22110 - Python and Unix for Bioinformaticians

Read trimmer for NGS data

Authors

Celia Burgos Sequeros s202423
Viktor Törnblom s200116

Workload distribution

	Celia	Viktor
Code	50%	50%
Report	50%	50%

Danmarks Tekniske Universitet

DTU

November 30, 2020

Contents

1 Introduction

Next generation sequencing (NGS) is the high-throughput sequencing technology that has revolutionized the biological sciences by allowing researchers to perform a wide range of experiments that before were unthinkable because of timely and costly matters. With millions and up to billions of DNA/RNA fragments sequenced in parallel, the technique generates massive amounts of data with reads of varied quality [1]. In order to be able use the data for further analysis, pre-processing of the reads is of the essence.

Nowadays, various kinds of sequencing platforms are used for NGS and Illumina is one of the most frequently used. For this project, we have developed a tool that trims low quality Illumina reads and prepares them for the next steps of the NGS data analysis pipeline. This report gives a detailed description of how the tool is built and how the program is used. We call it the `magicClipper`.

2 Theory

The data generated during sequencing is stored in a format called 'fastq', where every read is composed of 4 lines:

- Line 1: The header, starting with '@' and followed by the sequence identifier
- Line 2: The raw DNA sequence
- Line 3: The separator, with '+' and some optional additional text
- Line 4: The Phred-encoded quality score, with a character for each base call

  Example:

  @HWI-ST1294:69:D18RUACXX:2:1101:6288:2223_1:N:0:

  CTCAAAAGACCTACCTCTTAATTCTATCACACTGGAGATTC

  +

  <@?DDDEFFDHHFGE@GEGGGGGIGIIEHGIBEFG@FCDGIDG

The fourth line of every read contains the Phred quality score, which is a string of ASCII-characters. Each character codes for a numeric value that describes the probability of the corresponding base call being incorrect [2]. Most modern NGS platforms today use the so called Phred 33 encoding scheme, however, older Illumina platforms used a Phred 64 encoded scheme, which uses different ASCII characters. Our tool is able to automatically detect the encoding type of the given input files and handle them accordingly.

There are several factors which may affect the quality of a read. For example, *phasing* is a phenomenon caused by unsynchronized sequencing of the bases of a read. This usually affects longer reads and leads to a drop in quality towards the 3' end [3]. The *sliding window*

technique can help in dealing with these lower quality regions. It works by calculating the average quality score of a set of bases (the window) and if said average is found to be below a given threshold, the outermost base is removed [4]. If a base is removed, the window slides one base closer towards the center of the sequence and the process is repeated. Another frequently seen issue is a reduction in the quality of the first ∼12 bases of the 5' end. This is usually solved by simply removing the first few bases for all reads [4].

Something else worth mentioning is that NGS sequencing platforms can run in two different modes: single end and paired end. While single end sequencing generates one read per sequence fragment (the so-called *forward* read), paired end sequencing generates two (the *forward* and the *reverse* reads), since sequencing is performed from the two ends of each fragment. Working with paired end data thus means working with two input files simultaneousy. As will be further explained in the **Algorithm design** section below, our tool is designed to handle both single and paired end data.

3 Algorithm design

3.1 File Reading

As previously mentioned, fastq files are extremely large (usually several Gb in size), so the only sensible way to read them is line by line. We do so with the *.readline()* method.

3.2 Data Types

The main data type used in this program is a list of 4 elements. Those four elements correspond to the four lines that conform a read:

> [header, raw DNA sequence, '+', Phred-encoded quality score]

An example would be:

> ['@HWI-ST1294:69:D18RUACXX:2:1101:6288:2223_1:N:0:',
>
> 'CTCAAAAGACCTACCTCTTAATTCTATCACACTGGAGATTCAT',
>
> '+'
>
> '<@?DDDEFFDHHFGE@GEGGGGGIGIIEHGIBEFG@FCDGIDG']

As is explained in more detail in the following section, only one of these lists is stored in memory at a time in the single end mode, and two (one for each file) in the paired end mode.

3.3 Handling Single vs Paired End Reads

The program works in single or paired end mode, depending on the number of fastq files passed as arguments in the command line.

