MiNNN Documentation

Release 10.1

Aleksandr Popov

Apr 01, 2020

Contents

Contents:

CHAPTER 1

Description

Minnn is a toolset to process genetic data from sequencing machines and assemble sequenced molecules from raw FASTQ data. Consensus assembly in minnn consists of the following stages:

1. Extract barcodes from raw sequences.
2. Sort sequences by barcode values to group them for further correction.
3. Correct mismatches and indels in barcodes.
4. Sort sequences by barcode values to group them for further consensus assembly.
5. Assembly consensuses for each barcode. There can be one or many consensuses for each barcode, depending on the way of obtaining original data.
6. Export calculated consensuses to FASTQ format.

Also minnn has some other functions:

- Filter original data by barcode values.
- Filter calculated consensuses by quantity of reads from which they were assembled.
- Filter barcode values by their count.
- Decontaminate: remove barcodes from one cell that appear in samples from another cell.
- Split (demultiplex) data into separate files by barcode values.
- Collect statistics from data by barcode values and by barcode positions in sequences.

Minnn is free for academic and non-profit use (see *License*).

CHAPTER 2

Usage Chart

CHAPTER 3

Installation

3.1 Manual Installation

1. To use minnn, you need to install java version 8 or higher. For Mac OS X you can download it here: https://java.com/en/download/

   On Linux install it from repository of your distro. You can check that java is installed by typing in command line:

   ```
   java -version
   ```

   If everything is correct, it should display version 1.8 or higher.

2. Download latest minnn release from https://github.com/milaboratory/minnn/releases/

3. Extract zip archive. Copy minnn and minnn.jar into ~/bin directory, or you can use any other directory where you store executables. This directory must be in your PATH environment variable.

4. To check that minnn is installed, type:

   ```
   minnn help
   ```

   If everything is correct, minnn help will be displayed.

3.2 Installation with Homebrew

#. If you don't have homebrew, install it from https://brew.sh/ (on Mac OS X) or from https://docs.brew.sh/Homebrew-on-Linux (on Linux). #. Type minnn installation command:

```
brew install milaboratory/all/minnn
```

1. To check that minnn is installed, type:

```
minnn help
```

If everything is correct, minnn help will be displayed.

CHAPTER 4

Quick Start

Example: we have a pair of FASTQ files `R1.fastq` and `R2.fastq` from experiment where we attached 2 sample barcodes to sequence. We know that first sample barcode is first 5 nucleotides of sequence and 2nd barcode is `ATGNNNN`. We want to calculate single consensus for each combination of barcodes, and before this we want to filter out sequences where first sample barcode is `TTTTT` for which we know that this is garbage. Then we do the following actions:

1. Extract barcodes from data.

   ```
   minnn extract --input R1.fastq R2.fastq --output extracted.mif --pattern "^(SB1:N
   ↪{5}) & (SB2:ATGNNNN)\*"
   ```

   Note that extract action will check all reads in order specified in `--input` argument. It differs from behavior of old versions of MiNNN that also tried swapped `R1` and `R2` by default. Details about command line arguments and syntax can be found in *extract* and *Pattern Syntax* sections.

2. Sort reads by barcode values.

   ```
   minnn sort --input extracted.mif --output extracted_sorted.mif --groups SB1 SB2
   ```

3. Correct mismatches and indels in barcodes.

   ```
   minnn correct --input extracted_sorted.mif --output corrected.mif --groups SB1 SB2
   ```

4. Filter out garbage reads.

   ```
   minnn filter --input corrected.mif --output filtered.mif "SB1~'~TTTTT'"
   ```

   Details about command line arguments and syntax can be found in *filter* and *Filter Syntax* sections.

5. (Optionally) check statistics for collected barcodes.

   ```
   minnn stat-groups --input filtered.mif --output stat-groups.txt --groups SB1 SB2
   minnn stat-positions --input filtered.mif --output stat-positions.txt --groups SB2
   ```

6. Sort reads by barcode values.

```
minnn sort --input filtered.mif --output filtered_sorted.mif --groups SB1 SB2
```

7. Calculate consensuses.

```
minnn consensus --input filtered_sorted.mif --output consensus.mif --max-
↪consensuses-per-cluster 1 --groups SB1 SB2
```

8. Export consensuses to FASTQ files.

```
minnn mif2fastq --input consensus.mif --group R1=consensus-R1.fastq --group␣
↪R2=consensus-R2.fastq
```

CHAPTER 5

Routines

5.1 Barcode extraction

Barcode extraction can be performed with *extract* action. Typical case is when we have a pair of FASTQ files with `R1` and `R2` reads that contain barcodes. Main task here is to create pattern query for `extract` action, and barcodes will be extracted from sequences by this pattern. Patterns are similar to regular expressions, but with some features specific for nucleotide sequences. Detailed description of pattern syntax is in *Pattern Syntax* section. There are examples of patterns for some simple cases. In these examples we extract barcodes from `data-R1.fastq` and `data-R2.fastq` files and write results to `barcodes-R1.fastq` and `barcodes-R2.fastq` files. `extract` action writes output data in MIF format, so we use *mif2fastq* action to convert it to FASTQ format. Extracted barcodes will be in read description lines of output FASTQ files.

Example 1. Barcode is first 8 nucleotides of `R1`:

```
minnn extract --pattern "^(barcode:N{8})\*" --input data-R1.fastq data-R2.fastq --
↪output extracted.mif
minnn mif2fastq --input extracted.mif --group R1=barcodes-R1.fastq --group␣
↪R2=barcodes-R2.fastq
```

Example 2. There are 2 barcodes, first starting with `ATT` and ending with `AAA`, with length 9, and second starting with `GCC` and ending with `TTT`, with length 12. Swapping of `R1` and `R2` is not allowed, first barcode is always in `R1` and second in `R2`:

```
minnn extract --pattern "(B1:ATTNNNAAA)\(B2:GCCN{6}TTT)" --input data-R1.fastq data-
↪R2.fastq --output extracted.mif
minnn mif2fastq --input extracted.mif --group R1=barcodes-R1.fastq --group␣
↪R2=barcodes-R2.fastq
```

Example 3. Good sequence starts with `ATTAGACA`, and first 5 nucleotides can be possibly cut; and if sequence starts with something else, we want to skip it. First barcode with length 5 is immediately after `ATTAGACA`, then there must be `GGC` and any 5 nucleotides, and then the second barcode starting with `TTT` with length 12. Also, good sequence must end with `TTAGC`, and last 2 nucleotides can be possibly cut. `R1` and `R2` can be in reverse order in some reads. And we want to allow substitutions and indels (but with score penalties) inside sequences:

```
minnn extract --pattern "^<{5}attagaca(B1:n{5})gccn{5}(B2:tttn{9})+ttagc>>$\*" --try-
↪reverse-order --score-threshold -25 --input data-R1.fastq data-R2.fastq --output␣
↪extracted.mif
minnn mif2fastq --input extracted.mif --group R1=barcodes-R1.fastq --group␣
↪R2=barcodes-R2.fastq
```

5.2 Demultiplexing

Demultiplexing is splitting one dataset into multiple datasets by barcode values. Demultiplexing can be performed with *demultiplex* action. It works with MIF files, so if you want to demultiplex data from FASTQ files, you need to extract barcodes and convert data to MIF format first, see *Barcode extraction* section. Output MIF files can be converted to FASTQ with *mif2fastq* action. There are 2 common demultiplexing tasks: split file by barcode values and extract samples with specified combinations of barcode values.

Example 1. Split data by unique UMI values. We have input data where UMI is first 6 nucleotides, and we want to perform barcodes correction (see *Correcting UMI sequence* section) before demultiplexing.

```
minnn extract --pattern "^(UMI:N{6})\*" --input data-R1.fastq data-R2.fastq --output␣
↪extracted.mif
minnn sort --groups UMI --input extracted.mif --output sorted.mif
minnn correct --groups UMI --input sorted.mif --output corrected.mif
minnn demultiplex --by-barcode UMI corrected.mif --demultiplex-log demultiplex.log
```

Note that splitting data by unique UMI values can result in very big number of output files!

Example 2. Input data is like in previous example, but we will extract only data with the following UMI values: `AATTTT`, `AAAGGG`, `CCCCCC`, `AGACAT`, `TTTTTA`, `TTTTTG`. For this task we will create the following sample file `umi_samples.txt`:

```
Sample UMI
value_AATTTT AATTTT
value_AAAGGG AAAGGG
value_CCCCCC CCCCCC
value_AGACAT AGACAT
value_TTTTTA TTTTTA
value_TTTTTG TTTTTG
```

And then issue the following commands:

```
minnn extract --pattern "^(UMI:N{6})\*" --input data-R1.fastq data-R2.fastq --output␣
↪extracted.mif
minnn sort --groups UMI --input extracted.mif --output sorted.mif
minnn correct --groups UMI --input sorted.mif --output corrected.mif
minnn demultiplex --by-sample umi_samples.txt corrected.mif --demultiplex-log␣
↪demultiplex.log
```

Example 3. We extracted sequence barcodes with *extract* action into `extracted.mif` file, and we named these barcodes `SB1` and `SB2`. Now we want to put sequences with specified combinations of `SB1` and `SB2` into separate MIF files. There we will use sample file `samples.txt` with multiple barcodes:

```
Sample SB1 SB2
sample1 ATTAGACA CCCCCC
sample2 ATTAGACA GGGGGG
sample3 ATTACCCC TTTTTT
```

And then issue the following command:

```
minnn demultiplex --by-sample samples.txt extracted.mif --demultiplex-log demultiplex.
↪log
```

5.3 Correcting UMI sequence

UMI sequences in input data often contain substitutions and indels, and we want to correct such errors to cluster sequences by UMI without creating extra clusters for variants with errors. Barcodes correction is performed with *correct* action. It is performed after barcode extraction, see *Barcode extraction* section. **Important:** file must be sorted with *sort* action before using `correct` action, and `--groups` argument in `sort` action must contain the same groups in the same order as in `correct` action. In common cases you can use the default settings for `sort` and `correct` actions and specify only input and output files and list of barcode names in `--groups` argument:

```
minnn sort --groups UMI --input extracted.mif --output sorted.mif
minnn correct --groups UMI --input sorted.mif --output corrected.mif
```

You can convert output MIF file into FASTQ with *mif2fastq* action, or watch statistics for barcode values and positions with *stat-groups* and *stat-positions* actions. If you want to specify custom settings for barcode correction, see the description of available options on *correct* action page.

Example. We want to extract and correct UMI in pair of FASTQ files that contain `R1` and `R2`. We know that UMI is first 6 nucleotides of the read, and it starts with `ATT`. Then we use the following commands:

```
minnn extract --pattern "^(UMI:ATTNNN)\*" --input R1.fastq R2.fastq --output␣
↪extracted.mif
minnn sort --groups UMI --input extracted.mif --output sorted-UMI.mif
minnn correct --groups UMI --input sorted-UMI.mif --output corrected-UMI.mif
minnn mif2fastq --input corrected-UMI.mif --group R1=corrected-UMI-R1.fastq --group␣
↪R2=corrected-UMI-R2.fastq
```

5.4 Consensus assembly

Consensus assembly consists of 6 stages:

1. Extract barcodes from raw sequences.
2. Sort sequences by barcode values to group them for further correction.
3. Correct mismatches and indels in barcodes.
4. Sort sequences by barcode values to group them for further consensus assembly.
5. Assembly consensuses for each barcode. There can be one or many consensuses for each barcode, depending on the way of obtaining original data.
6. Export calculated consensuses to FASTQ format.

Example. We have 2 FASTQ files with `R1` and `R2`. We want to assemble consensuses by UMI that is 8 nucleotides after first 3 nucleotides `TTT`. And we know that there must be only 1 consensus for each UMI. Then we use the following commands:

```
minnn extract --pattern "^TTT(UMI:N{8})\*" --input R1.fastq R2.fastq --output␣
↪extracted.mif
minnn sort --groups UMI --input extracted.mif --output sorted-1.mif
```

(continues on next page)

(continued from previous page)

```
minnn correct --groups UMI --input sorted-1.mif --output corrected.mif
minnn sort --groups UMI --input corrected.mif --output sorted-2.mif
minnn consensus --groups UMI --max-consensuses-per-cluster 1 --input sorted-2.mif --
 ↪output consensus.mif
minnn mif2fastq --input consensus.mif --group R1=consensus-R1.fastq --group␣
 ↪R2=consensus-R2.fastq
```

To configure settings for consensus assembly, see the description of available options on *consensus* action page.

CHAPTER 6

Use Cases

6.1 T/B cell repertoire analysis with UMI

6.2 Variant calling with UMI

6.3 RNA sequencing

6.4 Single-cell sequencing

CHAPTER 7

Reference

7.1 Command Line Syntax

Note: command line arguments that use quality expect phred quality score in decimal format. Minimal available quality is 0, maximal is 58. 34 and higher values are considered good quality by default.

