CLC **Assembly Cell**

USER MANUAL

User manual for
CLC Assembly Cell 5.2.1

Windows, macOS and Linux

June 1, 2021

This software is for research purposes only.

QIAGEN Aarhus
Silkeborgvej 2
Prismet
DK-8000 Aarhus C
Denmark

Contents

1 Introduction **8**

1.1 Overview of Commands . . . 8

1.1.1 Core analysis tools . . . 8

1.1.2 Reporting tools overview . . . 9

1.1.3 Assemblies post-processing tools . . . 9

1.1.4 Sequences preparation tools . . . 9

1.1.5 Format conversion . . . 10

1.2 Latest improvements . . . 10

2 System Requirements and Installation **11**

2.1 System requirements . . . 11

2.1.1 Limitations on maximum number of cores . . . 12

2.1.2 Supported CPU architectures . . . 12

2.2 Disk space . . . 12

2.3 Downloading and installing the software . . . 12

2.4 Installing Static license . . . 13

2.4.1 Licensing the software on a networked Linux or Mac machine . . . 13

2.4.2 Licensing the software on a networked Windows machine . . . 13

2.4.3 Licensing the software on a non-networked machine . . . 14

2.5 Network Licenses . . . 14

2.5.1 Installing and Running the CLC Network License Manager . . . 15

2.5.2 Configuring the software to use a network license . . . 15

2.6 Restricting CPU usage . . . 15

3 The Basics **16**

3.1 How to use the programs 16
3.1.1 Getting Help 16
3.1.2 A basic example 17
3.2 Input Files 17
3.3 Cas File Format 17
3.3.1 Cas Format Basics 18
3.3.2 What a cas file contains 18
3.3.3 What a cas file does not contain 18
3.3.4 Considerations and limitations 18
3.3.5 Converting to and from SAM and BAM formats 19
3.4 Paired read Considerations 19
3.4.1 Relative orientation of the reads 20
3.4.2 Measuring the distance between the reads 20
3.4.3 Paired Read File Input 20

4 Read Mapping **22**
4.1 References and indexes 22
4.2 Reads 24
4.2.1 Single reads 24
4.2.2 Paired reads 24
4.3 Alignment parameters 26
4.3.1 Match score 26
4.3.2 Mismatch cost 26
4.3.3 Linear gap cost 26
4.3.4 Affine gap cost 27
4.3.5 Alignment mode 27
4.4 Reporting and filtering 28
4.5 Quality filtering 29

5 De novo assembly **30**
5.1 De novo assembly inputs 30
5.2 De novo assembly outputs 30
5.3 How it works 31

5.3.1 Resolve repeats using reads 34
Remove weak edges 34
Remove dead ends 34
Resolve repeats without conflicts 35
Resolve repeats with conflicts 36
5.3.2 Automatic paired distance estimation 36
5.3.3 Optimization of the graph using paired reads 39
5.3.4 AGP export 40
5.3.5 Bubble resolution 41
5.3.6 Converting the graph to contig sequences 42
5.3.7 Summary 42
5.4 Randomness in the results 43
5.5 Specific characteristics of CLC algorithm 43
5.6 SOLiD data support in de novo assembly 44
5.7 Command line options 44
5.7.1 Specifying the data for the assembly and how it should be used 44
5.7.2 Specifying information for the assembly 46
5.7.3 Specifying information about the outputs 47
5.7.4 Other options 47
5.7.5 Example commands 48
5.8 New long read assembly tools (beta) 49
5.9 The clc_correct_pacbio_reads tool (beta) 49
5.9.1 Interlude: Converting PacBio's BAM to FASTA 50
5.9.2 Basic usage 50
5.9.3 Advanced parameters 50
5.10 The clc_assembler_long tool (beta) 50
5.10.1 Method 51
5.10.2 Input parameters 52
5.10.3 Graph construction 52
5.10.4 Graph post-processing 52
5.10.5 Contig post-processing 52
5.10.6 Output parameters 52

5.10.7 An example including error-correction for PacBio reads 53

6 Reporting tools **54**

6.1 The clc_sequence_info Program 54

6.2 The clc_mapping_table Program 55

6.3 The clc_mapping_info Program 57

7 Assembly post-processing tools **60**

7.1 The clc_change_cas_paths Program 60

7.2 The clc_filter_matches Program 61

7.3 The clc_extract_consensus Program 61

7.4 The clc_join_mappings Program 62

7.5 The clc_submapping Program 62

7.5.1 Specifying Mapping Files 62

7.5.2 Extracting a Subset of Reference Sequences 62

7.5.3 Extracting a Part of a Single Reference Sequence 62

7.5.4 Extracting Only Long Contigs (Useful for De Novo Assembly) 62

7.5.5 Extracting a Subset of Read Sequences 63

7.5.6 Other Match Restrictions 63

7.5.7 Output Reference File 63

7.5.8 Output Read File 63

7.5.9 Handling of non-specific matches 64

7.6 The clc_unmapped_reads Program 64

7.7 The clc_unpaired_reads Program 64

7.8 The clc_agp_join Program 64

8 Sequence preparation tools **65**

8.1 The clc_adapter_trim Program 65

8.2 Quality trimming 66

8.2.1 Fastq quality scoring 66

8.3 The clc_remove_duplicates Program 67

8.3.1 Example of duplicate read removal 69

8.4 The clc_sample_reads Program 70

8.5 The clc_sort_pairs Program 70

8.6 The clc_split_reads Program (for 454 paired data) 71

8.7 The clc_overlap_reads Program 72

9 Format conversion tools **74**

9.1 The clc_cas_to_sam Program 74

9.2 The clc_sam_to_cas Program 74

9.3 The clc_convert_sequences Program 75

A Options for All Programs **76**

B Third Party Libraries **77**

Bibliography **78**

Chapter 1

Introduction

Welcome to CLC Assembly Cell 5.2.1 – a software package supporting your daily bioinformatics work. This package includes command line tools for performing de novo assemblies, read mappings, and basic downstream analysis of the results of these analyses. CLC Assembly Cell also includes utility tools for certain types of data pre-processing and sequence format conversion.

1.1 Overview of Commands

There are many tools within the CLC Assembly Cell. We list the tools included briefly here, with chapters dedicated to details about the core tools, and then other categories of tools, following.

Full usage information for all tools are given in section A. The full usage information for each program can also be viewed by executing it without any options.

1.1.1 Core analysis tools

De novo assembly and mapping reads to reference sequences form the core tools of CLC Assembly Cell. These tools can be accessed using the following commands:

clc_assembler De novo assembly.

clc_mapper Used for mapping reads to a reference sequence.

clc_mapper_legacy The read mapper included in earlier versions of CLC Assembly Cell. This tool is for mapping sequencing reads from the SOLiD color space platform to a reference sequence.

The output of a de novo assembly is a set of contig sequences in fasta format.

The output of read mapping tools is a file in a special format called cas. The file extension for this file is .cas.

1.1.2 Reporting tools overview

The following commands are available for reporting information about cas assembly files as well as contig data:

clc_sequence_info Print overview of any sequence file.

clc_mapping_info Print overview of a mapping.

clc_mapping_table Print details of each read in a mapping.

1.1.3 Assemblies post-processing tools

Various operations can be performed on cas assembly files.

clc_change_cas_paths Change the references and / or read file names in a mapping file.

clc_extract_consensus Extract a consensus sequence and get zero coverage regions.

clc_filter_matches Remove matches of low similarity.

clc_join_mappings Join a number of assemblies to the same reference.

clc_submapping Extract a part of an assembly

clc_unmapped_reads Extract unassembled reads from an assembly.

clc_unpaired_reads Extract reads from broken pairs.

If more advanced downstream analyses of assemblies are desired, the CLC Genomics Workbench can be used (see https://digitalinsights.qiagen.com/products-overview/discovery-insights-portfolio/analysis-and-visualization/qiagen-clc-genomics-workbench/). The Workbench uses the same de novo assembly and read mapping algorithms as the CLC Assembly Cell, so these tools can be directly run in the CLC Genomics Workbench, or alternatively, read mapping or assembly outputs created using CLC Assembly Cellcan be imported into the Workbench. However, note that the CLC Genomics Workbench requires a separate license to the CLC Assembly Cell.

1.1.4 Sequences preparation tools

A range of different tools are available for sequence preparation.

clc_adapter_trim Trim adapters from sequences.

clc_quality_trim Trim reads based on quality.

clc_remove_duplicates Remove duplicate reads from genomic data.

clc_sample_reads Random sampling of reads.

clc_sort_pairs Split paired read files into paired and unpaired files.

clc_split_reads Remove linker from 454 paired data and extracts pairs.

clc_overlap_reads Merge overlapping reads.

1.1.5 Format conversion

clc_cas_to_sam For conversion of cas format mapping files to sam format.

clc_sam_to_cas For conversion of sam format mapping files to cas format.

clc_to_fasta Converts fastq, sff, csfasta and genbank format files into fasta.

