
                colrs = c("green", "red", "blue"), border_color = "orange",
                plot_title = "this is the title",
                legend_sort = c("tp2_EC", "tp3_EC", "tp1_EC"),
                annotationSize = 2.5)
```

6. genBaRcode Shiny-App

There is also a shiny-app included within the package, allowing the user to use all main functionality of the package without typing any line of code at all. Or if you are well capable of programming you can also use it as a convenient method to learn about the possibilities of the package. There is an app-internal help and also an option to inspect the source-code necessary to redo all in-app done analysis. You can start the app with the `genBaRcode_app()` command and if you already have a data file which you are dying to analyze you just need to provide the path to the directory (`dat_dir`) of this particular file in order to choose it from within the app. If you have none and no path is provided the package‘s internal example file will be available for exemplary analysis.

```
# start Shiny app with the package internal test data file
genBaRcode_app()

# start Shiny app with access to a predefined directory
genBaRcode_app(dat_dir = "/my/test/directory/")
```

For more detailed information’s please consult the app-specific vignette.

7. Miscellaneous Functions

Since the package is closely related to several different and already established genetic barcodes, there is also a function which offers all established barcode-backbones, its called `getBackboneSelection()`. This is a convenient way to directly input those, sometimes lengthy, barcode backbones directly into the `processingRawData()` function.

```
  getBackboneSelection()
#>   name
#> 1 BC32-GFP
#> 2 BC32-Venus
#> 3 BC32-eBFP
#> 4 BC32-T-Sapphire
#> 5 BC16-GFP
#> 6 BC16-Venus
#> 7 BC16-mCherry
#> 8 BC16-Cerulean
#>   sequences
#> 1 ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN
#> 2 CGANNAGANNCTTNNCGANNCTANNGGANNCTTNNCGANNAGANNCTTNNCGANNCTANNGGANNCTTNNCGANNAGANN
#> 3 CTANNCAGNNCTTNNCGANNCTANNCTTNNGGANNCTANNCAGNNCTTNNCGANNCTANNCTTNNGGANNCTANNCAGNN
#> 4 CAGNNATCNNCTTNNCGANNGGANNCTANNCTTNNCAGNNATCNNCTTNNCGANNGGANNCTANNCTTNNCAGNNATCNN
#> 5 ATCNNTAGNNTCCNNAAGNNTCGNNAAGNNTCGNNAGTNNTAG
#> 6 CTANNCTANNCAGNNCTTNNCGANNCTANNCTTNNGGANNGAT
#> 7 CTANNCAGNNATCNNCTTNNCGANNGGANNCTANNCTTNNGAT
#> 8 CTANNCACNNAGANNCTTNNCGANNCTANNGGANNCTTNNGAT

  bb <- getBackboneSelection(1)
  show(bb)
#> [1] "ACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANNCTTNNCGANNCTTNNGGANNCTANNACTNNCGANN"

  bb <- getBackboneSelection("BC32-eBFP")
  show(bb)
#> [1] "CTANNCAGNNCTTNNCGANNCTANNCTTNNGGANNCTANNCAGNNCTTNNCGANNCTANNCTTNNGGANNCTANNCAGNN"
```

If you would like to revisit some already analysed data-files you can just read the already stored *csv*-files via the `readBCdat()` function. The function will read the file and return a *BCdat*-object. And since there are different *csv*-like file formats out there, it is also possible to state the file-specific separator symbol via the parameter `s`.

```
  BC_data <- readBCdat(path = "/my/test/firectory", label = "test_label", s = ";",
                       BC_backbone = "ACTNNGGCNNTGANN", file_name = "test_file.csv")
```

But you can also just transform a `data.frame()` into a valid *BCdat*-object.

```
  test_data_frame <- data.frame(read_count = seq(100, 400, 100),
                                barcode = c("AAAAAAAA", "GGGGGGGG",
                                            "TTTTTTTT", "CCCCCCCC"))

  BC_data <- asBCdat(dat = test_data_frame,
                     label = "test_label",
                     BC_backbone = "CCCNNAAANNTTTNNGGGNN",
                     resDir = "/my/results/directory/")
```

Furthermore, in case there is a need for a direct comparison of two *BCdat*-objects and you have no clue how to do that you can try the `com_pair()` function.

```
  test_data_frame <- data.frame(read_count = seq(100, 400, 100),
                                barcode = c("AAAAAAAA", "GGGGGGGG",
                                            "TTTTTTTT", "CCCCCCCC"))
```

```
  show(test_data_frame)
#>   read_count  barcode
#> 1        100 AAAAAAAA
#> 2        200 GGGGGGGG
#> 3        300 TTTTTTTT
#> 4        400 CCCCCCCC
```

```
  BC_data_1 <- asBCdat(dat = test_data_frame,
                       label = "test_label_1",
                       BC_backbone = "CCCNNAAANNTTTNNGGGNN",
                       resDir = getwd())

  test_data_frame <- data.frame(read_count = c(300, 99, 150, 400),
                                barcode = c("TTTTTTTT", "AATTTAAA",
                                            "GGGGGGGG", "CCCCCCCC"))
```

```
  show(test_data_frame)
#>   read_count  barcode
#> 1        300 TTTTTTTT
#> 2         99 AATTTAAA
#> 3        150 GGGGGGGG
#> 4        400 CCCCCCCC
```

```
  BC_data_2 <- asBCdat(dat = test_data_frame,
                       label = "test_label_2",
                       BC_backbone = "CCCNNAAANNTTTNNGGGNN",
                       resDir = getwd())

  test <- genBaRcode:::com_pair(BC_dat1 = BC_data_1, BC_dat2 = BC_data_2)
```

```
  show(test)
#> $shared
#>    barcode read_count_1 read_count_2 read_count_diff
#> 1 CCCCCCCC          400          400               0
#> 2 TTTTTTTT          300          300               0
#> 3 GGGGGGGG          200          150              50
#>
#> $unique_sample1
#>   read_count  barcode
#> 4        100 AAAAAAAA
#>
#> $unique_sample2
#>   read_count  barcode
#> 4         99 AATTTAAA
```