When a single fastq file is given, the **single end mode** is initiated. Here, the file reading is simple:

Algorithm 1: Single end mode

```
for read in input file do
    process read
    if (quality is high) and (length is long) then
        print read to output file
```

When two fastq files are given, the **paired end mode** is initiated. Here the file reading is a bit trickier.

The way in which the two fastq files of a paired end sequencing run are designed is keeping the same read order in both; this is, if we look at the same number line of each file we will always see the same read. This reduces the complexity of the task enormously, and in this program we make the assumption that the input files follow this design.

Because of how the next steps of the NGS read processing pipeline work, all reads must remain paired after the trimming. The extent to which the two reads in one pair are trimmed need not be the same (e.g. the *forward* read can be trimmed 4nt from the 3' end and *reverse* read, 8nt from the 5'), but if one is removed completely due to bad quality or too short length, then the other must also be removed. This means that the reading is to be done simultaneously on the two files. The step sequence now looks like this:

Algorithm 2: Paired end mode

```
for (read1 in input file1) and (read2 in input file2) do
    process read1
    process read2
    if (qualities are high) and (lengths are long) then
        print read1 to output file 1
        print read2 to output file2
```

3.4 Main Algorithm

Algorithm 3: General algorithm of magicClipper

Input: One or two compressed or uncompressed .fastq files
Output: One or two compressed or uncompressed .fastq files and a log file

```
1  if input file(s) is(are) compressed then
2      Open compressed input file(s)
3      Open compressed output file(s)
4  else
5      Open input file(s)
6      Open output file(s)
7  if single file is given then
8      for read in input file do
9          decode quality
10         –TRIMMING–
11         trim read according to user input
12         trim read based on quality
13         –FILTERING–
14         if (read quality is low) OR (read length is short) then
15             move on to next read without printing
16         else
17             print processed read onto output file
18     calculate statistics
19     write log file
20 else if two files are given then
21     for (read1 in input file1) and (read2 input file2) do
22         decode qualities
23         –TRIMMING–
24         trim read1 according to user input
25         trim read1 based on quality
26         trim read2 according to user input
27         trim read2 based on quality
28         –FILTERING–
29         if (read1 or read2 quality is low) OR (read1 or read2 length is short) then
30             move on to next reads without printing
31         else
32             print read1 onto output file1 and read2 onto output file2
33     calculate statistics
34     write log file
35 close file(s)
```