1.2 Latest improvements

CLC Assembly Cell is constantly under development and a detailed list that includes a description of new features, improvements, bugfixes, and changes for the current version of CLC Assembly Cell can be found at:

https://digitalinsights.qiagen.com/products/qiagen-clc-assembly-cell/latest-improvements/current-line/

Chapter 2

System Requirements and Installation

2.1 System requirements

- Windows 7, Windows 8, Windows 10, Windows Server 2012, Windows Server 2016 and Windows Server 2019
- OS X 10.10, 10.11 and macOS 10.12, 10.13, 10.14
- Linux: RHEL 6.7 and later, SUSE Linux Enterprise Server 11 and later. The software is expected to run without problem on other recent Linux systems, but we do not guarantee this.
- 1024 x 768 display recommended
- Intel or AMD CPU required
- 64 bit computer and operating system
- **Special requirements for read mapping**. The numbers below give minimum and recommended memory for systems running mapping and analysis tasks. The requirements suggested are based on the genome size.
  - **E. coli K12 (4.6 megabases)**
    - Minimum: 500 MB RAM
    - Recommended: 1 GB RAM
  - **C. elegans (100 megabases)** and **Arabidopsis thaliana (120 megabases)**
    - Minimum: 1 GB RAM
    - Recommended: 2 GB RAM
  - **Zebrafish (1.5 gigabases)**
    - Minimum: 2 GB RAM
    - Recommended: 4 GB RAM
  - **Human (3.2 gigabases)** and **Mouse (2.7 gigabases)**
    - Minimum: 6 GB RAM
    - Recommended: 8 GB RAM

- **Special requirements for de novo assembly**. *De novo* assembly may need more memory than stated above - this depends both on the number of reads, error profile and the complexity and size of the genome. See `http://resources.qiagenbioinformatics.com/white-papers/White_paper_on_de_novo_assembly_4.pdf` for examples of the memory usage of various data sets.

2.1.1 Limitations on maximum number of cores

Most modern CPUs implements hyper threading or a similar technology which makes each physical CPU core appear as two logical cores on a system. In this manual the term "core" always refer to a logical core unless otherwise stated.

For static licenses, there is a limitation on the number of logical cores on the computer. If there are more than 64 logical cores, the CLC Assembly Cell cannot be started. In this case, a network license is needed (read more at `https://digitalinsights.qiagen.com/licensing/`).

2.1.2 Supported CPU architectures

Software from QIAGEN Aarhus is developed for and tested on the x86 and x86-64 CPU architectures, which are used in most Intel and AMD CPUs. PowerPC CPUs, such as those used in Apple products until 2006, are not supported. To run Assembly Cell the CPU must also support the SSE2 instruction set, which is commonly available in Intel CPUs produced from 2001 and onwards and AMD CPUs produced from 2003 and onwards.

If you are not sure if your CPU is supported, send a mail to ts-bioinformatics@qiagen.com with all available technical information about your computer.

2.2 Disk space

Data from Next-Generation sequencing machines naturally take up a lot of disk space. Besides the output files, the CLC Assembly Cell will sometimes write temporary files. These files will be written to the directory specified in the TMP variable on Windows and TMPDIR on Linux and Mac.

2.3 Downloading and installing the software

1. Download the distribution from:

   `https://digitalinsights.qiagen.com/downloads/product-downloads/`.

2. Unzip the zip-file and ensure that the resulting folder is placed in the desired final location on your computer.

There are two main types of license for the CLC Assembly Cell software: static and network. Static licenses are tied to the hardware to which they are downloaded. Network licenses are served using a separate piece of software, the *CLC Network License Manager*[1], and allow CLC Assembly Cell programs to run on any machine that can contact that software to obtain a license[2].

[1] Earlier versions of this software were named CLC License Server.

[2] For running the software on a computer cluster, the most common license type would be a network license, which would then allow you to submit jobs to any node of your computer cluster.

For obtaining an evaluation license, please follow the instructions in the static license section.

2.4 Installing Static license

For purchased licenses, please ensure you have your License Order ID available, preferably in a form you can copy and paste, before embarking on the instructions in this section.

If you have not previously done so, you can download a 15 day evaluation license using the same license tool that is used to download purchased licenses.

These instructions assume that the machine you have installed CLC Assembly Cell on is connected to the network and can access outside sites. If this is not the case, please see the section below.

2.4.1 Licensing the software on a networked Linux or Mac machine

1. On the command line, run the **clc_cell_licutil** tool that you will find inside the installation directory of CLC Assembly Cell.

   You will need to run this tool as a user that has permissions to write the license file that is downloaded into the **licenses** folder in the installation directory of CLC Assembly Cell. If your software is installed centrally, this may mean running the tool with sudo.

2. You will be prompted as to whether you wish to **Request an evaluation license** or **Download license using a License Order ID**.

   For an evaluation license, just choose that option. As long as you have not previously trialled the software on your machine, the evaluation license should be downloaded. You will see a message printed to screen about the expiry date of the evaluation license and where the license was downloaded to. You should now be able to trial the software.

3. If you have a License Order ID, please copy it and then paste it in at the prompt.

After a few moments, your license should be downloaded and a message will be written to screen saying that it was successfully downloaded and where it was saved.

2.4.2 Licensing the software on a networked Windows machine

1. Go to the Windows start menu and in the search box, type **cmd**.

2. Click on the **cmd.exe** tool which will launch the windows command prompt.

   You need to run this as a user that has permissions to write the license file that is downloaded into the **licenses** folder in the installation directory of the CLC Assembly Cell. If your software is installed centrally, this will likely mean right clicking on the **cmd.exe** option and choosing to **Run as administrator**.

3. Navigate to the installation folder of the CLC Assembly Cell and execute the **clc_cell_licutil.bat** script.

4. You will be prompted as to whether you wish to **Request an evaluation license** or **Download license using a License Order ID**.

For an evaluation license, just choose that option. As long as you have not previously trialled the software on your machine, the evaluation license should be downloaded. You will see a message printed to screen about the expiry date of the evaluation license and where the license was downloaded to. You should now be able to trial the software.

5. If you have a License Order ID, please copy it and then paste it in at the prompt.

After a few moments, your license should be downloaded and a message will be written to screen saying that it was successfully downloaded and where it was saved.

2.4.3 Licensing the software on a non-networked machine

Using the tool distributed with CLC Assembly Cell for downloading a static license, the license will be specific to the machine you download it to. For a machine unable to connect to an outside network, you can follow the steps below to get a license for the software:

1. Get the host id for the machine that CLC Assembly Cell is installed on. To do this, run the **clc_cell_licutil** tool as per the instructions in the Linux and Mac or Windows sections above. (You do not need administrator privileges for this.)
2. Copy the **Host ID(s)** information that is printed near the top of the output.
3. On a machine that is able to reach external sites, go to the webpage `https://secure.clcbio.com/LmxWSv3/GetLicenseFile`
4. Paste in your License Order ID and your host ID information, as well as a host name. The host name is not important, but we recommend it is something that allows you to recognise or identify the machine that's been licensed if needed.
5. Click on the **Save** button.
6. Move the license file onto the machine where CLC Assembly Cell is installed.
7. Save the license file in the folder called **licenses** in the installation directory of CLC Assembly Cell[3].

2.5 Network Licenses

Network licenses are made available to users of CLC Assembly Cell software by using a separate piece of sofware, the *CLC Network License Manager*. Previously that software was called the CLC License Server.

In general terms, you need to:

- Download, install and start up the *CLC Network License Manager* on a machine that is accessible to the machines that CLC Assembly Cell will be running on. This would generally be a machine that is left on, with the license manager running as a service.

[3]Locations for static license files supported in earlier versions of the CLC Assembly Cell can continue to be used. We recommend, however, that you choose to store your static license in the **licenses** folder in the installation directory, as this could help us in troubleshooting any licensing issues you may contact us about.

- Configure the license settings for copies of CLC Assembly Cell that will make use of the network licenses.

2.5.1 Installing and Running the CLC Network License Manager

How to install and run the *CLC Network License Manager* is described in its manual, available from `https://resources.qiagenbioinformatics.com/manuals/clcnetworklicensemanager/current/User_Manual.pdf`.

The *CLC Network License Manager* software can be downloaded from `https://digitalinsights.qiagen.com/products/clc-network-license-manager-direct-download/`

2.5.2 Configuring the software to use a network license

In order to make CLC Assembly Cell contact the license manager for a license, you need to create a text file called `license.properties` including the following information:

```
serverip=192.168.1.200
serverport=6200
useserver=true
```

The `serverip` and `serverport` should be edited to match your license manager set-up.

This text file then should be placed in the **licenses** folder of the installation area of CLC Assembly Cell.

Locations supported in earlier versions of CLC Assembly Cell can still be used, although we recommend the location above. The full list of locations is:

- in the **licenses** folder of the installation area of CLC Assembly Cell.
- in the *working directory*
- in /etc/clcbio/licenses on the executing machine, or
- in $HOME/.clcbio/licenses/ where $HOME is the home directory of the user executing the program.

2.6 Restricting CPU usage

De novo assembly and mapping programs will use all cores available on the system if the job is large enough to warrant this. Should you wish to limit the number of cores to be used by a particular analysis, '–cpus' option can be used to set the maximum.