7.1.1 extract

Extract action is used to process reads from FASTQ or MIF files and extract information about barcodes. It works with *patterns*: query strings that specify which information we want to extract. Extract action patterns are similar to regular expressions, but with specific elements for nucleotide sequence processing and barcode extraction tasks. Details about pattern syntax can be found in *Pattern Syntax* section.

`--pattern` is the required argument for extract action; pattern query must be specified after it in double quotes `""`. `--input` and `--output` arguments are optional, but in most cases they must be specified. Missing `--input` argument means that input data will come from stdin, which is useful when working with pipes. Note that only single-read FASTQ file can be passed from stdin. Also, MIF file (single-read or multi-read) can be passed from stdin, but note that `--input-format MIF` argument must always be present when using extract action with input from MIF. If `--output` argument is missing, data will be written to stdout. Examples:

```
minnn extract --input R1.fastq R2.fastq --output extracted.mif --pattern
↪"(SB1:NNN)atta \ (SB2:NNN)gaca"
xzcat data.mif.xz | minnn extract --input-format MIF --pattern "ATCC\*" | xz >␣
↪extracted.mif.xz
minnn extract --input test.mif --input-format MIF --pattern "(UMI:^N{:8})" --output␣
↪test_umi.mif
```

Important: number of reads in the specified pattern must be equal to number of reads in the input data. For example, if there are 2 FASTQ files in the input, there must be a read separator (`\`) in the pattern with queries for `R1` and `R2`.

By default, if there is more than 1 read in the input, extract action will check input reads in order they specified in `--input` argument, or if input file is MIF, then in order they saved in MIF file. If `--try-reverse-order` argument is specified, it will also try the combination with 2 swapped last reads (for example, if there are 3 reads, it will try `R1, R2, R3` and `R1, R3, R2` combinations), and then choose the match with better score. This will be done for each multi-read sequence from the input.

Extract action uses bitap algorithm to quickly search nucleotide sequences from pattern in the target, and then it uses aligner to align pattern sequence with found section of the target and calculate match score. You can set maximum number of errors for bitap matcher with `--bitap-max-errors` argument. Arguments `--match-score`, `--mismatch-score`, `--uppercase-mismatch-score` and `--gap-score` set scoring parameters for the aligner. You can read details about the difference between uppercase and lowercase letters in *Pattern Syntax* section.

There are other arguments that affect match scoring: `--good-quality-value`, `--bad-quality-value` and `--max-quality-penalty` set penalties for bad quality nucleotides in the target. Good and better quality letter has no penalty, bad and worse quality letter has maximum specified penalty. Also, there are `--single-overlap-penalty` and `--max-overlap` arguments, you can read about them in *Pattern Syntax* section.

`--score-threshold` argument is used to set the score threshold: matches with lower score will not go to the output. If both `--score-threshold` and `--match-score` are `0`, only perfect matches without penalties will go to the output.

Extract action uses internal heuristics to speed up the search and reduce the number of combinations to check when using pattern sequences and logical operators. This can produce wrong results in rare cases. There is the option `--fair-sorting` to always do full search, but this is much slower than default unfair sorting.

`--threads` option sets the number of threads for pattern matching. By default, it is equal to available number of CPU cores.

`--not-matched-output` argument allows to write not matched reads to the separate MIF file. By default (if this argument is not present) not matched reads will not be written anywhere.

Sometimes read description contain barcodes or other nucleotide information. Extract action allows to parse that information from description and save it as groups in the output. Syntax for description groups parsing: `--description-group GROUPNAME='regular_expression'`. `GROUPNAME` is a group name where nucleotide sequence will be saved. It must **not** duplicate any group name from the pattern or built-in groups `R1`, `R2` etc. Multiple `description-group` arguments can be specified. `regular_expression` is common java regular expression (**not** the pattern syntax), and it must always be in single quotes `''`. Regular expressions will be applied to read descriptions of all reads (first `R1`, then `R2`, `R3` etc), and then there will be attempt to parse match as nucleotide sequence. If valid nucleotide sequence will not be found by this regexp in any read description, extract action will stop with error. Usage examples for `--description-group` arguments:

```
minnn extract --input R1.fastq R2.fastq --output extracted.mif --pattern "*\*" --
↪description-group G1='ATG.{10}'
minnn extract --pattern "(G1:ATTN{10})" --description-group G2='^.{12}' --description-
↪group G3='(?<=\=)ATTA.*(?=\;)'
```

`--description-group` arguments also support parsing sequences with qualities. If you specify 2 named groups `seq` and `qual` in regexp query, contents of these matched groups will be parsed as sequence and quality for this description group. Usage examples:

```
minnn extract --pattern "*" --description-group G1='UMI~(?<seq>.*?)~(?<qual>.*?)\{'
minnn extract --pattern "(G1:NNTTA)" --description-group G2='^(?<qual>.*?)~{5}(?<seq>
↪[a-zA-Z]*)'
```

Command line arguments reference for extract action:

```
--pattern: Query, pattern specified in MiNNN format.
--input: Input files. Single file means that there is 1 read or multi-read file;␣
↪multiple files mean that there is 1 file for each read. If not specified, stdin␣
↪will be used.
```

(continues on next page)

(continued from previous page)

```
--output: Output file in MIF format. If not specified, stdout will be used.
--not-matched-output: Output file for not matched reads in MIF format. If not␣
↪specified, not matched reads will not be written anywhere.
--input-format: Input data format. Available options: FASTQ, MIF.
--try-reverse-order: If there are 2 or more reads, check 2 last reads in direct and␣
↪reverse order.
--match-score: Score for perfectly matched nucleotide.
--mismatch-score: Score for mismatched nucleotide.
--uppercase-mismatch-score: Score for mismatched uppercase nucleotide.
--gap-score: Score for gap or insertion.
--score-threshold: Score threshold, matches with score lower than this will not go to␣
↪output.
--good-quality-value: This or better quality value will be considered good quality,␣
↪without score penalties.
--bad-quality-value: This or worse quality value will be considered bad quality, with␣
↪maximal score penalty.
--max-quality-penalty: Maximal score penalty for bad quality nucleotide in target.
--single-overlap-penalty: Score penalty for 1 nucleotide overlap between neighbor␣
↪patterns. Negative value or 0.
--max-overlap: Max allowed overlap for 2 intersecting operands in +, & and pattern␣
↪sequences. Value -1 means unlimited overlap size.
--bitap-max-errors: Maximum allowed number of errors for bitap matcher.
--fair-sorting: Use fair sorting and fair best match by score for all patterns.
-n, --number-of-reads: Number of reads to take; 0 value means to take the entire␣
↪input file.
--threads: Number of threads for parsing reads.
--report: File to write report in human readable form. If not specified, report is␣
↪displayed on screen only.
--json-report: File to write command execution stats in JSON format.
--description-group: Description group names and regular expressions to parse␣
↪expected nucleotide sequences for that groups from read description. Example: --
↪description-group CID1='ATTA.{2-5}GACA' --description-group CID2='.{11}$'
--overwrite-if-required: Overwrite output file if it is corrupted or if it was␣
↪generated from different input file or with different parameters. -f / --force-
↪overwrite overrides this option.
-f, --force-overwrite: Force overwrite of output file(s).
```

7.1.2 filter

Filter action is used to filter data from MIF file with specified query and write only matching reads to the output. There are 3 general cases of using filter action:

1. Filter input or corrected data by barcodes whitelist.
2. Filtering input data by group value or length.
3. Filtering output of *consensus* or *consensus-dma* action to exclude consensuses assembled from too small number of reads.

Filter action works with queries. Query must be enclosed in double quotes `""`. Details about filter syntax can be found in *Filter Syntax* section. Filter action must contain a query or at least one `--whitelist` or `--whitelist-patterns` argument.

`--whitelist` argument is used to set whitelist of barcode values from a text file. Whitelist file must contain exact barcode values, one value on the line. **Important:** `--whitelist` filtering doesn't support matching by wildcards. Entries with wildcards in whitelist file will be transformed to all corresponding combinations of entries with basic letters. Also, whitelist entries with wildcards allow exact matches: for example, for entry `NNGT` value `NNGT` in the input will match, but value `NAGT` will be filtered out. If you need full wildcards support, use `--whitelist-patterns` argument.

`--whitelist-patterns` argument is much slower than `--whitelist`, but it allows to use many advanced features in whitelist entries, and has full wildcards support. Whitelist file for `--whitelist-patterns` argument must contain queries with MiNNN *Pattern Syntax*, one query on the line. Exact barcode values are also possible with this syntax: to use exact barcode value in the whitelist, start it with `^` (it means that value must start from the beginning of the capture group) and end it with `$` (end is exactly at the end of the capture group); for example, `^TTTGGCAGC$`.

There can be multiple `--whitelist` and `--whitelist-patterns` options, it is useful for specifying whitelists for different barcodes. Also, whitelists and filter query can be used simultaneously. In this case, only reads that match both the query and the whitelists will pass to the output.

Examples:

```
minnn filter --input data.mif --output filtered.mif "MinGroupQuality(G1) = 7" --
↪whitelist-patterns G1=whitelist.txt
minnn filter --whitelist UMI=whitelist_umi.txt --whitelist SB=whitelist_sb.txt --
↪input in.mif --output out.mif
```

Example contents of whitelist file for `--whitelist` argument:

```
GGTCCTTCAGC
GGTAATTCAGC
GGTCCGTCAGC
GGTCCTTCTTT
GGTCCTTCTTAA
GGTCCTTCTGCA
```

Example contents of whitelist file for `--whitelist-patterns` argument:

```
^AAAAAATTCTTNNTTCT$
AAGGGGTTTCTCTGT$
^<<<AAAATCTGGGCCTGTGCT$
^AAAA + GGNAACT
AAAATTTT & TTTGGGCCT
```

`--input` and `--output` arguments are optional, and if they are missing, stdin and stdout will be used instead of input and output files. Filter action always uses MIF format for input and output. Usage examples for filter action can be found in *Filter Syntax* section.

If `--whitelist-patterns` argument or pattern filter is used in the filter query, then pattern matching algorithm will be used. This is the same algorithm as in *extract* action. `--fair-sorting` argument is option for this algorithm, you can read the details about it in help for *extract* action. If there are no `--whitelist-patterns` arguments and no pattern filters in the filter query, `--fair-sorting` argument has no effect.

`--threads` option sets the number of threads for filter query matching. By default, it is equal to available number of CPU cores.

Command line arguments reference for filter action:

```
--input: Input file in MIF format. If not specified, stdin will be used.
--output: Output file in MIF format. If not specified, stdout will be used.
--whitelist: Barcode Whitelist Options: Barcode names and names of corresponding␣
↪files with whitelists. Whitelist files must contain barcode values, one value on␣
↪the line. For example, --whitelist BC1=options_BC1.txt can be used, where options_
↪BC1.txt contains AAA, GGG and CCC lines: they are whitelist options for barcode BC1.
--whitelist-patterns: Barcode Whitelist Pattern Options: Barcode names and names of␣
↪corresponding files with whitelists. Whitelist files must contain barcode values or␣
↪queries with MiNNN pattern syntax, one value or query on the line. This is more␣
↪convenient way for specifying OR operator when there are many operands. So, for␣
↪example, instead of using "BC1~'^AAA' | BC1~'^GGG' | BC1~'^CCC$'" query, option --
↪whitelist-patterns BC2=options_BC2.txt can be used, where options_BC2.txt must␣
↪contain ^AAA, ^GGG and ^CCC$ lines. If multiple --whitelist and --whitelist-
↪patterns options specified for the same barcode, then barcode is considered␣
↪matching if at least 1 whitelist contains it.
```

(continues on next page)

(continued from previous page)

```
--fair-sorting: Use fair sorting and fair best match by score for all patterns.
-n, --number-of-reads: Number of reads to take; 0 value means to take the entire␣
↪input file.
--threads: Number of threads for parsing reads.
--report: File to write report in human readable form. If not specified, report is␣
↪displayed on screen only.
--json-report: File to write command execution stats in JSON format.
--overwrite-if-required: Overwrite output file if it is corrupted or if it was␣
↪generated from different input file or with different parameters. -f / --force-
↪overwrite overrides this option.
-f, --force-overwrite: Force overwrite of output file(s).
```

7.1.3 demultiplex

Demultiplex action is used to filter nucleotide sequences from one MIF file into multiple MIF files, separating sequences by barcode values or by samples.