3.5 Diagram

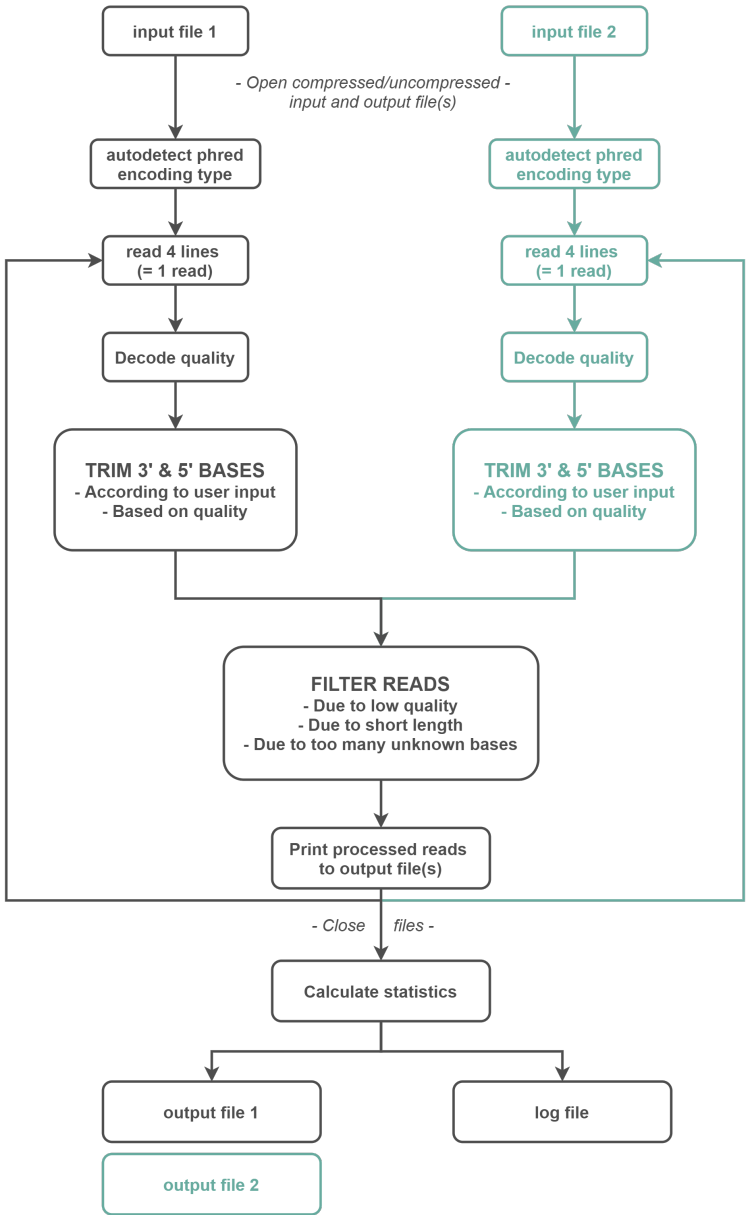

Figure 1: General algorithm of magicClipper. Created with `http://www.diagrams.net/`

4 Program design

4.1 Modules

magicClipper relies on 5 modules: *os*, *sys*, *gzip*, *argparse* and *trimmer_functions*. The *trimmer_functions* module compiles all the functions that were written specifically for this program, which can be found in the `trimmer_functions.py` file and are described below.

4.2 User input

For integrating the user's input in the program, we use the *argparse* library. This input must consist of one or two fastq files and (optionally) a number of parameters that allow the user to fine-tune the trimmer. Some examples are the number of leading bases to be trimmed globally in every read or the overall minimum average quality of that a read must have for it not to be removed. A more extensive list of these modifiable parameters can be found in the **Program Manual** section. If the user does not explicitly assign values to these parameters, defaults are used.

4.3 Functions

In total, 7 functions are stored in the `trimmer_functions.py` module:

run_arg_parser()

Retrieves the user input from the command line arguments.

controling_output_file(input_file)

Makes sure that no files in the working directory will be overwritten.

print_read(ID, seq, qual_str, file)

Prints the processed reads onto the output file(s).

phred_autodetect(input_file, USER_PHRED)

Determines the Phred encoding type of a given fastq file. It finds the ASCII encoded quality strings and returns either '33', '64' or 'unknown'. To do so, it checks whether the strings are a subset of either of the two predefined sets: phred33 and phred64, which contain the allowed ASCII characters in the corresponding phred encoding type. The function will read only as many reads are necessary to find the encoding type of the file, and then it will stop.

The '33' or '64' returned by this function are stored in the main body of the program in the variable *phred*, which is used by the next function *quality_score(quality_str, phred)*.

Algorithm 4: *phred_autodetect(input_file, USER_PHRED)*

```
open input_file
phred33_set = set of Phred 33 characters
phred64_set = set of Phred 64 characters
for read in input_file do
    if encoded quality string IS ONLY subset of phred33 then
        return '33' and exit function
    else if encoded quality string IS ONLY subset of phred64 then
        return '64' and exit function
if phred type cannot be autodetected then
    if USER_PHRED IS NOT empty then
        return USER_PHRED
    else
        return error message and exit the full program
close input_file
```

quality_score(quality_str, phred)

This function decodes the quality string from ASCII to numeric. It relies on two dictionaries, phred33_dict and phred_64, that have the ASCII characters as keys and their corresponding numeric scores as values. It takes as variables the quality string of a read (*quality_string*) and the autodetected phred encoding type (*phred*).

When an unknown character is found in a read, 'unknown' is returned and the read is removed in the main body of the program.

Algorithm 5: *quality_score(quality_str, phred)*

```
phred33_dict = dict of phred33 charachters and corresponding quality scores
phred64_dict = dict of phred64 characters and corresponding quality scores
if unknown character is found in quality string then
    return 'unknown'
else
    decode using dictionary
    return decoded quality string
```

global_trim(DNA_str,quality_str, quality_score, LEADING, TRAILING)

This function removes bases from the 3' and 5' ends of reads according to user input, simply by slicing it.

quality_trim(read, qual_str, qual_score, WIN_SIZE, AVG_QUALITY, BASE_QUALITY)

This function removes bases from the 3' and 5' ends of reads if their quality is low. It works in two different ways: if WIN_SIZE = 1, it evaluates single bases and removes them if their quality is below BASE_QUALITY; if WIN_SIZE > 1, it uses the *mean of sliding window* approach, where the mean of n=WIN_SIZE end bases is calculated and the first base is removed if the mean of the window is below AVG_QUALITY.