This option is included in the full listing of options for the relevant programs.

Chapter 3

The Basics

The chapter covers the basics of command line use in CLC Assembly Cell, including data format considerations such as supported data formats, the cas assembly file format and conversion to other assembly formats. The chapter ends with an overview of paired data handling in CLC Assembly Cell.

3.1 How to use the programs

The CLC Assembly Cell consists of standard command line tools, where the tool name is provided, followed by any flags or parameters required. All input to the command, including designating input and output files, is done via parameter arguments.

General things to be aware of when setting up a CLC Assembly Cell command include:

- For programs where there are choices between fasta and fastq as output formats, the format that is output is determined based on the filename you specify in the command. For example, for the clc_remove_duplicates program, if you provide an output filename ending in .fq or .fastq, then the output format will be fastq. Otherwise, it will be fasta. Any program with this sort of behaviour should include information about the convention used in the usage information produced by running the command without any arguments.
- When providing paired data in two files, where one file contains one member of a pair and the other file contains the other member of a pair, you must include the -i flag in front of **each** input file. More information is provided about this later in this chapter, when paired data input is discussed, as well as in the chapters on read mappings and de novo assembly.
- When providing information about sequences, such as fragment lengths, also referred to as distances, for paired data, the parameter values you enter will apply to **all read files after that point in the command**, until the point in the command where new parameter values are provided. This is discussed further in the chapters on read mappings and de novo assembly.

3.1.1 Getting Help

This manual gives information about the tools included in CLC Assembly Cell.

Full usage information for each program is available in Appendix A of this user manual, and also by running any of the CLC Assembly Cell commands without any arguments. For the core programs, clc_mapper and clc_assembler, particular parameters are discussed in more detail within the chapters dedicated to those tools.

3.1.2 A basic example

A basic example of a CLC Assembly Cell command would be running the clc_unpaired_reads program. This program generates an output file of reads that are not paired within a given mapping. Here, we would need to specify the mapping to look at and the name of the output file. Below is an example of how such a command might look:

```
clc_unpaired_reads -a assembly.cas -o unmapped.fasta
```

3.2 Input Files

The formats in the following table are recognized as valid input formats by one or more of the CLC Assembly Cell tools. Note that not all listed formats are valid for data to be treated as sequence reads, and not all listed formats are valid for data to be treated as reference sequences in the case of read mappings.

Input file formats are automatically detected by the software through consideration of the file contents. The filename is irrelevant with regards to input format.

Format	Reads	References
Fasta	+	+
Fastq	+	-
csfasta*	+	-
Sff**	+	-
GenBank	-	+

*The full sequence of any read containing one or more . symbols, present in a .csfasta format file, will be converted to contain only N characters when used by or output by any of the Assembly Cell tools.

**Please note that paired 454 data needs to be pre-processed using the clc_split_reads program.

Read and reference data compressed using gzip is supported as input by the CLC Assembly Cell programs except for clc_remove_duplicates and clc_mapper_legacy.

Single reference sequences longer than 2gb ($2 \cdot 10^9$ bases) and reads longer than 100,000 bases are not supported.

3.3 Cas File Format

CLC Assembly Cell uses the cas file format for read mappings. It is a custom file format that caters to the demands of high throughput sequencing data, while being flexible enough to handle other sequence data types also. No deep knowledge of this file format is necessary to work with it, but some basics can aid in understanding what this format contains and how it can be used.

3.3.1 Cas Format Basics

The cas format is a binary format. This is space efficient, taking only approximately 8 bytes per read assembled to the human genome. So a cas file with 100 million Solexa reads, each with a length of 35 bases, assembled to the human genome would only take up about 800 MB.

3.3.2 What a cas file contains

In essence, cas format files contain data about the relationships between sequences in other files.

In particular, cas files contain the following information:

- General info such as: program that made the file, its version and its parameters.
- The file names for the reference sequences.
- The file names for the read sequences.
- Information about the reference sequences: their number, lengths, etc.
- The scoring scheme used when making the file.
- Information about each read:
  - Whether it matches anywhere.
  - Which reference sequence does it match to.
  - Alignment between the reference sequence and the read.
  - The number of places the read matches.
  - Whether the read is part of a pair.

3.3.3 What a cas file does not contain

Cas format files do **not** contain any sequence data. Rather than the sequence information itself, cas files contain the names of the corresponding read and reference sequence files. As sequence reads and references already exist, much space can be saved by not generating a second copy of them as part of the assembly output file.

3.3.4 Considerations and limitations

The cas file format is designed with high volume assembly data in mind. However, there are certain considerations that should be kept in mind:

1. There is a limit of one alignment position per read. In other words, a read matching in multiple locations can only be assigned to one of these locations within the cas file. This limitation is in place because when assembling short reads to a large genome, some reads may match hundreds of thousands of locations. Keeping track of all such alignments would be problematic.

2. If you are planning to send your assembly to someone else for viewing or further processing, you need to include your read and reference files in addition to the cas assembly file. This is because the cas file contains information about the assembly, and does not contain any sequence information.

3. If you are planning to send your assembly to someone else, they must put the read and reference files in the same relative location to the cas file, as you did when you ran the assembly. This is because the cas file stores relative file names, and these must match the location of the read and reference files when further processing is undertaken. Please note though that the program change_assembly_files can be used to change the file names and locations.

4. If you plan to convert your cas file to SAM or BAM format, which include read information, you need to have the read data used for your mapping, as well as the cas file, available when you run the clc_cas_to_sam program.

3.3.5 Converting to and from SAM and BAM formats

CLC Assembly Cell includes a tool called **clc_cas_to_sam** to convert a cas format file to the SAM[1] or BAM[2] format. Also included is a tool called **clc_sam_to_cas** that converts from SAM or BAM format into the cas format.

These tools are described in more detail in their own sections: section 9.1 and section 9.2.

3.4 Paired read Considerations

You can specify that a read file came from a paired sequencing experiment using the '-p' option. This option is described in detail here, as well as within the read mapping and de novo assembly sections of the manual.

A typical set of information one would provide after the -p flag would look like this

```
-p fb ss 100 200
```

. The meaning of this would be:

- **fb** Specifies the relative orientation of the reads. Here, the first read of the pair is in the forward direction, the second read is in the backward, or reverse, orientation. The allowed values for this are provided below.

- **ss** Specifies the way the distances between the pair members should be measured. Here, the distances are given from the start (5 prime end) of the first read to the start (5 prime end) of the second read. Here, since the relative orientation is set to fb, the second read is reversed, so indicating ss means that the distance specified will include both the read lengths as well as the length of the sequence between the reads.

- **100 200** The range of distances expected between the specified start positions. Here this is between 100 and 200 bases.

[1]Sequence Alignment/Map format
[2]BAM is the binary compact format for SAM

3.4.1 Relative orientation of the reads

For all codes, it is possible to assemble the pair to any of the two reference sequence strands, so ‘ff’ may mean that both reads are placed in the forward direction or that both reads are placed in the reverse direction. There is still a difference between ‘ff’ and ‘bb’, though. For ‘bb’, the second read is effectively placed before the first read. The ’bb’ option is not widely used and is included for the sake of completeness.

The allowed values for the directions and their meanings are summarized in the table below.

	Read		
Code	First	Second	Description
ff	$\rightarrow$	$\rightarrow$	Both reads are forward.
fb	$\rightarrow$	$\leftarrow$	Reads point toward each other.
bf	$\leftarrow$	$\rightarrow$	Reads point away from each other.
bb	$\leftarrow$	$\leftarrow$	Both reads are backward.

3.4.2 Measuring the distance between the reads

How the distance between the reads should be measured depends on how the sequencing experiment is done. If the reads are sequenced in the upstream to downstream direction, the start of the reads is where the distance should be measured. This is indicated by the ‘ss’ code for start to start. The allowed values are ‘ss’, ‘se’, ‘es’, and ‘ee’, where the first letter indicates which end of the first read should be used and the second letter indicates which end of the second read should be used (‘s’ for start and ‘e’ for end). The ‘ss’ option is the most typical.

So, for typical paired end Illumina sequencing protocol, using the ‘fb ss’ combination ensures the correct relative directions of the reads. It also ensures that the distance is independent of the read length since typical sequencing experiment progress expands the reads toward each other from their starting points.

When the ‘-p’ option is used, it applies to all read files from that point and forward in the command line. If different experiments with different paired properties are combined, the ‘-p’ option can be used several times. To indicate that the following read files are not paired, used ‘-p no’. This is only necessary if another ‘-p’ option was previously used. An example:

```
clc_mapper -o assembly.cas -d human.gb -q reads1.fasta -p fb ss 180
                                              250 reads2.fasta -p no reads3.fasta
```

Here, we have three read files, where reads1.fasta and reads3.fasta are unpaired, while reads2.fasta are paired reads.

Note that the clc_sort_pairs and clc_split_reads program can be used to convert data from SOLiD and 454 systems, respectively, into an format accepted by the CLC Assembly Cell tools.