`filter_options` argument is mandatory. It must contain one or multiple filter options and input file name. Stdin is not supported in demultiplex action. Available filter options are:

`--by-barcode GROUPNAME` - split by values (nucleotide sequences) of group `GROUPNAME`, one output file for each value

`--by-sample SAMPLE_FILENAME` - extract samples specified in file `SAMPLE_FILENAME` (file format is described below), one output file for each sample; and reads that didn't match any sample will be skipped

You can specify multiple `--by-barcode` and `--by-sample` options, but note that output file will be created for each combination of barcode values and samples, so this command can create very big number of files!

`--demultiplex-log` argument is mandatory. It specifies the name of output text file where the list of all output files will be written. You can use the log file from previous run and add `--overwrite-if-required` argument if you want to use smart overwrite feature: overwrite output files only if the input file had changed.

`--output-path` argument allows to set the path for output files. If not specified, output files will be written to the same directory as input file. This option does not affect demultiplex log file; you can specify the path for demultiplex log file in `--demultiplex-log` argument.

Examples for demultiplex action:

```
minnn demultiplex --output-buffer-size 30000 --by-barcode SB1 corrected.mif --
↪demultiplex-log log.txt
minnn demultiplex --demultiplex-log 1.log --by-sample samples1.txt --by-sample␣
↪samples2.txt --by-barcode UMI in.mif
minnn demultiplex --by-barcode S1 data.mif --output-path ./data --demultiplex-log ./
↪logs/demultiplex.log
```

Sample file format:

Sample file is a plain-text table with values separated with spaces or tabs. First line contains the keyword `Sample` and then group names. Other lines start with sample names and then there are values of the groups for this sample. Multiple lines with the same sample name will be combined into one sample.

Example for sample file with single group:

```
Sample UMI
good_sample_1 AAAA
good_sample_1 TTTT
```

(continues on next page)

(continued from previous page)

```
good_sample_2 CCCC
good_sample_2 AAGG
error_value_1 GGAA
error_value_2 AATT
special_value TTAA
```

Asterisk wildcards `*` can be used if any value of a group must match. Example for sample file with multiple groups and wildcards:

```
Sample SB1 SB2 SB3
test_sample_1_1 AAAA CTT CGCGTCT
test_sample_1_2 * * CGCGGGG
test_sample_1_3 AACA * CGCGTCT
test_sample_1_4 AAAA GTT CGCGTCT
```

Output file names will contain input file name, and then sequence of sample names and barcode values separated with _ token, in the same order as these sample files and barcode group names were specified in the `filter_options` argument. For example, output file names can look like `data_GGT_test_sample_1_4_TTGG.mif`, `corrected_ATTAGACA.mif` or `extracted_sample4.mif`. One file will be created for each combination of barcode values and matched samples that contains at least one read.

`--output-buffer-size` argument allows to set the write buffer size (in bytes) manually. This buffer size is used for each output file, so it's better to set lower values if you intend to create a lot of output files.

Command line arguments reference for demultiplex action:

```
Filter Options: Barcodes and sample configuration files that specify sequences for␣
↪demultiplexing. At least 1 barcode or 1 sample file must be specified. Syntax␣
↪example: minnn demultiplex --by-barcode UID --by-sample samples.txt input.mif
--demultiplex-log: Demultiplex log file name, to record names of generated files.
--output-path: Path to write output files. If not specified, output files will be␣
↪written to the same directory as input file. This option does not affect␣
↪demultiplex log file; you can specify the path for demultiplex log file in --
↪demultiplex-log argument.
--output-buffer-size: Write buffer size for each output file.
-n, --number-of-reads: Number of reads to take; 0 value means to take the entire␣
↪input file.
--report: File to write report in human readable form. If not specified, report is␣
↪displayed on screen only.
--json-report: File to write command execution stats in JSON format.
--overwrite-if-required: Overwrite output file if it is corrupted or if it was␣
↪generated from different input file or with different parameters. -f / --force-
↪overwrite overrides this option.
-f, --force-overwrite: Force overwrite of output file(s).
```

7.1.4 mif2fastq

Mif2fastq action is used to convert MIF file to FASTQ format. It writes information about capture groups to read headers, and optionally allows to add original headers (that were in FASTQ files passed to *extract* action) to the start of output header comments. Also mif2fastq action allows to save capture groups contents as separate reads.

Group options are mandatory, and they specify what output files will be created. Group names may be built-in groups `R1`, `R2`, `R3` etc that contain the entire reads, also group names may be names of capture groups created by *extract* action, and also there may be built-in groups `CR1`, `CR2`, `CR3` etc if the MIF file was created by *consensus* or *consensus-dma* action with `--consensuses-to-separate-groups` parameter. Group options format is `--group GROUPNAME1=filename1.fastq --group GROUPNAME2=filename2.fastq`, and there can be any number of pairs of groups and corresponding file names, but at least 1 pair must be specified.

`--input` argument is optional, and if it's missing, stdin will be used instead of input file.

Examples for mif2fastq action:

```
minnn mif2fastq --input corrected.mif --copy-original-headers --group R1=R1.fastq --
↪group R2=R2.fastq
xzcat data.mif.xz | minnn mif2fastq --group R1=data-R1.fastq --group R2=data-R2.fastq␣
↪--group UMI=data-UMI.fastq
minnn mif2fastq --input consensus.mif --group R1=consensus-R1.fastq --group␣
↪R2=consensus-R2.fastq
minnn mif2fastq --input consensus-separate.mif --group R1=data-R1.fastq --group␣
↪CR1=consensus-R1.fastq
```

`--copy-original-headers` parameter specifies to copy original headers (that were in FASTQ files passed to *extract* action) to the start of output header comments. If it isn't specified, only minnn comments will be in output FASTQ files.

Output FASTQ files comments format:

`@[original_headers]~group_descriptions~[||~]`

`~` symbol is used as separator between sections, `|` symbol is used as separator between groups inside `group_descriptions` section. `original_headers` section is present only if `--copy-original-headers` parameter is specified. `||~` token is present only if this was a reversed match (with swapped `R1` and `R2`) in *extract* action. In reversed matches `R1` read in extract action input file becomes `R2` in mif2fastq output file and vice versa, so `||~` token is used as notification for reversed matches.

`group_descriptions` section contains descriptions of all capture groups except built-in groups `R1`, `R2`, `R3` etc. Groups are separated by `|` token. There can be 3 types of groups in this section:

1. Group that is inside *current target*. Format: `GROUPNAME~SEQ~QUAL~{FROM~TO}`. Example: `G1~ATTAGGG~111BFF1~{10~17}`. `GROUPNAME` is capture group name, `SEQ` is target sequence where this group matched, `QUAL` is quality of this target fragment, `FROM` is start coordinate in the current target (inclusive), `TO` is end coordinate in the current target (exclusive). *Current target* is read corresponding to the current output FASTQ file. It can be built-in group `R1`, `R2` etc that represents the entire input read; it can be overridden `R1`, `R2` etc if there was override for these built-in groups in *extract* action query; and it can be capture group used as output read in group options of mif2fastq action.
2. Group that is matched but not inside current target. Format: `GROUPNAME~SEQ~QUAL`. Example: `UMI~AAAGGCCC~\\\111C`. This format is used for groups that matched somewhere in another read or not in bounds of current target.
3. Not matched group. Format: `GROUPNAME`. Example: `SB1`. Used for not matched groups. *Pattern Syntax* page contains the information where such groups can appear.

Command line arguments reference for mif2fastq action:

```
--group: Group Options: Groups and their file names for output reads. At least 1␣
↪group must be specified. Built-in groups R1, R2, R3... used for input reads.␣
↪Example: --group R1=out_R1.fastq --group R2=out_R2.fastq --group UMI=UMI.fastq
--input: Input file in MIF format. If not specified, stdin will be used.
--copy-original-headers: Copy original comments from initial fastq files to comments␣
↪of output fastq files.
-n, --number-of-reads: Number of reads to take; 0 value means to take the entire␣
↪input file.
--report: File to write report in human readable form. If not specified, report is␣
↪displayed on screen only.
```

(continues on next page)

(continued from previous page)

```
--json-report: File to write command execution stats in JSON format.
-f, --force-overwrite: Force overwrite of output file(s).
```

7.1.5 correct

Correct action is used to correct errors in barcodes. It collects sequences from a capture group to the barcode tree, and cluster barcodes by similarity and frequency, creating new cluster for each frequent barcode and for each barcode that doesn't have matching cluster by similarity. Then it replaces rare barcodes with the most frequent ones from the same cluster and writes reads with corrected barcodes to the output.

Correct action works in 4 stages (if `--primary-groups` argument is not present):

1. Reading input file and collecting all barcodes from there.
2. Clustering barcodes by wildcards to find barcodes with wildcards that can be merged with more specific barcodes.
3. Clustering barcodes to find possible mismatches and indels; creating correction table.
4. Reading input file again, correcting barcodes and writing output file.

Important: file must be sorted with *sort* action before using correct action, and `--groups` argument in sort action must contain the same groups in the same order as in correct action.

`--input` argument is mandatory, reading data from stdin is not supported because correct action reads input file twice, on stages 1 and 3. `--output` argument is optional: results will be written to stdout if `--output` argument is missing. `--groups` argument is mandatory, it must contain space separated list of groups that will be corrected. Built-in groups `R1`, `R2`, `R3` etc are not supported in correct action.

Examples for correct action:

```
minnn correct --groups UMI --input sorted.mif --output corrected.mif --cluster-
↪threshold 0.01
minnn correct --groups G1 G3 G2 --max-unique-barcodes 7000 --input filtered.mif | xz >
↪ corrected.mif.xz
minnn correct --groups SB1 SB2 --max-errors 2 --max-errors-share -1 --input data.mif -
↪-output corrected.mif
```

`--max-unique-barcodes` and `--min-count` arguments can be used to filter barcode values by their count *after* correction. They work in the same way as in *filter-by-count* action.

`--max-errors-share` argument specifies how two barcodes can differ in the same cluster. This share is multiplied on average barcode length to calculate maximal allowed number of errors (Levenshtein distance) between barcodes; but if result is less than 1, it rounds up to 1. Barcodes with bigger number of errors will not be corrected. The maximal allowed number of errors is calculated separately for each group, for example, if there is short group `CB` and long group `UMI`, more errors will be allowed in `UMI` group. Negative value after `--max-errors-share` means that `--max-errors-share` argument is disabled and you must set the `--max-errors` argument.

You can specify the maximal allowed number of errors directly, same value for all groups. `--max-errors` argument can be used for this. It is disabled (set to `-1`) by default.

Important: If both `--max-errors-share` and `max-errors` arguments are enabled, then the lowest value of max errors from these arguments will be used. If you want, for example, to use only `--max-errors`, then disable (set to `-1`) the `--max-errors-share` argument.

Clustering algorithm uses probabilities of substitutions and indels in sequence to check when barcode cannot be added to cluster; for example, if the barcode's count is big, and there is low probability that this barcode emerged because of errors. You can change the default values for these probabilities with `--single-substitution-probability` and `--single-indel-probability` arguments. If you don't need this feature, set both probabilities to `1`.

In addition to `--single-substitution-probability` and `--single-indel-probability` arguments, clustering algorithm also allows to directly specify the frequency threshold that prevents two unique but similar barcodes from merging into one barcode. `--cluster-threshold` argument can be used for it. If the current barcode's count divided to cluster's largest barcode's count is below this threshold, the current barcode can be merged to the cluster, otherwise it will form a new cluster. This feature is turned off (set to `1`) by default.

Barcode clustering algorithm can use multiple layers: there is cluster head (the most frequent barcode in the cluster), then the layer contains barcodes clustered to the head, and there can be more layers of barcodes clustered to the previous layer. Maximum number of layers is specified by `--max-cluster-depth` argument. If `--max-cluster-depth` is `1` then there will be only 1 layer below the head; if `--max-cluster-depth` is `2`, there will be second layer clustered to the first layer etc.

If there are wildcards in the barcodes, then barcodes with wildcards can be merged with more specific barcodes *before* performing search for mismatches and indels. On this wildcards pre-processing stage, when merging cluster of barcodes with pure letter in a position and cluster of barcodes with wildcard in that position, clusters will be merged if pure letter cluster size multiplied on the threshold is greater or equal to wildcard cluster size, otherwise clusters will be treated as different barcodes. The threshold can be specified with `--wildcards-collapsing-merge-threshold` argument. If this argument is absent, default threshold value will be used.

`--primary-groups` argument means that barcodes must be corrected inside clusters that are formed from reads with the same values of the primary groups. Usage example is correcting UMI separately for each unique cell barcode. **Important:** if `--primary-groups` argument is used, then input file must be sorted by both primary and secondary groups, and primary groups must be *first* in `--groups` argument of sort action. For example, if correct action contains arguments `--primary-groups CB1 CB2` and `--groups UMI1 UMI2`, then sort action must contain argument `--groups CB1 CB2 UMI1 UMI2`. If primary groups need correction, they must be corrected before this sorting, with separate correct action.