Algorithm 6: *quality_trim(read, qual_str, qual_score, WIN_SIZE, AVG_QUALITY, BASE_QUALITY)*

```
if WIN_SIZE > 1 –SLIDING WINDOW APPROACH– then
    calculate mean quality of window of length = WIN_SIZE in 3' end
    while window mean < AVG_QUALITY do
        remove 3' end base
        calculate mean quality of new window of length = WIN_SIZE in 3' end
    repeat from 5' end
else if WIN_SIZE == 1 –SINGLE BASE APPROACH– then
    while quality of 3' end base < BASE_QUALITY do
        remove 3' end base
    repeat from 5' end
```

4.4 Error handling

These are the error scenarios that are handled in the program:

Handled Errors	
Problem Found	**Error Message**
Input files do not exist/cannot be opened	'File could not be opened. Reason: (IO error)'
Input files have the wrong extension	'Your input file(s) must have .fastq or .fastq.gz extension'
Output files already exist	'(filename) will be overwritten. Do you want to continue? y/n:'
Unequal number of reads in paired end mode	'Input files do not contain equal number of reads. Output cannot be trusted. Check your files.'
Settings input is unexpected	'Invalid input. Reason: (IO Error)'
Phred quality encoding type cannot be autodetected and no user input given	'We cannot autodetect the phred encoding type of your file(s). Please specify it in the input.'

These are some errors that are not currently handled by the program, which relies on proper file format:

Unhandled Errors (program relies on proper file format)	
Problem Found	**Error Message**
Wrong file format, despite correct extension	Raises unhandled errors
Unequal read and quality string length	No error but wrong output

5 Program manual

5.1 Settings

As mentioned before, a series of parameters can be specified by the user in order to fine-tune the trimmer. These can be found through the terminal typing `magicClipper.py -h`, which shows the following:

```
usage: magicClipper.py [-h] [-PH] [-L] [-T] [-W] [-AQ] [-BQ] [-ML] [-N]
                       File name 1 [File name 2]

---- THE magicClipper NGS READ TRIMMER ---- When given one or two .fastq
files, this program performs a complete user-guided and quality-based trimming
of the reads contained in it. If you like, you can adjust the settings listed
below. Otherwise, their defaults will be used.

positional arguments:
  File name 1           Your forward fastq file.
  File name 2           Your reverse fastq file (optional, only for paired end
                        mode).

optional arguments:
  -h, --help            show this help message and exit
  -PH , --PHRED         Phred encoding type (phred 33 or phred 64). The
                        encoding type will automatically be determined by the
                        program, and if user input does not match true type, a
                        warning will be printed onto the log file.
  -L , --LEADING        Number of bases to be trimmed from 3' end of all
                        reads, regardless of quality. Default is 0.
  -T , --TRAILING       Number of bases to be trimmed from 5' end of all
                        reads, regardless of quality. Default is 0.
  -W , --WINDOWSIZE     Window size for sliding window trimming approach.
                        Default is 4. If WINDOWSIZE is 1, then the single base
                        appoach is used and BASEQUALITY is taken to be the
                        quality threshold, instead of AVGQUALITY.
  -AQ , --AVGQUALITY    Average quality threshold for sliding window trimming
                        approach. Default is 15.
  -BQ , --BASEQUALITY   Quality threshold for single base trimming approach.
                        Default is 3.
  -ML , --MINLEN        Minimum allowed length of reads. Default is 36.
  -N , --MAXN           Maximum number of unknown bases allowed in a read.
                        Default is 15.
```

Figure 2: magicClipper help page. Adjustable settings and their defaults.

5.2 Example Runs

Single end mode

```
magicClipper.py BRISCOE_0069_BD18RUACXX_L2_1_pf.fastq.gz
```

This will perform the trimming in single end mode on file `BRISCOE_0069_BD18RUACXX_L2_1_pf.fastq.gz` with all the default settings:

- No leading global trim
- No trailing global trim
- Sliding window trimming approach with `WINDOWSIZE 4` and `AVGQUALITY 15`
- Read filtering with minimum read length `MINLEN 36`, average read quality threshold `AVGQUALITY 15` and maximum number of unknown bases `MAXN 15`

Since the input file is `gzipped`, so will the output file, whose name will be `BRISCOE_0069_BD18RUACXX_L2_1_pf_trimmed.fastq.gz`.