3.4.3 Paired Read File Input

Paired data may be contained in a single file, where the pairs are sorted such that the first two sequences are one pair, the second two sequences the next pair, and so on. Paired data may also exist in two files, with one file containing the first member of all pairs, and the other file containing the second member of all pairs, with each member appearing in the same ordered position in each file. For example, the 51st sequence in file A is the mate of the 51st sequence in file B.

The CLC Assembly Cell programs assume the single file form for paired data as the default. For paired data with separate files for first and second members of the pair, both files need to be included as input, with each of these files being preceeded by the '-i' option (for interleave). The order of the files on the command line matters. The first file should contain the first member of the pair. The second file should contain the second member of the pair.

To further illustrate this, consider a situation where we have two fasta files like this (first.fasta):

```
>pair_1/1
ACTGTCTAGCTACTGCATTGACTGCGAC
>pair_2/1
TAGCGACGATGCTACTACTCTACTCGAC
>pair_3/1
GATCTCTAGGACTACGCTACGAGCCTCA
```

and this (second.fasta):

```
>pair_1/2
GGATCATCTACGTCATCGACTAGTACAC
>pair_2/2
AAGCGACACCTACTCATCGATCATCAGA
>pair_3/2
TATCGACTCAGACACTCTATACTACCAT
```

where `pair_1/1` and `pair_1/2` belong together, `pair_2/1` and `pair_2/2` belong together, etc. The programs expect to see these sequences as one fasta file like this (joint.fasta):

```
>pair_1/1
ACTGTCTAGCTACTGCATTGACTGCGAC
>pair_1/2
GGATCATCTACGTCATCGACTAGTACAC
>pair_2/1
TAGCGACGATGCTACTACTCTACTCGAC
>pair_2/2
AAGCGACACCTACTCATCGATCATCAGA
>pair_3/1
GATCTCTAGGACTACGCTACGAGCCTCA
>pair_3/2
TATCGACTCAGACACTCTATACTACCAT
```

This is accomplished using the '-i' option like this:

```
clc_mapper -o assembly.cas -d human.gb -q -p fb ss 180 250
                                                 -i first.fasta second.fasta
```

This is identical to:

```
clc_mapper -o assembly.cas -d human.gb -q -p fb ss 180 250 joint.fasta
```

Note that the '-i' option has to immediately proceed the input files.

Chapter 4

Read Mapping

The *read mapper*, `clc_mapper`, maps a list of sequencing reads to a set of reference sequences, collectively referred to as *the reference genome*.

For each sequencing read, the read mapper reports the location(s) in the reference genome where that read is most likely to have originated from. The reported location is the result of this procedure:

1. A search is carried out for the longest stretches of matching bases between the reference genome and a read by considering each base position of the read as a start position of a seed candidate.
2. End-positions of seeds are then determined by elongating the seeds as long as there are fully matching sequences in the reference index.
3. Seeds are reduced down to 2/3 of the length of the longest one.
4. Finally, the seeds are examined in detail using a banded Smith-Waterman algorithm. Seeds from paired reads are examined together.

The seed lengths in this mapping tool are variable, but have a minimum size of 15bp. The variable seed length enables identification of short seeds where the alignment score is higher than the alignment score for longer seeds. This leads to a better mapping of some reads, and improves the chance of identifying the optimal mapping, especially for reads with high error rates.

4.1 References and indexes

The read mapper supports mapping against a mixture of linear and circular reference sequences.

Reference sequences are typically provided in the form of one or more FASTA files[1], that can be passed to the mapper using the `-d/--references` parameter. Files containing sequences, to be interpreted as circular, must be *individually* prefixed with the `-z/--circular` parameter:

```
clc_mapper -d linear.fa -z circular.fa linear_again.fa ...
```

[1]FASTQ is also supported.

Internally, the provided sequences are converted into a *reference index*, allowing the read mapper to efficiently search and navigate them.

If you find yourself working repeatedly with the same large genomes, such as human, you can save a lot of time by writing the index to disk using the `-n/--indexoutput` parameter, eg.

```
clc_mapper -d chr1.fa chr2.fa ... chrY.fa -z mito.fa -n human.idx ...
```

The next time you need to use that human reference, simply provide it as a reference:

```
clc_mapper -d human.idx ...
```

Note, that the size of the index, both on disk and in memory, is comparable to the cumulative size of the reference sequences in FASTA format. It typically uses slightly less than one byte per base. For example, a human index needs around 2.8 gigabytes of space.

4.2 Reads

The read mapper supports mapping both single and paired reads.

Reads are typically provided in the form of one or more FASTA or FASTQ files, that are passed to the mapper using the `-q/--reads` parameter.

4.2.1 Single reads

In the case of single reads, you simply list the files after the `-q/--reads` parameter:

```
clc_mapper -q single_reads.fastq more_single_reads.fasta ...
```

In the special case of PacBio reads, you can optimize both quality and speed of the alignment process by providing type information as follows:

```
clc_mapper -q --type pacbio long_pacbio_reads.fastq ...
```

The type, as defined by the `--type` parameter, applies to all following files.

4.2.2 Paired reads

Paired reads often come split across two files: one containing the first mates, `paired_1.fa`:

```
>SLXA-EAS1_89:1:1:672:654/1
GCTACGGAATAAAACCAGGAACAACAGACCCAGCA
>SLXA-EAS1_89:1:1:657:649/1
GCAGAAAATGGGAGTGAAAATCTCCGATGAGCAGC
>SLXA-EAS1_89:1:1:708:653/1
GAGAGAGCAGTGGGCGAGGTTGGGACATGTCATGA
```

and one containing the second mates, `paired_2.fa`:

```
>SLXA-EAS1_89:1:1:672:654/2
ATTAACAACAAAGGGTAAAAGGCATCATGGCTTCA
>SLXA-EAS1_89:1:1:657:649/2
TGATGCGACGACGCACCTCGTTGTTACGCACTTCA
>SLXA-EAS1_89:1:1:708:653/2
CTGTGGATAACATGGTGTAAGATCCTGTTTATTTT
```

Use the `-p/--paired` parameter to indicate that the following files contain paired reads of a certain type, and use the `-i/--interleave` parameter to pair up the two files:

```
clc_mapper -q -p fb ss 50 500 -i paired_1.fa paired_2.fa ...
```

Behind the scenes, the pairs are interleaved and processed together:

If you have a paired file, that is already interleaved, you can leave out the `-i/--interleave` parameter:

```
>SLXA-EAS1_89:1:1:672:654/1
GCTACGGAATAAAACCAGGAACAACAGACCCAGCA
>SLXA-EAS1_89:1:1:672:654/2
ATTAACAACAAAGGGTAAAAGGCATCATGGCTTCA
>SLXA-EAS1_89:1:1:657:649/1
GCAGAAAATGGGAGTGAAAATCTCCGATGAGCAGC
>SLXA-EAS1_89:1:1:657:649/2
TGATGCGACGACGCACCTCGTTGTTACGCACTTCA
```

```
clc_mapper -q -p fb ss 50 500 interleaved_pairs.fastq ...
```

For detailed information about the `-p fb ss 50 500` parameter, please refer to section 3.4.

4.3 Alignment parameters

The read mapper aligns the reads according to a user-specified *scoring scheme*, that expresses your expectation of what the data looks like and guides the mapper in deciding whether a particular alignment is good or bad.

As a rule of thumb, you want to choose a scoring scheme that matches the expected characteristics of the data. For example, if you are using a sequencing technology that has a tendency to produce errors in the form of gaps, you may want to lower the gap cost slightly to indicate, that gaps are expected to occur frequently.

Keep in mind, that the mapper deals with reads individually and so cannot distinguish between biological events and technical errors. Since technical errors are (typically) much more likely to occur than true genetic variation, you should set your costs according to what you expect to see in the raw reads, rather than which biological events you expect to be present in the sample itself.

4.3.1 Match score

The `-m/--matchscore` parameter determines the score attributed to a pair of matching bases in an alignment (default is 1).

4.3.2 Mismatch cost

The `-x/--mismatchcost` parameter determines the cost incurred when a mismatch is accepted (default is 2).

4.3.3 Linear gap cost

If each base inserted or deleted is taxed independently, we have what is known as a *linear gap cost* model, because the total cost of a gap $c(\lambda)$ is proportional to the length of the gap λ: $c(\lambda) = g\lambda$

where the cost per base g is set using the `-g/--gapcost` parameter. By default, it is set to 3, but you may want to lower that, if you are mapping data from sequencing technology that is prone to produce gaps, such as Ion Torrent, PacBio, or Oxford Nanopore.

If you would like to set the insertion and deletion costs differently, you can do so using a combination of the `-g/--gapcost` and the `-e/--deletioncost` parameters.

The `-g/--gapcost` parameter sets both the insertion and the deletion cost, while the `-e/--deletioncost` sets deletion cost only. For example, if you would like to set insertion cost to 1 and the deletion cost to 3, you would do the following:

```
clc_mapper ... -g 1 -e 3 ...
```

i.e. first set both the deletion and insertion cost to 1 and the set deletion cost back to 3.