For example, we have 2 cell barcodes in groups `CB1` and `CB2` and UMI in group `UMI`, and we want to correct cell barcodes, and correct UMI for each unique combination of cell barcodes separately. Then we can use the following sequence of commands:

```
minnn sort --groups CB1 CB2 --input data.mif --output sorted-primary.mif
minnn correct --groups CB1 CB2 --input sorted-primary.mif --output corrected-primary.
↪mif
minnn sort --groups CB1 CB2 UMI --input corrected-primary.mif --output sorted-all.mif
minnn correct --primary-groups CB1 CB2 --groups UMI --input sorted-all.mif --output␣
↪corrected-secondary.mif
```

Note that `--max-unique-barcodes` and `--min-count` are counted separately for each cluster if `--primary-groups` argument is present, so you may want to set lower values for `--max-unique-barcodes` and `--min-count` arguments if you use them.

`--threads` argument works only if `--primary-groups` argument is specified. Correction of secondary barcodes for each combination of primary barcodes can be performed in separate thread.

Command line arguments reference for correct action:

```
--groups: Group names for correction.
--primary-groups: Primary group names. If specified, all groups from --groups␣
↪argument will be treated as secondary. Barcode correction will be performed not in␣
↪scale of the entire input file, but separately in clusters with the same primary␣
↪group values. If input file is already sorted by primary groups, correction will be␣
↪faster and less memory consuming. Usage example: correct cell barcodes (CB) first,␣
↪then sort by CB, then correct UMI for each CB separately. So, for first correction␣
↪pass use "--groups CB", and for second pass use "--groups UMI --primary-groups CB".␣
↪If multiple primary groups are specified, clusters will be determined by unique␣
↪combinations of primary groups values.
```

(continues on next page)

(continued from previous page)

```
--input: Input file in MIF format. This argument is required; stdin is not supported.
--output: Output file in MIF format. If not specified, stdout will be used.
--max-errors-share: Relative maximal allowed number of errors (Levenshtein distance)␣
↪between barcodes for which they are considered identical. It is multiplied on␣
↪average barcode length to calculate maximal allowed number of errors; if result is␣
↪less than 1, it rounds up to 1. This max errors calculation method is enabled by␣
↪default. It is recommended to set only one of --max-errors-share and --max-errors␣
↪parameters, and set the other one to -1. Negative value means that this max errors␣
↪calculation method is disabled. If both methods are enabled, the lowest calculated␣
↪value of max errors is used.
--max-errors: Maximal Levenshtein distance between barcodes for which they are␣
↪considered identical. It is recommended to set only one of --max-errors-share and --
↪max-errors parameters, and set the other one to -1. Negative value means that this␣
↪max errors calculation method is disabled. If both methods are enabled, the lowest␣
↪calculated value of max errors is used.
--cluster-threshold: Threshold for barcode clustering: if smaller barcode count␣
↪divided to larger barcode count is below this threshold, barcode will be merged to␣
↪the cluster. This feature is turned off (set to 1) by default, because there is␣
↪already filtering by --single-substitution-probability and --single-indel-
↪probability enabled. You can turn on this filter (set the threshold) and set single␣
↪error probabilities to 1; or you can use both filters (by cluster threshold and by␣
↪single error probabilities) if you want.
--max-cluster-depth: Maximum cluster depth for algorithm of similar barcodes␣
↪clustering.
--single-substitution-probability: Single substitution probability for clustering␣
↪algorithm.
--single-indel-probability: Single insertion/deletion probability for clustering␣
↪algorithm.
--max-unique-barcodes: Maximal number of unique barcodes that will be included into␣
↪output. Reads containing barcodes with biggest counts will be included, reads with␣
↪barcodes with smaller counts will be excluded. Value 0 turns off this feature: if␣
↪this argument is 0, all barcodes will be included.
--min-count: Barcodes with count less than specified will not be included in the␣
↪output.
--excluded-barcodes-output: Output file for reads with barcodes excluded by count. If␣
↪not specified, reads with excluded barcodes will not be written anywhere.
-w, --wildcards-collapsing-merge-threshold: On wildcards collapsing stage, when␣
↪merging cluster of barcodes with pure letter in a position and cluster of barcodes␣
↪with wildcard in that position, clusters will be merged if pure letter cluster size␣
↪multiplied on this threshold is greater or equal to wildcard cluster size,␣
↪otherwise clusters will be treated as different barcodes.
-n, --number-of-reads: Number of reads to take; 0 value means to take the entire␣
↪input file.
--threads: Number of threads for barcodes correction. Multi-threading is used only␣
↪with --primary-groups argument: correction for different primary groups can be␣
↪performed in parallel.
--report: File to write report in human readable form. If not specified, report is␣
↪displayed on screen only.
--json-report: File to write command execution stats in JSON format.
--overwrite-if-required: Overwrite output file if it is corrupted or if it was␣
↪generated from different input file or with different parameters. -f / --force-
↪overwrite overrides this option.
-f, --force-overwrite: Force overwrite of output file(s).
-nw, --no-warnings: Suppress all warning messages.
```

7.1.6 filter-by-count

Filter by count action is used to filter barcode values by their count. It works in 2 stages:

1. Reading input file and counting barcodes.
2. Reading input file again, and writing only matching barcodes (with count above the threshold) to the output file.

`--input` argument is mandatory, reading data from stdin is not supported because filter-by-count action reads input file twice. `--output` argument is optional: results will be written to stdout if `--output` argument is missing. `--groups` argument is mandatory, it must contain space separated list of groups that will be filtered by count. Built-in groups `R1`, `R2`, `R3` etc are not supported in filter-by-count action.

Examples for filter-by-count action:

```
minnn filter-by-count --groups UMI --input extracted.mif --output filtered.mif --max-
↪unique-barcodes 7000
minnn filter-by-count --groups G1 G3 G2 --min-count 100 --input data.mif | xz >␣
↪filtered.mif.xz
minnn filter-by-count --groups G1 --min-count 10 --input input.mif --output high-
↪count.mif --excluded-barcodes-output low-count.mif
```

`--max-unique-barcodes` argument is useful for filtering cell barcodes in single cell sequencing: it sets maximal count of unique barcodes that will be included in the output file. Only barcodes with highest counts will be included; barcodes with low counts will be filtered out. This limit is the same for each group and calculated for each group separately, so if you want to set different limits for different groups (for example, for cell barcodes and UMI in single cell sequencing), perform separate `filter-by-count` action runs for different groups. Reads that contain at least 1 filtered out barcode will not be included in the output. You can use `--excluded-barcodes-output` argument if you want to write filtered out reads to the separate MIF file. If `--max-unique-barcodes` argument is absent or set to `0`, filtering by maximal number of unique barcodes will be disabled.

`--min-count` argument allows to specify count threshold for barcodes directly. Barcodes with lower counts will be filtered out. Reads that contain at least 1 filtered out barcode will not be included in the output.

`--max-unique-barcodes` and `--min-count` arguments can be used simultaneously; in this case, both filters will be applied to each barcode.

Command line arguments reference for filter-by-count action:

```
--groups: Group names for filtering by count.
--input: Input file in MIF format. This argument is required; stdin is not supported.
--output: Output file in MIF format. If not specified, stdout will be used.
--max-unique-barcodes: Maximal number of unique barcodes that will be included into␣
↪output. Reads containing barcodes with biggest counts will be included, reads with␣
↪barcodes with smaller counts will be excluded. Value 0 turns off this feature: if␣
↪this argument is 0, all barcodes will be included.
--min-count: Barcodes with count less than specified will not be included in the␣
↪output.
--excluded-barcodes-output: Output file for reads with barcodes excluded by count. If␣
↪not specified, reads with excluded barcodes will not be written anywhere.
-n, --number-of-reads: Number of reads to take; 0 value means to take the entire␣
↪input file.
--report: File to write report in human readable form. If not specified, report is␣
↪displayed on screen only.
--json-report: File to write command execution stats in JSON format.
--overwrite-if-required: Overwrite output file if it is corrupted or if it was␣
↪generated from different input file or with different parameters. -f / --force-
↪overwrite overrides this option.
-f, --force-overwrite: Force overwrite of output file(s).
```

7.1.7 sort

Sort action is used to sort reads by contents (nucleotide sequences) of specified groups. Merged sorting algorithm is used for faster processing of large amounts of data. `--groups` argument is mandatory, there must be space separated list of groups, and sorting will be performed by contents of these groups. Order of groups in this list determines the priority: reads will be compared by contents of 1st group in the list, if they are equal, then by 2nd etc. The information about groups by which the sorting was performed is saved in output file, so warnings can be displayed if *consensus* or *consensus-dma* action is used with unsorted groups.

Important: sort action must be used before *consensus* or *consensus-dma* action with the same groups in `--groups` argument as in consensus action, otherwise consensus calculation will consume much more memory!

Sort action must be used after *correct* action, and not before it, because correcting barcodes will cause groups to be unsorted again. However, if correcting barcodes is not needed, sort action can be used right after *extract* or *filter* action. Also, if *correct* action is used with `--primary-groups` argument, it is recommended to make additional sorting by primary groups before correction because correction by unsorted primary groups is much slower and memory consuming.

`--input` and `--output` arguments are optional, and if they are missing, stdin and stdout will be used instead of input and output files.

Examples for sort action:

```
minnn sort --groups UMI --input corrected.mif --output sorted.mif
xzcat data.mif.xz | minnn sort --groups G1 G3 G2 | xz > sorted_data.mif.xz
```

`--chunk-size` argument sets the chunk size in bytes for merged sorting. Too large chunks can cause out of memory errors, and too small chunks can lead to poor performance. Default value `-1` means automatically calculate chunk size by input file size.

Command line arguments reference for sort action:

```
--groups: Group names to use for sorting. Priority is in descending order.
--input: Input file in MIF format. If not specified, stdin will be used.
--output: Output file in MIF format. If not specified, stdout will be used.
--chunk-size: Chunk size for sorter.
--report: File to write report in human readable form. If not specified, report is␣
↪displayed on screen only.
--json-report: File to write command execution stats in JSON format.
--overwrite-if-required: Overwrite output file if it is corrupted or if it was␣
↪generated from different input file or with different parameters. -f / --force-
↪overwrite overrides this option.
-f, --force-overwrite: Force overwrite of output file(s).
-nw, --no-warnings: Suppress all warning messages.
```

7.1.8 consensus

Consensus actions are used to calculate consensus sequences for all combinations of barcode values. They also allow to find multiple consensuses in the same combination of barcodes if there are multiple sequences with the same barcodes in the data.

Important: *sort* action must be used before any consensus action with the same groups in `--groups` argument as in consensus action, otherwise consensus calculation will consume much more memory!

Consensus action aligns multiple sequences by searching the most frequent K-mer in all of these sequences and then placing all of these sequences to the same coordinate system by offset of the most frequent K-mer in each of the sequences. Sequences where the most frequent K-mer (with allowed number of errors specified by

`--kmer-max-errors` argument) is not found are marked as remaining sequences and not included in current consensus calculation. Then consensus action calculates consensus from the current cluster of sequences that are already positioned in the same coordinate system. After this, quality trimming is performed on both sides of the consensus, and if the resulting sequence is too short, this consensus is discarded. But if there are many remaining sequences (both in case that consensus was calculated and in case it was discarded), next consensus calculation will start with the cluster of remaining sequences.

`--groups` argument with space separated list of groups must be specified in consensus action to specify which groups will be used for consensus calculation. This argument is mandatory. `--input` and `--output` arguments are optional, and if they are missing, stdin and stdout will be used instead of input and output files.

Examples for consensus action:

```
minnn consensus --groups SB1 SB2 --input sorted.mif --output consensus.mif --kmer-max-
 ↪errors 2
xzcat sorted.mif.xz | minnn consensus --output consensus.mif
minnn consensus --max-consensuses-per-cluster 1 --kmer-length 10 --input data.mif --
 ↪output consensus.mif
```

`--kmer-max-errors` argument specifies the maximal number of mismatches when the most frequent K-mer will be still considered found. `--kmer-length` argument allows to set K-mer length, and `--kmer-offset` can be used to set maximal allowed offset calculated from the middle of found K-mer to the middle of the sequence.

`--max-consensuses-per-cluster` is important option: it allows to set maximal number of consensuses for one combination of barcode values. If each molecule is marked by unique set of barcodes, you can set `--max-consensuses-per-cluster` to `1`.

If enough sequences from cluster were not used in consensus calculations, and `--max-consensuses-per-cluster` is not exceeded, then new consensus calculation will start from remaining sequences. You can set the threshold with `--skipped-fraction-to-repeat` option. If number of remaining sequences divided to cluster size is not below this threshold, new consensus calculation will start, otherwise all reads with remaining sequences will be discarded and not used in any consensus calculation. Also, if `--max-consensuses-per-cluster` is exceeded, all remaining reads will be discarded. `--not-used-reads-output` argument allows to write all discarded reads to the separate MIF file.