The log file will be found under `BRISCOE_0069_BD18RUACXX_L2_1_pf.log`.

Paired end mode

```
magicClipper.py BRISCOE_0069_BD18RUACXX_L2_1_pf_trimmed.fastq
BRISCOE_0069_BD18RUACXX_L2_2_pf.fastq -LEADING 3 -TRAILING 3 -WINDOWSIZE 1
-BASEQUALITY 5
```

This will perform the trimming in paired end mode on files `BRISCOE_0069_BD18RUACXX_L2_1_pf.fastq` and `BRISCOE_0069_BD18RUACXX_L2_2_pf.fastq` with the following settings:

- Leading global trim of `LEADING 3`
- Trailing global trim of `TRAILING 3`
- Single base trimming approach (because `WINDOWSIZE 1`) with `BASEQUALITY 5`
- Read filtering with default minimum read length `MINLEN 36`, average read quality threshold `AVGQUALITY 15` and maximum number of unknown bases `MAXN 15`

Since the input file is not `gzipped`, neither will the output files, whose names will be `BRISCOE_0069_BD18RUACXX_L2_1_pf_trimmed.fastq` and `BRISCOE_0069_BD18RUACXX_L2_2_pf_trimmed.fastq`

The log file will be found under `BRISCOE_0069_BD18RUACXX_L2_1_pf.log`.

5.3 Example Log File

With every run of the `magicClipper`, a log file with some information regarding the input, settings and output is created. Any errors or problems that were overcome by the program or that didn't cause it to stop can also be found in this file.

```
This is the log file for the trimming of BRISCOE_0069_BD18RUACXX_L2_1_pf.fastq

===============
SETTINGS
===============
File 1: BRISCOE_0069_BD18RUACXX_L2_2_pf.fastq
Base quality: 3
Average quality: 15
Lead trim: 0
Trail trim: 0
Window size: 4
Maximum unknown bases: 15
Min lenght: 36
Phred: 33

===============
STATS
===============
*** Before trimming ***
Total number of reads: 1000
Average length of reads: 101.0
Average quality of reads: 36.66

*** After trimming ***
Read pairs removed due to low quality/short length: 4
Read pairs kept: 996
Reads trimmed: 98
Average length of kept reads: 99.33
Average quality of kept reads: 37.29
```

Figure 3: Example of magicClipper log file, with input file(s), settings and statistics.

5.4 Example Standard Output

In the standard output, the user will receive some general messages about the process they have initiated and updates on how far along the trimmer is. In case of terminating errors or extra input required, the user will also receive those here.

```
bigO_1_trimmed.fastq.gz will be overwritten. Do you want to continue? y/n: y

You have initialized the single end mode of magicClipper.
This might take a while... So please be patient!

--- 100000 reads processed ---
--- 200000 reads processed ---
--- 300000 reads processed ---
--- 400000 reads processed ---

Congratulations! Your trimming was successful.
You can find your results in the bigfile_trimmed.fastq file and some additional
info in the the bigfile.log file.
Pleasure working with you!
```

Figure 4: Example of magicClipper standard output.

6 Run-time analysis

To estimate the complexity of our program we processed files of various sizes and measured the execution time for each run, the result can be seen in Figure 5. The result of the test shows that the execution time is increased linearly as the the size of the files increase. Considering this and the fact that the most complex statements in the program's code are linear (the line by line reading of the input file(s) and the iteration over each read's 4 element list), we can express the performance of `magicClipper` in BigO notation as O(N), where N is the total number of reads (or lines) in the input file(s).

It is worth noting that the runtime trend is consistent among the single and paired end modes, which shows that they both have the exact same dependence on the total number of reads that they are fed.