4.3.4 Affine gap cost

If you believe that your data contains relatively few gaps, though they may be quite long, you can use the `-G/--gapopen` parameter to introduce an additional penalty for opening the gap in the first place. This would typically be used along with a low per-base gap cost, typically 1.

In this scheme the total cost $c(\lambda)$ of a gap of length λ is given by the following formula:

$$c(\lambda) = G + g\lambda \tag{4.1}$$

where the open cost G and the extension cost g are specified using `-G` and `-g`, respectively. Note that the affine model reduces to the linear one, when the open cost, G, is set to zero.

Using a combination of a relatively high open cost and a low extension (per-base) gap cost, indicates to the mapper that you expect few gaps, but that these may be quite long.

You should be careful about using high open costs, if you have lots of sequencing errors in the form of deletions or insertions, as these may be penalized to the point, where you get lots of long unaligned ends. In such cases linear gap cost may be a better choice.

Like linear gap costs, affine gap costs can be used asymmetrically. This is done by independently setting the deletion open cost using the `-E/--deletionopen` parameter.

4.3.5 Alignment mode

The mapper supports two different modes of alignment: *local* (default) and *global*.

The alignment mode is set using the `-a/--alignmode` parameter.

In local mode, `-a local`, alignments are chosen to be locally optimal, i.e. the mapper looks for the highest scoring alignment of *part of* the read. That is, unaligned ends are not penalized in any way.

In global mode[2], `-a global`, the mapper is forced to look for the highest scoring alignment of the entire read. That is, unaligned ends are no longer free.

Generally, we recommend that you stick to local alignment, as there are several reasons, why you may not want to force the ends of a raw read sequence to align:

1. Read quality deteriorates towards the end of a read, so you may end up aligning noise.
2. If present, untrimmed adapter sequence cannot be aligned in any meaningful way.
3. Structural variation may be present in the sample, so the reference may not have corresponding sequence for all of the read.

In general, unaligned ends express a lack of confidence in a portion of the read; information that can be very helpful for several, common downstream analyses.

[2]This is sometimes called semi-global alignment, because the alignment is global with respect to the read, but remains local with respect to the reference.

4.4 Reporting and filtering

The resulting alignments are output to a single CAS file[3] using the `-o/--output` parameter:

```
clc_mapper -d reference.fa -q reads.fastq -o result.cas
```

By default, if there are multiple, equally good alignments for a particular read, the number of alignments is stored, but detailed alignment information is only stored explicitly for one of them. Should you wish to change this behavior, you can use the following pair of parameters to do so:

The `-t/--maxalign` parameter determines the maximum number of alignments that are to be stored explicitly. By default, only one is stored explicitly.

The `-r/--repeat` parameter accepts one of two arguments: `random` or `ignore`. By default, it is set to `random`, which means that the explicitly stored alignments are sampled, randomly, from the total set of alignments found for a given read. The `ignore` argument results in reads with more alignments than can be explicitly stored are ignored and reported as unmapped.

As an example, suppose that you only want to report unique hits as mapped, i.e. ignore alignments with more than one hit. Then you can use the following combination:

```
clc_mapper ... -t 1 -r ignore ...
```

[3]The CAS file is a compact, binary file containing information about the alignment(s) of the individual reads. The read and reference sequences are not stored explicitly, but rather the paths to the input files are stored.

4.5 Quality filtering

Once an alignment has been found, it is examined by *the quality filter*. If it fails to meet the criteria enforced by the filter, it is discarded. If all alignments of a particular read fail to meet the criteria, the read is reported as unmapped.

The quality filter has two parameters: `-s/--similarity` and `-l/--lengthfraction`. They interact and must be understood together.

```
CTAACCACCT         CTAACCACCT
||||  ||||         ||||  ||||
CTAAGGACCT         CTAAGGACCT
s=0.8                s=0.8

GATTACA         GATTACA
||| |||         ||| |||
GATCACA         GATCACA
s=0.75          s=0.86
```

For example, the default setting (similarity=0.8, length fraction=0.5) means that a read will be reported as unmapped, if none of its alignments feature a stretch, at least 50% of the total alignment length[4], consisting of at least 80% matches.

If you want to see all alignments found, you can disable the filter altogether by specifying `-s 0`.

[4]By *total alignment length* we mean the combined length of all matches, mismatches, insertions, and deletions.

Chapter 5

De novo assembly

The clc_assembler program performs assembly of reads without a known reference. The input data consists of files containing read sequences.

5.1 De novo assembly inputs

Any number of read files can be input to a de novo assembly. These includes files containing paired reads and files containing single reads. Different types of read data can be input to a single de novo assembly. Below is a table of the accepted formats for data input.

Format	option	
Fasta	+	+
Fastq	+	-
csfasta	+	-
Sff *	+	-
GenBank	-	+

* Please note that paired 454 data needs to be pre-processed using the clc_split_reads program

5.2 De novo assembly outputs

The output of the `clc_assembler` is a fasta file containing all the contig sequences.

This means that there is no information about where the reads are placed, how they align, coverage levels etc. If this information is desired, you can use the clc_mapper or clc_mapper_legacy program and use the newly created contig sequences as references. The cas format file created using the mapping program will contain this sort of information.

If the -f option has been used, then a file containg features related to scaffolding will be generated. Choosing to name the file given as an argument to the -f option with a .agp suffix will generate an AGP format file. This format specification can be found online https://www.ncbi.nlm.nih.gov/projects/genome/assembly/agp/AGP_Specification.shtml.

Choosing to name the file given as an argument to the -f option with a .gff suffix will generate a

GFF format file. The columns of this file contain the following information:

Column 1: Name of contig Column 2: Source program Column 3: Annotation type (see below) Column 4: Start position Column 5: End position Column 6: Score (see below) Column 7, 8 and 9: no meaning: there to conform to the GFF format.

Further details about Annotation types (column 3)

There are three annotation types that can appear in the third column:

1) Alternatives Excluded: More than one path through the graph was possible in this region but evidence from paired data suggested the exclusion of one or more alternative routes in favor of the route chosen.

2) Contigs Joined: More than one route was possible through the graph such that an unambiguous choice of how to traverse the graph cannot by made. However evidence from paired data supports one of these routes and on this basis, this route is followed to the exclusion of the other(s).

3) Scaffold: The route through the graph is not clear but evidence from paired data supports the connection of two contigs. A single contig is then reported with N characters between the two connected regions. This entity is also known as a scaffold. The number of N characters represents the expected distance between the regions, based on the evidence the paired data.

If one chooses not to scaffold, a resulting gff annotation file will still report any "Contigs joined" and "Alternatives excluded" optimizations, as these are still performed in this case.

Further details about Scores (column 6)

For annotation type Scaffold, the size of the gap that has been estimated between scaffolded sections of the contig is reported in the score column.

For annotation type Alternatives Excluded, the score is reported as the (word size + 1). This value merely serves as a reminder that the region reported for this event is associated with the word size used for the assembly.

For annotation type Contigs Joined, the value in the score column is 0.

5.3 How it works

Our de novo assembly algorithm works by using de Bruijn graphs. This is a common approach used with de novo assembly algorithms described in Zerbino and Birney, 2008, Zerbino et al., 2009, Li et al., 2010, Gnerre et al., 2011. The basic idea is to make a table of all sub-sequences of a certain length (called words) found in the reads. The words are relatively short, e.g. about 20 for small data sets and 27 for a large data set. In the following section, we use the term word size to denote the length of a word. The word size is by default determined automatically (see explanation below).

Given a word in the table, we can look up all the potential neighboring words (in all the examples here, word of length 16 are used) as shown in figure 5.1.

Typically, only one of the backward neighbors and one of the forward neighbors will be present in the table. A graph can then be made where each node is a word that is present in the table and edges connect nodes that are neighbors. This is called a de Bruijn graph.

For genomic regions without repeats or sequencing errors, we get long linear stretches of

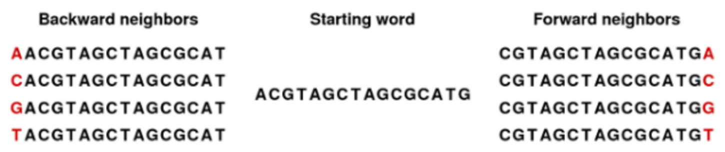


Figure 5.1: *The word in the middle is 16 bases long, and it shares the 15 first bases with the backward neighboring word and the last 15 bases with the forward neighboring word.*

connected nodes. We may choose to reduce such stretches of nodes with only one backward and one forward neighbor into nodes representing sub-sequences longer than the initial words.

Figure 5.2 shows an example where one node has two forward neighbors:

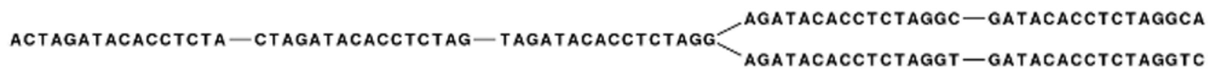


Figure 5.2: *Three nodes connected, each sharing 15 bases with its neighboring node and ending with two forward neighbors.*

After reduction, the three first nodes are merged, and the two sets of forward neighboring nodes are also merged as shown in figure 5.3.