Calculated consensuses are processed with quality trimmer to remove low quality tails on the left and right sides. You can set trim window size (length of subsequence for which average quality is calculated) with `--trim-window-size` option. Quality threshold is set by `--avg-quality-threshold` option, and minimal allowed consensus length after trimming - by `--min-good-sequence-length` option. If resulting consensus will be shorter, it will be discarded.

Input reads are prepared with the same quality trimmer before calculating consensus, but trimming parameters are configured separately. Use `--reads-trim-window-size` to set trim window size for input reads, `--reads-avg-quality-threshold` for quality threshold and `--reads-min-good-sequence-length` for minimum input read length after trimming. Too short reads will not be included in consensus calculation.

`--consensuses-to-separate-groups` parameter changes action behavior significantly. If this parameter is not specified, output file will contain calculated consensuses. If it is specified, original sequences will be written to the output files, consensuses will be written as capture groups `CR1`, `CR2` etc, so it will be possible to cluster original reads by consensuses using *filter* / *demultiplex* actions, or export original reads and corresponding consensuses into separate reads using *mif2fastq* action. Note that input file must not contain any groups named `CR1`, `CR2` etc if you use `--consensuses-to-separate-groups` parameter.

`--original-read-stats` parameter allows to write consensus calculation stats for each original read into separate file. This is text file in space separated format, and it contains the following information for each read:

1. Original read ID: number of read, starting from 0, in the original FASTQ data that was the input of *extract* action.
2. Consensus ID (number of consensus in the output) or -1 if this read was discarded.

3. Read status. Possible values are:
   - `NOT_MATCHED` - read was not matched in *extract* action,
   - `READ_DISCARDED_TRIM` - read was discarded by length after quality trimming,
   - `KMERS_NOT_FOUND` - read was in group of reads that was fully discarded because there were no reads in the group that have the most frequent K-mer found for all targets (`R1`, `R2` etc),
   - `NOT_USED_IN_CONSENSUS` - read was discarded and not used in any consensus calculation,
   - `USED_IN_CONSENSUS` - read used in consensus,
   - `CONSENSUS_DISCARDED_TRIM` - consensus was calculated and discarded after quality trimming.
4. Number of reads in this consensus, or 0 if this read was not used in consensus.
5. Sequences and qualities for each target of this read.
6. Sequences and qualities for each target of this consensus, or "-" sign if this read was not used in consensus.
7. Levenshtein distances between this read and this consensus, separate values for all targets (`R1`, `R2` etc), or -1 values if this read is not used in any consensus.
8. Number of nucleotides that were removed from this read on quality trimming, separate values for all targets (`R1`, `R2` etc).
9. Number of nucleotides that were removed from this consensus on quality trimming, separate values for all targets (`R1`, `R2` etc).

Command line arguments reference for consensus action:

```
--input: Input file in MIF format. If not specified, stdin will be used.
--output: Output file in MIF format. If not specified, stdout will be used.
--groups: List of groups that represent barcodes. All these groups must be sorted␣
↪with "sort" action.
--skipped-fraction-to-repeat: Fraction of reads skipped by score threshold that must␣
↪start the search for another consensus in skipped reads. Value 1 means always get␣
↪only 1 consensus from one set of reads with identical barcodes.
--max-consensuses-per-cluster: Maximal number of consensuses generated from 1 cluster.
↪ Every time this threshold is applied to stop searching for new consensuses,␣
↪warning will be displayed. Too many consensuses per cluster indicate that score␣
↪threshold, aligner width or skipped fraction to repeat is too low.
--reads-min-good-sequence-length: Minimal length of good sequence that will be still␣
↪considered good after trimming bad quality tails. This parameter is for trimming␣
↪input reads.
--reads-avg-quality-threshold: Minimal average quality for bad quality tails trimmer.␣
↪This parameter is for trimming input reads.
--reads-trim-window-size: Window size for bad quality tails trimmer. This parameter␣
↪is for trimming input reads.
--min-good-sequence-length: Minimal length of good sequence that will be still␣
↪considered good after trimming bad quality tails. This parameter is for trimming␣
↪output consensuses by quality and coverage.
--low-coverage-threshold: Coverage is calculated as number of reads that have letters␣
↪on current position divided by total number of reads for this consensus. Values␣
↪lower than this parameter will be considered low. This parameter is for trimming␣
↪output consensuses by quality and coverage.
--avg-quality-threshold: Minimal average quality for parts of consensus with good␣
↪coverage. This parameter is for trimming output consensuses by quality and coverage.
--avg-quality-threshold-for-low-coverage: Minimal average quality for parts of␣
↪consensus with low coverage. This parameter is for trimming output consensuses by␣
↪quality and coverage.
```

(continues on next page)

(continued from previous page)

```
--trim-window-size: Window size for bad quality tails trimmer. This parameter is for␣
↪trimming output consensuses by quality and coverage.
--original-read-stats: Save extra statistics for each original read into separate␣
↪file. Output file in space separated text format.
--not-used-reads-output: Write reads not used in consensus assembly into separate␣
↪file. Output file in MIF format.
--consensuses-to-separate-groups: If this parameter is specified, consensuses will␣
↪not be written as reads R1, R2 etc to output file. Instead, original sequences will␣
↪be written as R1, R2 etc and consensuses will be written as CR1, CR2 etc, so it␣
↪will be possible to cluster original reads by consensuses using filter /␣
↪demultiplex actions, or export original reads and corresponding consensuses into␣
↪separate reads using mif2fastq action.
-n, --number-of-reads: Number of reads to take; 0 value means to take the entire␣
↪input file.
--max-warnings: Maximum allowed number of warnings; -1 means no limit.
--threads: Number of threads for calculating consensus sequences.
--report: File to write report in human readable form. If not specified, report is␣
↪displayed on screen only.
--json-report: File to write command execution stats in JSON format.
--kmer-length: K-mer length. Also affects --min-good-sequence-length because good␣
↪sequence length must not be lower than k-mer length, so the biggest of --kmer-
↪length and --min-good-sequence-length will be used as --min-good-sequence-length␣
↪value.
--kmer-offset: Max offset from the middle of the read when searching k-mers.
--kmer-max-errors: Maximal allowed number of mismatches when searching k-mers in␣
↪sequences.
--overwrite-if-required: Overwrite output file if it is corrupted or if it was␣
↪generated from different input file or with different parameters. -f / --force-
↪overwrite overrides this option.
-f, --force-overwrite: Force overwrite of output file(s).
-nw, --no-warnings: Suppress all warning messages.
```

7.1.9 consensus-dma

Consensus actions are used to calculate consensus sequences for all combinations of barcode values. They also allow to find multiple consensuses in the same combination of barcodes if there are multiple sequences with the same barcodes in the data.

Important: *sort* action must be used before any consensus action with the same groups in `--groups` argument as in consensus action, otherwise consensus calculation will consume much more memory!

Consensus algorithm "Double multi-align" (`consensus-dma` action) uses multi-sequence alignment to put multiple sequences to the same coordinate system. Then it calculate consensus letter (or deletion) for each position. `consensus-dma` action works in 2 stages: first it aligns the cluster (group of sequences with same barcodes) to the best sequence from this cluster; best sequence is determined by length and quality. On 2nd stage, it aligns all sequences from this cluster to the consensus from 1st stage. After both stages, quality trimming is performed on both sides of the consensus, and if the resulting sequence is too short, this consensus is discarded. Sequences that have low alignment score with the best sequence are not included in consensus calculation. But if there are many remaining sequences (both in case that consensus was calculated and in case it was discarded), next consensus calculation will start with the cluster of remaining sequences.

`--groups` argument with space separated list of groups must be specified in `consensus-dma` action to specify which groups will be used for consensus calculation. This argument is mandatory. `--input` and `--output` arguments are optional, and if they are missing, stdin and stdout will be used instead of input and output files.

Examples for `consensus-dma` action:

```
minnn consensus-dma --groups SB1 SB2 --input sorted.mif --output consensus.mif --
↪score-threshold -750
xzcat sorted.mif.xz | minnn consensus-dma --max-consensuses-per-cluster 1 --output␣
↪consensus.mif
minnn consensus-dma --consensuses-to-separate-groups --width 10 --input data.mif --
↪output result.mif
```

`--score-threshold` is very important option: it is alignment score threshold by which it will be determined to include sequence to the consensus or don't include. Too low threshold score can result in including garbage data to consensus or melding multiple molecules to one consensus. Too high threshold score can result in splitting one molecule to multiple consensuses, and in leaving out sequences with valuable data. Score values for single matches, mismatches and indels in alignment can be set with `--aligner-match-score`, `--aligner-mismatch-score` and `--aligner-gap-score` arguments.

`consensus-dma` action uses banded aligner, it reduces size of the matrix to increase speed and reduce memory usage. `--width` option allows to specify width of the banded matrix. Lower values mean faster consensus calculation and lower memory usage, but less accuracy. It's recommended to set `--width` to maximum allowed length of single insertion or deletion, multiplied by 1.5.

`--max-consensuses-per-cluster` is important option: it allows to set maximal number of consensuses for one combination of barcode values. If each molecule is marked by unique set of barcodes, you can set `--max-consensuses-per-cluster` to `1`.

If enough sequences from cluster were not used in consensus calculations, and `--max-consensuses-per-cluster` is not exceeded, then new consensus calculation will start from remaining sequences. You can set the threshold with `--skipped-fraction-to-repeat` option. If number of remaining sequences divided to cluster size is not below this threshold, new consensus calculation will start, otherwise all reads with remaining sequences will be discarded and not used in any consensus calculation. Also, if `--max-consensuses-per-cluster` is exceeded, all remaining reads will be discarded. `--not-used-reads-output` argument allows to write all discarded reads to the separate MIF file.

Calculated consensuses are processed with quality trimmer to remove low quality tails on the left and right sides. You can set trim window size (length of subsequence for which average quality is calculated) with `--trim-window-size` option. Quality threshold is set by `--avg-quality-threshold` option, and minimal allowed consensus length after trimming - by `--min-good-sequence-length` option. If resulting consensus will be shorter, it will be discarded.

Input reads are prepared with the same quality trimmer before calculating consensus, but trimming parameters are configured separately. Use `--reads-trim-window-size` to set trim window size for input reads, `--reads-avg-quality-threshold` for quality threshold and `--reads-min-good-sequence-length` for minimum input read length after trimming. Too short reads will not be included in consensus calculation.

`--consensuses-to-separate-groups` parameter changes action behavior significantly. If this parameter is not specified, output file will contain calculated consensuses. If it is specified, original sequences will be written to the output files, consensuses will be written as capture groups `CR1`, `CR2` etc, so it will be possible to cluster original reads by consensuses using *filter* / *demultiplex* actions, or export original reads and corresponding consensuses into separate reads using *mif2fastq* action. Note that input file must not contain any groups named `CR1`, `CR2` etc if you use `--consensuses-to-separate-groups` parameter.

`--original-read-stats` parameter allows to write consensus calculation stats for each original read into separate file. This is text file in space separated format, and it contains the following information for each read:

1. Original read ID: number of read, starting from 0, in the original FASTQ data that was the input of *extract* action.
2. Consensus ID (number of consensus in the output) or -1 if this read was discarded.
3. Read status. Possible values are:

- `NOT_MATCHED` - read was not matched in *extract* action,
- `READ_DISCARDED_TRIM` - read was discarded by length after quality trimming,
- `NOT_USED_IN_CONSENSUS` - read was discarded and not used in any consensus calculation,
- `USED_IN_CONSENSUS` - read used in consensus,
- `CONSENSUS_DISCARDED_TRIM_STAGE1` - consensus was calculated and discarded after quality trimming on the 1st stage,
- `CONSENSUS_DISCARDED_TRIM_STAGE2` - consensus was calculated and discarded after quality trimming on the 2nd stage.

4. ID of the best read of this consensus that was determined on stage 1, or -1 if this read was not used in consensus.
5. Number of reads in this consensus, or 0 if this read was not used in consensus.
6. Sequences and qualities for each target of this read.
7. Sequences and qualities for each target of this consensus, or "-" sign if this read was not used in consensus.
8. Levenshtein distances between this read and this consensus, separate values for all targets (`R1`, `R2` etc), or -1 values if this read is not used in any consensus.
9. Number of nucleotides that were removed from this read on quality trimming, separate values for all targets (`R1`, `R2` etc).
10. Number of nucleotides that were removed from this consensus on quality trimming, separate values for all targets (`R1`, `R2` etc).
11. Alignment scores for 1st and 2nd stages of consensus calculation, or large negative value if this read was not used in consensus.