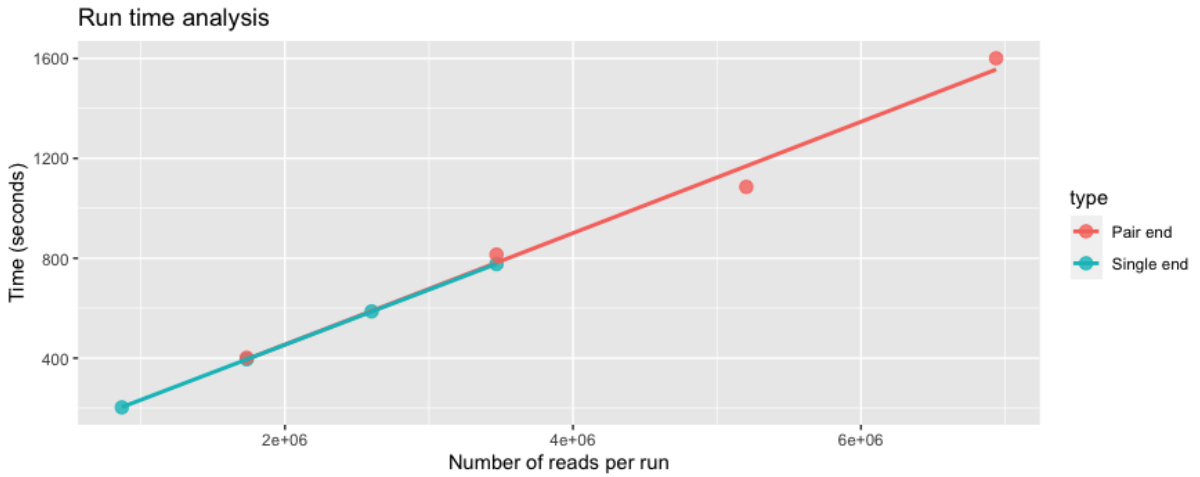

Figure 5: Run-time analysis showing how the execution time increases linearly with the size of the file(s). Created with R.

The fastq data is usually stored in compressed format and one way to possibly make the program run faster would be by opening the these files in binary mode. The data necessary for the data processing would have to be converted to text format, but the header (line 1) and the line for optional information (line 3) of the read could be kept in binary mode.

7 Conclusion

The NGS trimmer created in this project fulfills all the basic and advanced functionality criterion stated in the project description, but in what ways could our program be improved?

If compared to previously existing tools for trimming NGS reads like `Trimmomatic` [5], our program is quite slow. With a runtime of approximately 20 minutes in the paired end mode of a 2Gb (1+1) paired end piece of data, processing real size data (sometimes up to 100Gb) would take days. This is clearly not reasonable and shows that an overall optimization of the program is necessary. Reducing the amount of statements and condition evaluations, especially those inside of the main file-reading loop, would be a good place to start and has big potential to improve the program's performance. As mentioned in the **Run-time Analysis** section, working in binary mode could also be an option.

Regarding error handling, we believe we have covered most of the possible scenarios that could cause the program to break or, even worse, give wrong results without raising errors. However, as mentioned in the **Error Handling** section, it still relies on some assumptions that, if not met, prevent its proper functioning.

Looking at `Trimmomatic` also gives us ideas of other functionalities that could be implemented in `magicClipper`: removal of adapter sequences, more sophisticated quality and length-based trimming...

To sum up, even though there aspects of the code which could be improved, we are very satisfied with the final result and hope the reader is too!

References

[1] "An introduction to Next-Generation Sequencing Technology." https://www.illumina.com/content/dam/illumina-marketing/documents/products/illumina_sequencing_introduction.pdf. Accessed: 2020-11-23.

[2] "FASTQ files explained." https://support.illumina.com/bulletins/2016/04/fastq-files-explained.html. Accessed: 2020-11-29.

[3] "Diagnosing problems with phasing and pre-phasing on Illumina platforms." http://lab.loman.net/high-throughput%20sequencing/2013/11/21/diagnosing-problems-with-phasing-and-pre-phasing-on-illumina-platforms/. Accessed: 2020-11-29.

[4] "Trimmomatic Manual: V0.32." http://www.usadellab.org/cms/uploads/supplementary/Trimmomatic/TrimmomaticManual_V0.32.pdf. Accessed: 2020-11-29.

[5] B. U. Anthony M. Bolger, Marc Lohse, "Trimmomatic: a flexible trimmer for Illumina sequence data. [On]," *Bioinformatics*, 2014.