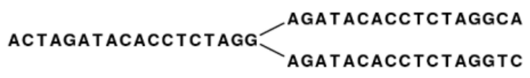


Figure 5.3: *The five nodes are compacted into three. Note that the first node is now 18 bases and the second nodes are each 17 bases.*

So bifurcations in the graph leads to separate nodes. In this case we get a total of three nodes after the reduction. Note that neighboring nodes still have an overlap (in this case 15 nucleotides since the word size is 16).

Given this way of representing the de Bruijn graph for the reads, we can consider some different situations:

When we have a *SNP or a sequencing error*, we get a so-called bubble (this is explained in detail in section 5.3.5) as shown in figure 5.4.

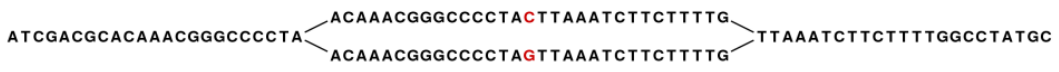

ATCGACGCACAAACGGGCCCCTA
ACAAACGGGCCCCTACTTAAATCTTCTTTTG
ACAAACGGGCCCCTAGTTAAATCTTCTTTTG
TTAAATCTTCTTTTGGCCTATGC

Figure 5.4: *A bubble caused by a heterozygous SNP or a sequencing error.*

Here, the central position may be either a C or a G. If this was a sequencing error occurring only once, it would be represented in the bubble as a path that is associated with a word that only occurs a single time'. On the other hand if this was a heterozygous SNP we would see both paths represented more or less equally in terms of the number of words that support each path. Thus, having information about how many times this particular word is seen in all the reads is very useful and this information is stored in the initial word table together with the words.

The most difficult problem for de novo assembly is repeats. Repeat regions in large genomes often get very complex: a repeat may be found thousands of times and part of one repeat may also be part of another repeat. Sometimes a repeat is longer than the read length (or the paired distance when pairs are available) and then it becomes impossible to resolve the length of the repeat. This is simply because there is no information available about how to connect the nodes before the repeat to the nodes after the repeat, and we just do not know how long the repeat is.

In the simple example, if we have a *repeat sequence* that is present twice in the genome, we would get a graph as shown in figure 5.5.

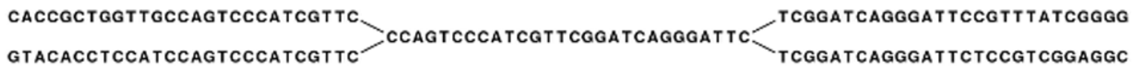

CACCGCTGGTTGCCAGTCCCATCGTTC
GTACACCTCCATCCAGTCCCATCGTTC
CCAGTCCCATCGTTCGGATCAGGGATTC
TCGGATCAGGGATTCCGTTTATCGGGG
TCGGATCAGGGATTCTCCGTCGGAGGC

Figure 5.5: *The central node represents the repeat region that is represented twice in the genome. The neighboring nodes represent the flanking regions of this repeat in the genome.*

Note that this repeat is 57 nucleotides long (the length of the sub-sequence in the central node above plus regions into the neighboring nodes where the sequences are identical). If the repeat had been shorter than 15 nucleotides, it would not have shown up as a repeat at all since the word size is 16. This is an argument for using long words in the word table. On the other hand, the longer the word, the more words from a read are affected by a sequencing error. Also, for each increment in the word size, we get one less word from each read. This is in particular an issue for very short reads. For example, if the read length is 35, we get 16 words out of each read if the word size is 20. If the word size is 25, we get only 11 words from each read.

To strike a balance, our de novo assembler chooses a word size based on the amount of input data: the more data, the longer the word length. It is based on the following:

```
word size 12: 0 bp - 30000 bp
word size 13: 30001 bp - 90002 bp
word size 14: 90003 bp - 270008 bp
word size 15: 270009 bp - 810026 bp
word size 16: 810027 bp - 2430080 bp
word size 17: 2430081 bp - 7290242 bp
word size 18: 7290243 bp - 21870728 bp
word size 19: 21870729 bp - 65612186 bp
word size 20: 65612187 bp - 196836560 bp
word size 21: 196836561 bp - 590509682 bp
word size 22: 590509683 bp - 1771529048 bp
word size 23: 1771529049 bp - 5314587146 bp
word size 24: 5314587147 bp - 15943761440 bp
word size 25: 15943761441 bp - 47831284322 bp
word size 26: 47831284323 bp - 143493852968 bp
word size 27: 143493852969 bp - 430481558906 bp
word size 28: 430481558907 bp - 1291444676720 bp
word size 29: 1291444676721 bp - 3874334030162 bp
word size 30: 3874334030163 bp - 11623002090488 bp
etc.
```

This pattern (multiplying by 3) continues until word size of 64 which is the max. The word size can also be specified manually using the `-w` option. Using the `-v` (verbose) option, you can see the word size that is automatically calculated by the assembler

5.3.1 Resolve repeats using reads

Having build the de Bruijn graph using words, our de novo assembler removes repeats and errors using reads. This is done in the following order:

- Remove weak edges
- Remove dead ends
- Resolve repeats using reads without conflicts
- Resolve repeats with conflicts
- Remove weak edges
- Remove dead ends

Each phase will be explained in the following subsections.

Remove weak edges

The de Bruijn graph is expected to contain artifacts from errors in the data. The number of reads agreeing upon an error is likely to be low especially compared to the number of reads without errors for the same region. When this relative difference is large enough, it's possible to conclude something is an error.

In the remove weak edges phase we consider each node and calculate the number c_1 of edges connected to the node and the number of times k_1 a read is passing through these edges. An average of reads going through an edge is calculated $avg_1 = k_1/c_1$ and then the process is repeated using only those edges which have more than or equal avg_1 reads going though it. Let c_2 be the number of edges which meet this requirement and k_2 the number of reads passing through these edges. A second average $avg_2 = k_2/c_2$ is used to calculate a limit,

$$limit = \frac{\log(avg_2)}{2} + \frac{avg_2}{40}$$

and each edge connected to the node which has less than or equal $limit$ number of reads passing through it will be removed in this phase.

Remove dead ends

Some read errors might occur more often than expected, either by chance or because they are systematic sequencing errors. These are not removed by the "Remove weak edges" phase and will cause "dead ends" to occur in the graph, which are short paths in the graph that terminate after a few nodes. Furthermore, the "Remove weak edges" sometimes only removes a part of the graph, which will also leave dead ends behind. Dead ends are identified by searching for paths in the graph where there exits an alternative path containing four times more nucleotides. All nodes in such paths are then removed in this step.

Resolve repeats without conflicts

Repeats and other shared regions between the reads lead to ambiguities in the graph. These must be resolved otherwise the region will be output as multiple contigs, one for each node in the region.

The algorithm for resolving repeats without conflicts considers a number of nodes called the *window*. To start with, a window only contains one node, say *R*. We also define the *border nodes* as the nodes outside the window connected to a node in the window. The idea is to divide the border nodes into sets such that border nodes *A* and *C* are in the same set if there is a read going through *A*, through nodes in the window and then through *C*. If there are strictly more than one of these sets we can resolve the repeat area, otherwise we expand the window.

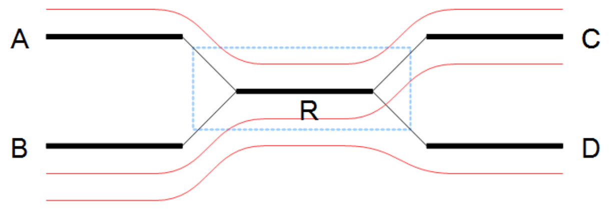


Figure 5.6: *A set of nodes.*

In the example in figure 5.6 all border nodes *A*, *B*, *C* and *D* are in the same set since one can reach every border nodes using reads (shown as red lines). Therefore we expand the window and in this case add node *C* to the window as shown in figure 5.7.

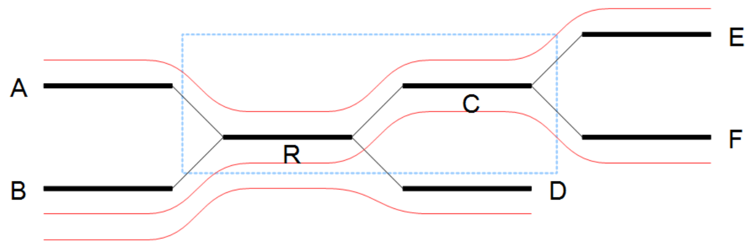


Figure 5.7: *Expanding the window to include more nodes.*

After the expansion of the window, the border nodes will be grouped into two groups being set *A*, *E* and set *B*, *D*, *F*. Since we have strictly more than one set, the repeat is resolved by copying the nodes and edges used by the reads which created the set. In the example the resolved repeat is shown in figure 5.8.