Command line arguments reference for consensus-dma action:

```
--input: Input file in MIF format. If not specified, stdin will be used.
--output: Output file in MIF format. If not specified, stdout will be used.
--groups: List of groups that represent barcodes. All these groups must be sorted␣
↪with "sort" action.
--width: Window width (maximum allowed number of indels) for banded aligner.
--aligner-match-score: Score for perfectly matched nucleotide, used in sequences␣
↪alignment.
--aligner-mismatch-score: Score for mismatched nucleotide, used in sequences␣
↪alignment.
--aligner-gap-score: Score for gap or insertion, used in sequences alignment.
--good-quality-mismatch-penalty: Extra score penalty for mismatch when both sequences␣
↪have good quality.
--good-quality-mismatch-threshold: Quality that will be considered good for applying␣
↪extra mismatch penalty.
--score-threshold: Score threshold that used to filter reads for calculating␣
↪consensus.
--skipped-fraction-to-repeat: Fraction of reads skipped by score threshold that must␣
↪start the search for another consensus in skipped reads. Value 1 means always get␣
↪only 1 consensus from one set of reads with identical barcodes.
--max-consensuses-per-cluster: Maximal number of consensuses generated from 1 cluster.
↪ Every time this threshold is applied to stop searching for new consensuses,␣
↪warning will be displayed. Too many consensuses per cluster indicate that score␣
↪threshold, aligner width or skipped fraction to repeat is too low.
--reads-min-good-sequence-length: Minimal length of good sequence that will be still␣
↪considered good after trimming bad quality tails. This parameter is for trimming␣
↪input reads.
--reads-avg-quality-threshold: Minimal average quality for bad quality tails trimmer.␣
↪This parameter is for trimming input reads.
```

(continues on next page)

(continued from previous page)

```
--reads-trim-window-size: Window size for bad quality tails trimmer. This parameter␣
↪is for trimming input reads.
--min-good-sequence-length: Minimal length of good sequence that will be still␣
↪considered good after trimming bad quality tails. This parameter is for trimming␣
↪output consensuses by quality and coverage.
--low-coverage-threshold: Coverage is calculated as number of reads that have letters␣
↪on current position divided by total number of reads for this consensus. Values␣
↪lower than this parameter will be considered low. This parameter is for trimming␣
↪output consensuses by quality and coverage.
--avg-quality-threshold: Minimal average quality for parts of consensus with good␣
↪coverage. This parameter is for trimming output consensuses by quality and coverage.
--avg-quality-threshold-for-low-coverage: Minimal average quality for parts of␣
↪consensus with low coverage. This parameter is for trimming output consensuses by␣
↪quality and coverage.
--trim-window-size: Window size for bad quality tails trimmer. This parameter is for␣
↪trimming output consensuses by quality and coverage.
--original-read-stats: Save extra statistics for each original read into separate␣
↪file. Output file in space separated text format.
--not-used-reads-output: Write reads not used in consensus assembly into separate␣
↪file. Output file in MIF format.
--consensuses-to-separate-groups: If this parameter is specified, consensuses will␣
↪not be written as reads R1, R2 etc to output file. Instead, original sequences will␣
↪be written as R1, R2 etc and consensuses will be written as CR1, CR2 etc, so it␣
↪will be possible to cluster original reads by consensuses using filter /␣
↪demultiplex actions, or export original reads and corresponding consensuses into␣
↪separate reads using mif2fastq action.
-n, --number-of-reads: Number of reads to take; 0 value means to take the entire␣
↪input file.
--max-warnings: Maximum allowed number of warnings; -1 means no limit.
--threads: Number of threads for calculating consensus sequences.
--report: File to write report in human readable form. If not specified, report is␣
↪displayed on screen only.
--json-report: File to write command execution stats in JSON format.
--overwrite-if-required: Overwrite output file if it is corrupted or if it was␣
↪generated from different input file or with different parameters. -f / --force-
↪overwrite overrides this option.
-f, --force-overwrite: Force overwrite of output file(s).
-nw, --no-warnings: Suppress all warning messages.
```

7.1.10 stat-groups

Stat-groups action is used to collect summary statistics about values of capture groups in MIF file and create table with sequences, qualities and counts for these capture groups. The table is plain text, space separated. Example:

```
G1.seq G1.qual.min G1.qual.avg G2.seq G2.qual.min G2.qual.avg G3.seq G3.qual.min G3.
↪qual.avg G4.seq G4.qual.min G4.qual.avg count percent
TCTCAG 111111 FFFFFF CGA 1// FFF GGAGC ////0 FFFFF CG // FF 1937 7.8%
TCTCAG 111111 FFFFFF ACA ... EFE GGTGC ../// EEEEF CT ./ FF 1638 6.6%
TCTCAG 111111 FFFFFF AGA 0// FFE AGTAC ///// FFFFF AA // FE 1598 6.44%
TCTCAG 111111 FFFFFF AGA ../ FFF AGTAC ///// EFFFF AC // FF 1425 5.74%
TCTCAG ;11:1: FFFFFF CAA 900 FFF AGTAC 01:>> FFFFF AA 10 FF 1122 4.52%
TCTCAG 111111 FFFFFF AAA 011 EFF GTCAC 11111 FFFFF AT 11 FF 1050 4.23%
TCTCAG 111111 FFFFFF AGA /./ FFF GGGGC /.... FEFEF GA .. FE 1025 4.13%
```

`--groups` argument is mandatory: you must specify space separated list of groups to collect statistics. Stat-groups action will count occurrences for each unique combination of group values, or for each unique value if only 1 group is specified. The table is sorted by number of occurrences (`count` column).

This table contains the following columns:

`GROUPNAME.seq` (for each group) - value (nucleotide sequence) of the group

`GROUPNAME.qual.min` (for each group) - minimal quality for each letter from all occurrences of this value

`GROUPNAME.qual.avg` (for each group) - average quality for each letter counted by all occurrences of this value

`count` - number of occurrences of this value

`percent` - percentage of this value occurrences in all checked reads (in the entire input file if `--number-of-reads` argument is not specified)

Examples for stat-groups action:

```
minnn stat-groups --groups UMI --input corrected.mif --read-quality-filter 10 --min-
↪frac-filter 0.05
xzcat extracted.mif.xz | minnn stat-groups --groups G1 G2 G3 --output stat-groups.txt␣
↪--min-count-filter 100 -n 10000
```

`--input` argument means input file in MIF format, or if this argument is missing, stdin will be used. `--output` argument means output plain text file where the table will be written. If `--output` argument is missing, the table will be written to stdout.

`--read-quality-filter` argument allows to filter input reads by **minimal** quality of their letters. If any letter in any group value will have quality below this threshold, the entire read will be ignored (but still counted as checked read: number of checked reads is used as total in `percent` column and as stop condition in `--number-of-reads` argument).

`--min-quality-filter` argument allows to exclude table lines in which minimal quality for at least 1 letter is below the specified threshold. `--avg-quality-filter` argument allows to exclude table lines in which average quality of at least 1 group is below the specified threshold. Quality of a group is calculated as average quality of all letters of this group.

`--min-count-filter` argument allows to exclude table lines with `count` lower than the specified threshold, and `--min-frac-filter` allows to exclude lines where count divided to total number of checked reads is below the threshold (this is like value in `percent` column, but in fractions of 1, not percents).

`--number-of-reads` or `-n` argument allows to collect statistics not from the entire file, but from first specified number of reads.

In the end, stat-groups action will display total percentage of reads included in the table (that passed all filters) to the total number of checked reads.

Command line arguments reference for stat-groups action:

```
--groups: Space separated list of groups to output, determines the keys by which the␣
↪output table will be aggregated.
--input: Input file in MIF format. If not specified, stdin will be used.
--output: Output text file. If not specified, stdout will be used.
--read-quality-filter: Filter group values with a min (non-aggregated) quality below␣
↪a given threshold, applied on by-read basis, should be applied prior to any␣
↪aggregation. 0 value means no threshold.
--min-quality-filter: Filter group values based on min aggregated quality. 0 value␣
↪means no filtering.
--avg-quality-filter: Filter group values based on average aggregated quality. 0␣
↪value means no filtering.
--min-count-filter: Filter unique group values represented by less than specified␣
↪number of reads.
```

(continues on next page)

(continued from previous page)

```
--min-frac-filter: Filter unique group values represented by less than specified␣
↪fraction of reads.
-n, --number-of-reads: Number of reads to take; 0 value means to take the entire␣
↪input file.
--report: File to write brief command execution stats in human readable form. If not␣
↪specified, these stats are displayed on screen only.
--json-report: File to write command execution stats in JSON format.
-f, --force-overwrite: Force overwrite of output file(s).
```

7.1.11 stat-positions

Stat-positions action is used to collect summary statistics about positions of group matches in MIF file and create table with group names, reads where they matched, their match positions, and counts for each combination occurrences. Example:

```
group.id read pos count percent
G1 R1 15 4240 4.27%
G1 R2 15 3936 3.96%
G1 R1 16 3490 3.51%
G1 R2 16 3387 3.41%
G1 R1 17 2203 2.22%
G1 R2 17 1967 1.98%
G1 R2 18 724 0.73%
G1 R1 18 702 0.71%
G1 R2 19 627 0.63%
G1 R1 19 579 0.58%
G4 R1 10 446 0.45%
G4 R2 10 426 0.43%
G2 R1 1 407 0.41%
G2 R1 4 406 0.41%
G3 R1 4 406 0.41%
```

`--groups` argument is mandatory: you must specify space separated list of groups to collect statistics. Stat-positions action will count occurrences for each group separately; occurrences are counted for unique combinations *group* + *read* + *position*. The table is sorted by number of occurrences (`count` column).

This table contains the following columns:

`group.id` - group name

`read` - read (`R1`, `R2`, `R3` etc) where this group matched

`pos` - position of this match inside read, starting from 0

`count` - number of occurrences

`percent` - number of occurrences in percentage to number of all checked reads (to number of lines the entire input file if `--number-of-reads` argument is not specified)

Examples for stat-positions action:

```
minnn stat-positions --groups UMI --reads R2 R3 --input corrected.mif -n 10000
xzcat extracted.mif.xz | minnn stat-positions --groups G1 G2 G3 --output stat-
↪positions.txt --output-with-seq
```

`--input` argument means input file in MIF format, or if this argument is missing, stdin will be used. `--output` argument means output plain text file where the table will be written. If `--output` argument is missing, the table will be written to stdout.

`--reads` argument allows to include only matches in specified reads (`R1`, `R2`, `R3` etc) in the output table. Allowed reads specified as space separated list. If `--reads` argument is missing, all reads will be included.

`--output-with-seq` argument changes behavior of stat-positions action. New column `seq` is added to the table, and it contains nucleotide sequence for this match. With this argument occurrences are counted not by *group* + *read* + *position* (default), but by *group* + *read* + *position* + *sequence*. Output table example with `--output-with-seq` argument:

```
group.id read pos count percent seq
G1 R1 15 4231 4.26% TCTCAG
G1 R2 15 3927 3.95% TCTCAG
G1 R1 16 3484 3.51% TCTCAG
G1 R2 16 3379 3.4% TCTCAG
G1 R1 17 2200 2.22% TCTCAG
G1 R2 17 1964 1.98% TCTCAG
G1 R2 18 715 0.72% TCTCAG
G1 R1 18 694 0.7% TCTCAG
G1 R2 19 626 0.63% TCTCAG
G1 R1 19 579 0.58% TCTCAG
G4 R1 10 437 0.44% AC
G4 R2 10 422 0.42% AC
G4 R2 13 351 0.35% AT
G3 R2 8 349 0.35% GTCAC
G2 R2 5 348 0.35% AAA
G4 R1 13 344 0.35% AT
G4 R1 40 342 0.34% CG
G1 R1 15 342 0.34% TCTCAA
```

`--min-count-filter` argument allows to exclude table lines with `count` lower than the specified threshold, and `--min-frac-filter` allows to exclude lines where count divided to total number of checked reads is below the threshold (this is like value in `percent` column, but in fractions of 1, not percents).

`--number-of-reads` or `-n` argument allows to collect statistics not from the entire file, but from first specified number of reads.

In the end, stat-positions action will display total percentage of keys (*group* + *read* + *position* or *group* + *read* + *position* + *sequence* with `--output-with-seq` argument) that were included in the table (passed all filters) to the total number of checked keys.