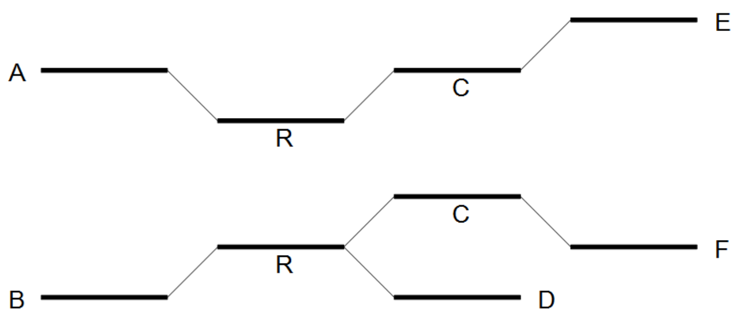


Figure 5.8: *Resolving the repeat.*

The algorithm for resolving repeats without conflict can be described the following way:

1. A node is selected as the window
2. The border is divided into sets using reads going through the window. If we have multiple sets, the repeat is resolved.
3. If the repeat cannot be resolved, we expand the window with nodes if possible and go to step 2.

The above steps are performed for every node.

Resolve repeats with conflicts

In the previous section repeats were resolved without excluding any reads that goes through the window. While this lead to a simpler graph, the graph will still contain artifacts, which have to be removed. The next phase removes most of these errors and is similar to the previous phase:

1. A node is selected as the initial window
2. The border is divided into sets using reads going through the window. If we have multiple sets, the repeat is resolved.
3. If the repeat cannot be resolved, the border nodes are divided into sets using reads going through the window where reads containing errors are excluded. If we have multiple sets, the repeat is resolved.
4. The window is expanded with nodes if possible and step 2 is repeated.

The algorithm described above is similar to the algorithm used in the previous section, except step 3 where the reads with errors are excluded. This is done by calculating an average $avg_1 = m_1/c_1$ where m_1 is the number of reads going through the window and c_1 is the number of distinct pairs of border nodes having one (or more) of these reads connecting them. A second average $avg_2 = m_2/c_2$ is calculated where m_2 is the number of reads going through the window having at least avg_1 or more reads connecting their border nodes and c_2 the number of distinct pairs of border nodes having avg_1 or more reads connecting them. Then, a read between two border nodes *B* and *C* is excluded if the number of reads going through *B* and *C* is less than or equal to $limit$ given by

$$limit = \frac{\log(avg_2)}{2} + \frac{avg_2}{16}$$

An example where we resolve a repeat with conflicts is given in 5.9 where we have a total of 21 reads going through the window with $avg_1 = 21/3 = 7$, $avg_2 = 20/2 = 10$ and $limit = 1/2 + 10/16 = 1.125$. Therefore all reads between border nodes *B* and *C* are excluded resulting in two sets of border nodes *A*, *C* and *B*, *D*. The resolved repeat is shown in figure 5.10.

5.3.2 Automatic paired distance estimation

The default behavior of the de novo assembler is to use the paired distances provided by the user. If the automatic paired distance estimation is enabled, the assembler will attempt to estimate the distance between paired reads. This is done by analysing the mapping of paired

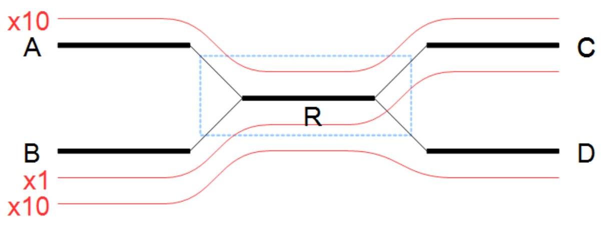


Figure 5.9: *A repeat with conflicts.*

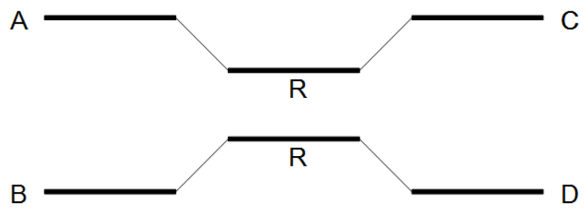


Figure 5.10: *Resolving a repeat with conflicts.*

reads to the long unambiguous paths in the graph which are created in the read optimization step described above. The distance estimation algorithm creates a histogram (H) of the paired distances between reads in each set of paired reads (see figure 5.11). Each of these histograms are then used to estimate paired distances as described in the following.

We denote the average number of observations in the histogram $H_{avg} = \frac{1}{|H|}\Sigma_d H(d)$ where $H(d)$ is the number of observations (reads) with distance d and $|H|$ is the number of bins in H. The gradient of H at distance d is denoted $H'(d)$. The following algorithm is then used to compute a distance interval for each histogram.

- Identify peaks in H as $\max_{i \leq d \leq j} H(d)$ where $[i, j]$ is any interval in H where $\{H(d) \geq \frac{H_{avg}}{2} | i \leq d \leq j\}$.
- For the two largest peaks found, expand the respective intervals $[i, j]$ to $[k, l]$ where $H'(k) < 0.001 \wedge k \leq i \wedge H'(l) > -0.001 \wedge j \leq l$. I.e. we search for a point in both directions where the number of observations becomes stable. A window of size 5 is used to calculate H' in this step.
- Compute the total number of observations in each of the two expanded intervals.
- If only one peak was found, the corresponding interval $[k, l]$ is used as the distance estimate unless the peak was at a negative distance in which case no distance estimate is calculated.
- If two peaks were found and the interval $[k, l]$ for the largest peak contains less than 1% of all observations, the distance is not estimated.
- If two peaks were found and the interval $[k, l]$ for the largest peak contains <2X observations compared to the smaller peak, the distance estimate is only computed if the range of distances is positive for the largest peak and negative for the smallest peak. If this is the case the interval $[k, l]$ for the positive peak is used as a distance estimate.

- If two peaks were found and the largest peak has $\geq$2X observations compared to the smaller peak, the interval $[k, l]$ corresponding to the largest peak is used as the distance estimate.

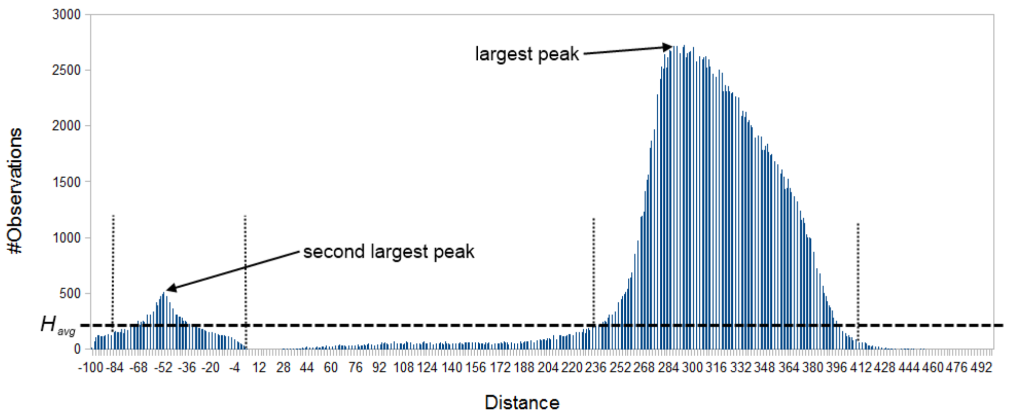


Figure 5.11: *Histogram of paired distances where H_{avg} is indicated by the horizontal dashed line. There is two peaks, one is at a negative distance while the other larger peak is at a positive distance. The extended interval $[k, l]$ for each peak is indicated by the vertical dotted lines.*

If a distance estimate for a data set is deemed unreliable, the estimate is ignored and replaced by the distance supplied by the user using the '-p' option for that data set. The '-e' option requires a file name argument ,which is used to output the result of the distance estimation for each dataset. The output is a tab-delimited file containing the estimated distances, if any, and a status code for each data set. The possible status codes are:

- **DISTANCE_ESTIMATED** The distance interval was estimated and used for scaffolding.
- **NO_DATA** No or very few reads were mapped as paired reads.
- **NOT_ENOUGH_DATA** Not enough reads were mapped as paired reads to give a reliable distance estimate.
- **NEGATIVE_DISTANCE** The distance interval was in the negative range which is usually caused by either wrong orientation of the reads or paired-end contamination in a mate-pair data set.
- **AMBIGIOUS_DISTANCE** Several possible distance intervals were detected but there was not enough data to select the correct one.
- **WRONG_DIRECTION** The orientation of the reads was not set correctly.

Only distance estimates with the DISTANCE_ESTIMATED status code is used for the assembly. In general we do not recommend that the automatic paired distance estimation is used on mate-pair reads where the expected distance is larger than 10Kbp as the distance estimate will often either fail or be inaccurate.

5.3.3 Optimization of the graph using paired reads

When paired reads are available, we can use the paired information to resolve large repeat regions that are not spanned by individual reads, but are spanned by read pairs. Given a set of paired reads that align to two nodes connected by a repeat region, the repeat region may be resolved for those nodes if we can find a path connecting the nodes with a length corresponding to the paired read distance. However, such a path must be supported by a minimum of four sets of paired reads before the repeat is resolved.