Command line arguments reference for stat-positions action:

```
--groups: Space separated list of groups to output, determines IDs allowed in group.
↪id column.
--reads: Space separated list of original read IDs to output (R1, R2 etc), determines␣
↪IDs allowed in read column. If not specified, all reads will be used.
--output-with-seq: Also output matched sequences. If specified, key columns are group.
↪id + read + seq + pos; if not specified, key columns are group.id + read + pos.
--input: Input file in MIF format. If not specified, stdin will be used.
--output: Output text file. If not specified, stdout will be used.
--min-count-filter: Filter unique group values represented by less than specified␣
↪number of reads.
--min-frac-filter: Filter unique group values represented by less than specified␣
↪fraction of reads.
-n, --number-of-reads: Number of reads to take; 0 value means to take the entire␣
↪input file.
--report: File to write brief command execution stats in human readable form. If not␣
↪specified, these stats are displayed on screen only.
--json-report: File to write command execution stats in JSON format.
-f, --force-overwrite: Force overwrite of output file(s).
```

7.1.12 mif-info

MifInfo action is used to show information about MIF file: MiNNN version used to create the file, number of targets (R1, R2 etc), number of reads, original number of reads, group names, corrected and sorted groups. Original number of reads is number of reads in original data, before Extract action.

Input file in MIF format must be passed as argument for MifInfo action.

`--no-reads-count`, `--quick` or `-q` argument means to display only information from MIF header, so number of reads will not be displayed (because counting reads requires to read the entire MIF file); original number of reads will also not be displayed, because it is contained in MIF footer. Running MifInfo with this argument is much faster for big MIF files.

Examples for mif-info action:

```
minnn mif-info --quick data.mif
minnn mif-info consensus.mif --report consensus-mif-info.txt
```

Command line arguments reference for mif-info action:

```
-q, --quick, --no-reads-count: Don't count reads, display only info from header.
--report: File to write report in human readable form. If not specified, report is␣
↪displayed on screen only.
--json-report: File to write command execution stats in JSON format.
```

7.1.13 decontaminate

Decontaminate action is used to remove molecular barcodes from one cell that appear in samples from another cell. It counts molecular barcodes separately in each cell; cell is identified by `--primary-groups` command line argument. Then it filters reads, removing reads that contain molecular barcode with low count which appears in another cell with high count.

Decontaminate action works in 2 stages:

1. Reading input file and counting molecular barcodes in each cell.
2. Reading input file again, filtering reads by molecular barcodes and writing output file.

`--input` argument is mandatory, reading data from stdin is not supported because decontaminate action reads input file twice. `--output` argument is optional: results will be written to stdout if `--output` argument is missing. `--groups` argument is mandatory, it must contain space separated list of groups that identify a molecule (molecular barcodes). If multiple groups are specified, molecule is identified by unique combination of values of all specified groups. `--primary-groups` argument is also mandatory, it must contain space separated list of group that idenitfy a cell (cell barcodes). If multiple primary groups are specified, cell is identified by unique combination of values of all specified primary groups. Built-in groups R1, R2, R3 etc are not allowed both as molecular and as cell barcodes.

Examples for decontaminate action:

```
minnn decontaminate --primary-groups CB --groups UMI --input data.mif --output␣
↪filtered.mif --min-count-share 0.01
minnn decontaminate --primary-groups G1 G2 --groups M1 M2 --input extracted.mif | xz >
↪ decontaminated.mif.xz
minnn decontaminate --primary-groups A --groups B --input in.mif --output out.mif --
↪excluded-barcodes-output ex.mif
```

`--min-count-share` argument allows to specify the threshold for filtering out molecular barcodes. If a molecular barcode is present in a cell, but it's count is lower than count of the same barcode in different cell, multiplied on this share, then reads in the cell with lower count of this molecular barcode will be considered contaminated and will be filtered out.

`--excluded-barcodes-output` argument allows to write filtered out reads to the separate MIF file. By default (if this argument is not present) filtered out reads will not be written anywhere.

Command line arguments reference for decontaminate action:

```
--groups: Group names for molecular barcodes (UMI). Reads where these barcodes are␣
↪contaminated from other cells will be filtered out.
--primary-groups: Primary group names. These groups contains cell barcodes: each␣
↪combination of primary group values corresponds to 1 cell. Molecular barcodes are␣
↪counted separately for each cell, and then reads containing molecular barcodes with␣
↪significantly lower counts than in other cell will be removed.
--input: Input file in MIF format. This argument is required; stdin is not supported.
--output: Output file in MIF format. If not specified, stdout will be used.
--excluded-barcodes-output: Output file for reads with filtered out barcodes. If not␣
↪specified, reads with filtered out barcodes will not be written anywhere.
--min-count-share: Threshold for filtering out molecular barcodes. If count of a␣
↪molecular barcode is lower than count of the same barcode in different cell,␣
↪multiplied on this share, then reads in the cell with lower count of this barcode␣
↪will be considered contaminated and will be filtered out.
-n, --number-of-reads: Number of reads to take; 0 value means to take the entire␣
↪input file.
--report: File to write report in human readable form. If not specified, report is␣
↪displayed on screen only.
--json-report: File to write command execution stats in JSON format.
--overwrite-if-required: Overwrite output file if it is corrupted or if it was␣
↪generated from different input file or with different parameters. -f / --force-
↪overwrite overrides this option.
-f, --force-overwrite: Force overwrite of output file(s).
```

7.1.14 report

Report action is used to find match and groups in query and display report on the screen. Most of this action arguments are the same that of *extract* action, and it uses the same pattern syntax (see *Pattern Syntax* page). The difference from extract is that report action doesn't work with files; instead it uses pattern with single target, which can be single-read or multi-read, and displays result on the screen immediately. It can be useful to quickly test complex patterns before using them on big files in extract action.

Mandatory command line arguments are `--pattern` and `--target`. `--pattern` argument has the same syntax as in *extract* action. `--target` is used to specify target nucleotide sequence, or multiple space separated sequences for multi-read targets. Space separated multi-targets must be enclosed in double quotes `""`. Nucleotide sequences are without quality, case insensitive (`ATTAGACA` and `attagaca` means the same in the target), and can contain wildcards (N, W, S, M etc). Examples:

```
minnn report --pattern "(SB1:NNN)atta \ (SB2:NNN)gaca" --target "CCTCCCCACCA ATTAGACA"
minnn report --pattern "NCCN" --target WCNTDTBCDATD
minnn report --pattern "[TTT \ CCC] || [AAA \ GGG && AGC \ GTT]" --score-threshold 0 -
↪-target "aaagc gggtt"
```

Example output:

```
$ minnn report --pattern "(SB1:NNN)atta \ (SB2:NNN)gaca" --target "CCTCCCCACCA␣
↪ATTAGACA"
Found match in target 1 (CCTCCCCACCA): CCCACCA
```

(continues on next page)

(continued from previous page)

```
Range in this target: (4->11)

Found match in target 1 (ATTAGACA): TTAGACA
Range in this target: (1->8)

Found matched group SB1 in target 1 (CCTCCCCACCA): CCC
Range in this target: (4->7)

Found matched group R1 in target 1 (CCTCCCCACCA): CCTCCCCACCA
Range in this target: (0->11)

Found matched group SB2 in target 1 (ATTAGACA): TTA
Range in this target: (1->4)

Found matched group R2 in target 2 (ATTAGACA): ATTAGACA
Range in this target: (0->8)
```

Command line arguments reference for report action:

```
--pattern: Query, pattern specified in MiNNN format.
--target: Target nucleotide sequence, where to search.
--match-score: Score for perfectly matched nucleotide.
--mismatch-score: Score for mismatched nucleotide.
--uppercase-mismatch-score: Score for mismatched uppercase nucleotide.
--gap-score: Score for gap or insertion.
--score-threshold: Score threshold, matches with score lower than this will not go to␣
↪output.
--good-quality-value: This or better quality value will be considered good quality,␣
↪without score penalties.
--bad-quality-value: This or worse quality value will be considered bad quality, with␣
↪maximal score penalty.
--max-quality-penalty: Maximal score penalty for bad quality nucleotide in target.
--single-overlap-penalty: Score penalty for 1 nucleotide overlap between neighbor␣
↪patterns. Negative value or 0.
--max-overlap: Max allowed overlap for 2 intersecting operands in +, & and pattern␣
↪sequences. Value -1 means unlimited overlap size.
--bitap-max-errors: Maximum allowed number of errors for bitap matcher.
--fair-sorting: Use fair sorting and fair best match by score for all patterns.
```

CHAPTER 8

Pattern Syntax

Patterns are used in *extract* action to specify which sequences must pass to the output and which sequences must be filtered out. Also, capture groups in patterns are used for barcode extraction. Patterns must always be specified after `--pattern` option and must always be in double quotes. Examples:

```
minnn extract --pattern "ATTAGACA"
minnn extract --pattern "*\*" --input R1.fastq R2.fastq
minnn extract --pattern "^(UMI:N{3:5})attwwAAA\*" --input-format mif
```

8.1 Basic Syntax Elements

Many syntax elements in patterns are similar to regular expressions, but there are differences. Uppercase and lowercase letters are used to specify the sequence that must be matched, but uppercase letters don't allow indels between them and lowercase letters allow indels. Indels on left and right borders of uppercase letters are also not allowed. Also, score penalty for mismatches in uppercase and lowercase letters can be different: `--mismatch-score` parameter used for lowercase mismatches and `--uppercase-mismatch-score` for uppercase mismatches. Standard IUPAC wildcards (N, W, S, M etc) are also allowed in both uppercase and lowercase sequences.

\ character is very important syntax element: it used as read separator. There can be single-read input files, in this case \ character must not be used. In multi-read inputs \ must be used, and number of reads in pattern must be equal to number of input FASTQ files (or to number of reads in input MIF file if `--input-format MIF` parameter is used). There can be many reads, but the most common case is 2 reads: `R1` and `R2`. By default, extract action will check input reads in order they specified in `--input` argument, or if input file is MIF, then in order they saved in MIF file. If `--try-reverse-order` argument is specified, it will also try the combination with 2 swapped last reads (for example, if there are 3 reads, it will try `R1, R2, R3` and `R1, R3, R2` combinations), and then choose the match with better score.

Another important syntax element is capture group. It looks like `(group_name:query)` where `group_name` is any sequence of letters and digits (like `UMI` or `SB1`) that you use as group name. Group names are case sensitive, so `UMI` and `umi` are different group names. `query` is part of query that will be saved as this capture group. It can contain nested groups and some other syntax elements that are allowed inside single read (see below).

R1, R2, R3 etc are built-in group names that contain full matched reads. You can override them by specifying manually in the query, and overridden values will go to output instead of full reads. For example, query like this

```
minnn extract --input R1.fastq R2.fastq --pattern "^NNN(R1:(UMI:NNN)ATTAN{*})\^
↪NNN(R2:NNNGACAN{*})"
```

can be used if you want to strip first 3 characters and override built-in R1 and R2 groups to write output reads without stripped characters. Note that R1, R2, R3 etc, like any common groups, can contain nested groups and can be nested inside other groups.

Important: in matches that come from swapped reads (when --try-reverse-order argument is specified), if you don't use built-in group names override, R1 in input will become R2 in output and vice versa (or there can be, for example, swapped R2 and R3 in case of 3 reads). If you use the override, R1, R2, R3 etc in output will come from the place where they matched. If you export the output MIF file from *extract* action to FASTQ and want to determine whether the match came from straight or swapped reads, check the comments for ||~ character sequence: it is added to matches that came from swapped reads. Look at *mif2fastq* section for detailed information.

* character can be used instead of read contents if any contents must match. It can be enclosed in one or multiple capture groups, but can't be used if there are other query elements in the same read. If there are other query elements, use N{*} instead. For example, the following queries are **valid**:

```
minnn extract --input R1.fastq R2.fastq --try-reverse-order --pattern
↪"(G1:ATTA)\(G2:(G3:*))"
minnn extract --input R1.fastq R2.fastq R3.fastq --pattern "*\*\*"
minnn extract --input R1.fastq R2.fastq --pattern "(G1:ATTAN{*})\(G2:*)"
```

and this is **invalid**:

```
minnn extract --input R1.fastq R2.fastq --pattern "(G1:ATTA*)\*"
```

Curly brackets after nucleotide can be used to specify number of repeats for the nucleotide. There can be any nucleotide letter (uppercase or lowercase, basic or wildcard) and then curly braces with quantity specifier. The following syntax constructions are allowed:

a{*} - any number of repeats, from 1 to the entire sequence

a{:} - same as the above

a{14} - fixed number of repeats

a{3:6} - specified interval of allowed repeats, interval borders are inclusive

a{:5} - interval from 1 to specified number, inclusive

a{4:} - interval from specified number (inclusive) to the entire sequence

Special Case: if n or N nucleotide is used before curly brackets, indels and pattern overlaps (see --max-overlap parameter below) are disabled, so lowercase n and uppercase N are equivalent when used before curly brackets.

Symbols ^ and $ can be used to restrict matched sequence to start or end of the target sequence. ^ mark must be in the start of the query for the read, and it means that the query match must start from the beginning of the read sequence. $ mark must be in the end, and it means that the query match must be in the end of the read. Examples:

```
minnn extract --pattern "^ATTA"
minnn extract --input R1.fastq R2.fastq --pattern "TCCNNWW$\^(G1:ATTAGACA)N{3:18}
↪(G2:ssttggca)$"
```

8.2 Advanced Syntax Elements

There are operators `&`, `+` and `||` that can be used inside the read query.

`&` operator is logical AND, it means that 2 sequences must match in any order and gap between them. Examples:

```
minnn extract --pattern "ATTA & GACA"
minnn extract --input R1.fastq R2.fastq --pattern "AAAA & TTTT & CCCC \ *"
minnn extract --input R1.fastq R2.fastq --pattern "(G1:AAAA) & TTTT & CCCC \ ATTA &␣
↪(G2:GACA)"
```

Note that `AAAA`, `TTTT` and `CCCC` sequences can be in any order in the target to consider that the entire query is matching. `&` operator is not allowed within groups, so this example is **invalid**:

```
minnn extract --pattern "(G1:ATTA & GACA)"
```

`+` operator is also logical AND but with order restriction. Nucleotide sequences can be matched only in the specified order. Also, `+` operator can be used within groups. Note that in this case the matched group will also include all nucleotides between matched operands. Examples:

```
minnn extract --pattern "(G1:ATTA + GACA)"
minnn extract --input R1.fastq R2.fastq --pattern "(G1:AAAA + TTTT) + CCCC \ ATTA +␣
↪(G2:GACA)"
```

`||` operator is logical OR. It is not allowed within groups, but groups with the same name are allowed inside operands of `||` operator. Note that if a group is present in one operand of `||` operator and missing in another operand, this group may appear not matched in the output while the entire query is matched. Examples:

```
minnn extract --pattern "^AAANNN(G1:ATTA) || ^TTTNNN(G1:GACA)"
minnn extract --input R1.fastq R2.fastq --pattern "(G1:AAAA) || TTTT || (G1:CCCC) \␣
↪ATTA || (G2:GACA)"
```

`+`, `&` and `||` operators can be combined in single query. `+` operator has the highest priority, then `&`, and `||` has the lowest. Read separator (\) has lower priority than all these 3 operators. To change the priority, square brackets `[]` can be used. Examples:

```
minnn extract --pattern "^[AAA & TTT] + [GGG || CCC]$"
minnn extract --input R1.fastq R2.fastq --pattern "[(G1:ATTA+GACA)&TTT]+CCC\(G2:AT+AC)
↪"
```

Square brackets can be used to create sequences of patterns. Sequence is special pattern that works like `+` but with penalty for gaps between patterns. Examples of sequence pattern:

```
minnn extract --pattern "[AAA & TTT]CCC"
minnn extract --input R1.fastq R2.fastq --pattern "[(G1:ATTA+GACA)][(G2:TTT)&ATT]\*"
```

Also square brackets allow to set separate score threshold for the query inside brackets. This can be done by writing score threshold value followed by `:` after opening bracket. Examples:

```
minnn extract --pattern "[-14:AAA & TTT]CCC"
minnn extract --input R1.fastq R2.fastq --pattern "[0:(G1:ATTA+GACA)][(G2:TTT)&ATT]\[-
↪25:c{*}]"
```

Matched operands of `&`, `+` and sequence patterns can overlap, but overlaps add penalty to match score. You can control maximum overlap size and overlapping letter penalty by `--max-overlap` and `--single-overlap-penalty` parameters. `-1` value for `--max-overlap` parameters means no restriction on maximum overlap size.

Important: parentheses that used for groups are not treated as square brackets; instead, they treated as group edges attached to nucleotide sequences. So, the following examples are different: first example creates sequence pattern and second example adds end of G1 and start of G2 to the middle of sequence TTTCCC.

```
minnn extract --pattern "[(G1:AAA+TTT)][(G2:CCC+GGG)]"
minnn extract --pattern "(G1:AAA+TTT)(G2:CCC+GGG)"
```

If some of nucleotides on the edge of nucleotide sequence can be cut without gap penalty, tail cut pattern can be used. It looks like repeated < characters in the beginning of the sequence, or repeated > characters in the end of the read, or single < or > character followed by curly braces with number of repeats. It is often used with ^/$ marks. Examples:

```
minnn extract --input R1.fastq R2.fastq --pattern "^<<<ATTAGACA>>$\[^<TTTT || ^<<CCCC]
↪"
minnn extract --input R1.fastq R2.fastq --pattern "<{6}ACTCACTCGC + GGCTCGC>{2}$\<
↪<AATCC>"
```

Important: < and > marks belong to nucleotide sequences and not to complex patterns, so square brackets between < / > and nucleotide sequences are **not** allowed. Also, the following examples are different: in first example edge cut applied only to the first operand, and in second example - to both operands.

```
minnn extract --pattern "<{3}ATTA & GACA"
minnn extract --pattern "<{3}ATTA & <{3}GACA"
```

8.3 High Level Logical Operators

There are operators ~, && and || that can be used with full multi-read queries. Note that || operator have the same symbol as read-level OR operator, so square brackets must be used to use high level ||.

|| operator is high-level OR. Groups with the same name are allowed in different operands of this operator, and if a group is present in one operand of || operator and missing in another operand, this group may appear not matched in the output while the entire query is matched. Examples:

```
minnn extract --pattern "[AA\*\TT] || [*\GG\CG]" --input R1.fastq R2.fastq R3.fastq
minnn extract --pattern "[^(G1:AA) + [ATTA || GACA]$ \ *] || [AT(G1:N{:8})\(G2:AATGC)]
↪" --input R1.fastq R2.fastq
```

&& operator is high-level AND. For AND operator it is not necessary to enclose multi-read query in square brackets because there is no ambiguity. Groups with the same name are **not** allowed in different operands of && operator. Examples:

```
minnn extract --pattern "AA\*\TT && *\GG\CG" --input R1.fastq R2.fastq R3.fastq
minnn extract --pattern "^(G1:AA) + [ATTA || GACA]$ \ * && AT(G2:N{:8})\(G3:AATGC)" --
↪input R1.fastq R2.fastq
```

~ is high-level NOT operator with single operand. It can sometimes be useful with single-read queries to filter out wrong data. Groups are **not** allowed in operand of ~ operator.

```
minnn extract --pattern "~ATTAGACA"
minnn extract --pattern "~[TT \ GC]" --input R1.fastq R2.fastq
```

Important: ~ operator always belongs to multi-read query that includes all input reads, so this example is **invalid**:

```
minnn extract --pattern "[~ATTAGACA] \ TTC" --input R1.fastq R2.fastq
```

Instead, this query can be used:

```
minnn extract --pattern "~[ATTAGACA \ *] && * \ TTC" --input R1.fastq R2.fastq
```

Note that if `--try-reverse-order` argument is specified, reads will be swapped synchronously for all multi-read queries that appear as operands in the entire query, so this query will never match:

```
minnn extract --pattern "~[ATTA \ *] && ATTA \ *" --input R1.fastq R2.fastq
```

Square brackets are not required for `~` operator, but recommended for clarity if input contains more than 1 read. `~` operator have lower priority than `\`; `&&` has lower priority than `~`, and high-level `||` has lower priority than `&&`. But remember that high-level `||` requires to enclose operands or multi-read blocks inside operands into square brackets to avoid ambiguity with read-level OR operator.

Square brackets with score thresholds can be used with high-level queries too:

```
minnn extract --pattern "~[0: ATTA \ GACA && * \ TTT] || [-18: CCC \ GGG]" --input R1.
↪fastq R2.fastq
```

CHAPTER 9

Appendix

9.1 Filter Syntax

Filters are used in *filter* action to set filtering conditions. There are 2 general cases of using filter action:

1. Filtering input data by group value or length.
2. Filtering output of *consensus* and *consensus-dma* actions to exclude consensuses assembled from too small number of reads.

Filtering query must always be specified as separate argument in double quotes. Examples:

```
minnn filter --input extracted.mif --output filtered.mif "UMI~'ATTAGACA'"
minnn filter "Len(SB)=11" --input corrected.mif --output filtered.mif
minnn filter --input consensus.mif "MinConsensusReads=150" --output consensus_
↪filtered.mif
minnn filter "Len(G1)=4 & Len(G2)=6 & G2~'TCCA'"
```

Filter syntax uses 3 basic filters: pattern filter, length filter and consensus reads filter. Also, there are logic operators and parentheses that can be used to combine multiple filters in 1 action.

9.1.1 Basic Filters

Pattern filter is used for filtering reads by group contents. Only reads where group value matches the pattern will be passed to the output. Pattern syntax is the same as for *extract* action, it is described in in *Pattern Syntax* section. Pattern filter uses the following syntax: `group_name~'pattern_query'`. Pattern query must always be in single quotes (`''`). Groups (parentheses) and read separators (\) are **not** allowed in pattern query. Examples of pattern filter usage:

```
minnn filter "UMI~'GCC || ^TCA'"
minnn filter "GROUP1~'ATTA'"
minnn filter "G1~'~ATT$ && ~^GCC'"
```

Length filter is used for filtering reads by group length. The syntax is `Len(group_name)=value`. Only reads where length of the specified group equals to the `value` will be passed to the output. Example:

```
minnn filter "Len(G1) = 3"
```

Group quality filters are used to filter out reads with low quality barcodes. The syntax for these filters is `MinGroupQuality(group_name)=value` and `AvgGroupQuality(group_name)=value`. `MinGroupQuality` will filter out reads where at least 1 nucleotide in the specified group has quality lower than the specified value. `AvgGroupQuality` will filter out reads where average quality of all nucleotides in the specified group is lower than the specified value. Examples:

```
minnn filter "MinGroupQuality(G1) = 7"
minnn filter "AvgGroupQuality(UMI) = 20"
```

N count filters can be used to filter out matched barcodes with too many `N` letters. `GroupMaxNCount(group_name)=value` excludes reads where the specified groups contains more `N` letters than the specified value. `GroupMaxNFraction(group_name)=value` allows to specify the maximal number of `N` letters as a fraction of group length. Specified value in this filter must be floating point in range from `0` to `1`. Examples:

```
minnn filter "GroupMaxNCount(SB) = 3"
minnn filter "GroupMaxNFraction(UMI) = 0.1"
```

All filters that have `group_name` as argument allow to use `*` instead of group name. This option allows to apply filter to all groups in the input (except built-in groups `R1`, `R2` etc). Examples:

```
minnn filter "Len(*) = 5"
minnn filter "MinGroupQuality(*) = 10"
minnn filter "AvgGroupQuality(*) = 15"
minnn filter "GroupMaxNCount(*) = 0"
minnn filter "GroupMaxNFraction(*) = 0.15"
```

Consensus reads filter is used for filtering MIF files written by *consensus* and *consensus-dma* actions. The syntax is `MinConsensusReads=value`. Only consensuses calculated from `value` or more reads will be passed to the output. Example:

```
minnn filter "MinConsensusReads = 18"
```

9.1.2 Logic Operators

There are logic operators `&` (AND) and `|` (OR) that can be used in filtering query to combine multiple basic filters. There can be multiple logic operators in 1 query; `&` has higher priority than `|`. Parentheses `()` can be used to manage operations priority. Examples:

```
minnn filter "MinConsensusReads=25 & G1~'TCGCC'"
minnn filter "G1~'N{4:8}' & (G2~'ATTA' | G3~'GACA')"
minnn filter "Len(G1)=10 & Len(G2)=8 | Len(G1)=8 & Len(G2)=10"
```

If there are many arguments for `|` operator, `--whitelist-patterns` option is more convenient way to specify them. Arguments can be specified in a text file instead of filter query. For more information, see *filter* action.

CHAPTER 10

License

CHAPTER 11

Description

Minnn is a toolset to process genetic data from sequencing machines and assemble sequenced molecules from raw FASTQ data. Consensus assembly in minnn consists of the following stages:

1. Extract barcodes from raw sequences.
2. Sort sequences by barcode values to group them for further correction.
3. Correct mismatches and indels in barcodes.
4. Sort sequences by barcode values to group them for further consensus assembly.
5. Assembly consensuses for each barcode. There can be one or many consensuses for each barcode, depending on the way of obtaining original data.
6. Export calculated consensuses to FASTQ format.

Also minnn has some other functions:

- Filter original data by barcode values.
- Filter calculated consensuses by quantity of reads from which they were assembled.
- Filter barcode values by their count.
- Decontaminate: remove barcodes from one cell that appear in samples from another cell.
- Split (demultiplex) data into separate files by barcode values.
- Collect statistics from data by barcode values and by barcode positions in sequences.

Minnn is free for academic and non-profit use (see *License*).

CHAPTER 12

Usage Chart