If it's not possible to resolve the repeat, scaffolding is performed where paired read information is used to determine the distances between contigs and the orientation of these. Scaffolding is only considered between two contigs if both are at least 120 bp long, to ensure that enough paired read information is available. An iterative greedy approach is used when performing scaffolding where short gaps are closed first, thus increasing the paired read information available for closing gaps (see figure 5.12).

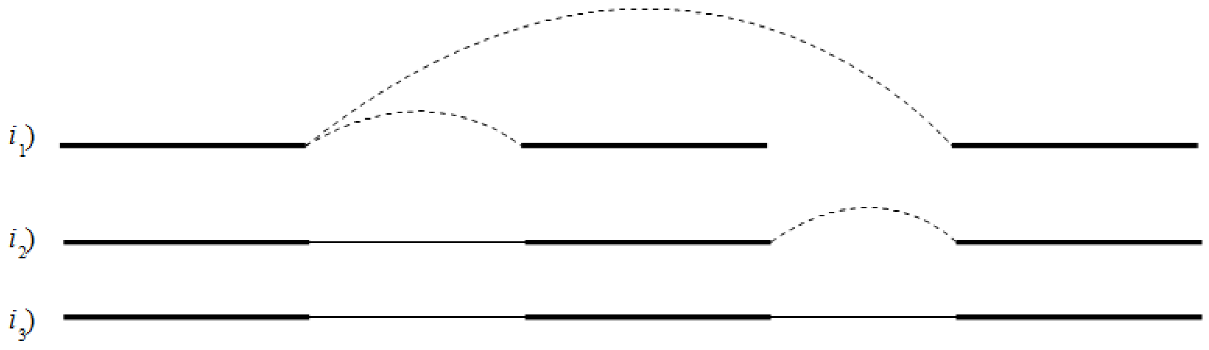


Figure 5.12: *Performing iterative scaffolding of the shortest gaps allows long pairs to be optimally used.* i_1 *shows three contigs with dashed arches indicating potential scaffolding.* i_2 *is after first iteration when the shortest gap has been closed and long potential scaffolding has been updated.* i_3 *is the final results with three contigs in one scaffold.*

Contigs in the same scaffold are output as one large contig with Ns inserted in between. The number of Ns inserted correspond to the estimated distance between contigs, which is calculated based on the paired read information. More precisely, for each set of paired reads spanning two contigs a distance estimate is calculated based on the supplied distance between the reads. The average of these distances is then used as the final distance estimate. The distance estimate will often be negative which happens when the paired information indicate that two contigs overlap. The assembler will attempt to align the ends of such contigs and if a high quality overlap is found the contigs are joined into a single contig. If no overlap is found, the distance estimate is set to two so that all remaining scaffolds have positive distance estimates.

Furthermore, Ns can also be present in output contigs in cases where input sequencing reads themselves contain Ns.

Additional information on how paired reads have been used to in the scaffolding step can be printed by using `-f` to specify an output file for GFF or AGP 2.0 formatted annotations.

The annotations in table format can be viewed by clicking the "Show Annotation Table" icon ( ) at the bottom of the Viewing Area. "Show annotation types" in the side panel allows you to select the annotation "Scaffold" among a list of other annotations. The annotations tell you about the scaffolding that was performed by the de novo assembler. That is, it tells you where particular contigs and those areas containing complete sequence information were joined together across regions without complete sequence information.

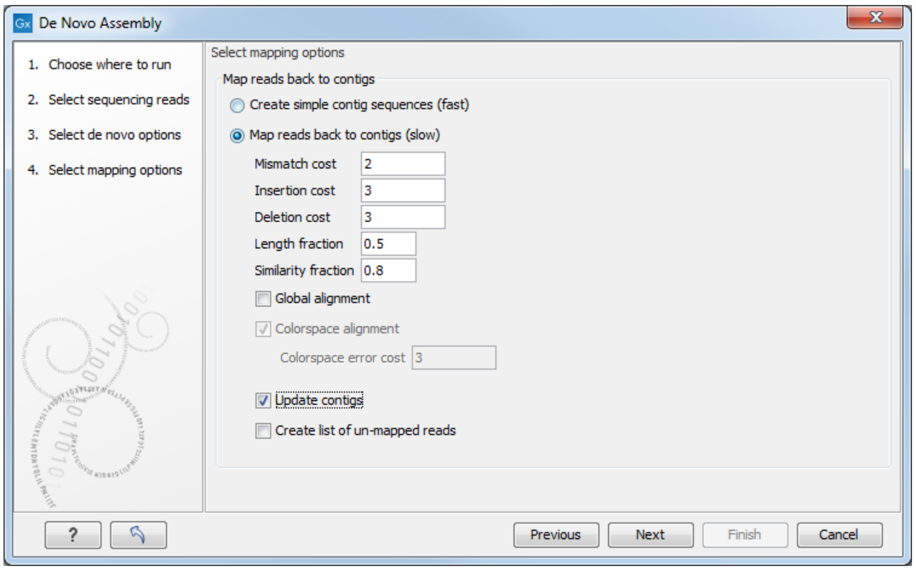
For the GFF format there are three types of annotations:

- **Scaffold** refers to the estimated gap region between two contigs where Ns are inserted.
- **Contigs joined** refers to the joining of two contigs connected by a repeat or another ambiguous structure in the graph, that was resolved using paired reads. Can also refer to overlapping contigs in a scaffold that were joined using an overlap.
- **Alternatives excluded** refers to the exclusion of a region in the graph using paired reads that resulted in a join of two contigs.

5.3.4 AGP export

The AGP annotations describe the components that an assembly consists of. This format can be validated by the NCBI AGP validator.

If the exporter is executed on an assembly where the contigs have been updated using a read mapping, the N's in some scaffolds might be resolved if you select the option "Update contigs" (figure 5.13).


De Novo Assembly

1. Choose where to run
2. Select sequencing reads
3. Select de novo options
4. Select mapping options

Select mapping options

Map reads back to contigs

( ) Create simple contig sequences (fast)
(●) Map reads back to contigs (slow)

Mismatch cost	2
Insertion cost	3
Deletion cost	3
Length fraction	0.5
Similarity fraction	0.8

☐ Global alignment
☑ Colorspace alignment
Colorspace error cost 3
☑ Update contigs
☐ Create list of un-mapped reads

Figure 5.13: *Select "update contigs" by ticking the box if you want to resolve scaffolds based on a read mapping.*

If the exporter encounters such a region, it will give a warning but not stop. If the exporter is executed on an assembly from GWB versions older than 6.5, it will often stop with an error saying that it encountered more than 10 N's which wasn't marked as a scaffold region. In this case the user would have to rerun the assembly with GWB version 6.5 or newer of the de novo assembler if they wish to be able to export to AGP.

Currently we output two types of annotations in AGP format:

- **Contig** a non-redundant sequence not containing any scaffolded regions.
- **Scaffold** the estimated gap region between two contigs.

5.3.5 Bubble resolution

Before the graph structure is converted to contig sequences, bubbles are resolved. As mentioned previously, a bubble is defined as a bifurcation in the graph where a path furcates into two nodes and then merge back into one. An example is shown in figure 5.14.

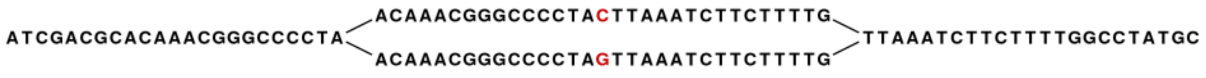
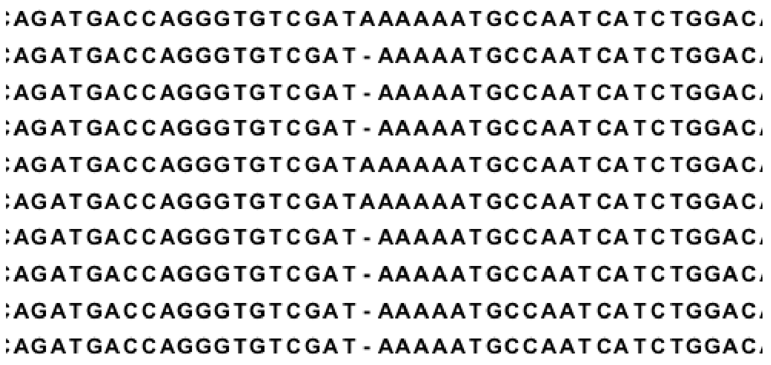

ATCGACGCACAAACGGGCCCCTA
ACAAACGGGCCCCTACTTAAATCTTCTTTTG
ACAAACGGGCCCCTAGTTAAATCTTCTTTTG
TTAAATCTTCTTTTGGCCTATGC

Figure 5.14: *A bubble caused by a heteroygous SNP or a sequencing error.*

In this simple case the assembler will collapse the bubble and use the route through the graph that has the highest coverage of reads. For a diploid genome with a heterozygous variant, there will be a fifty-fifty distribution of reads on the two variants, and this means that the choice of one allele over the other will be arbitrary. If heterozygous variants are important, they can be identified after the assembly by mapping the reads back to the contig sequences and performing standard variant calling. For random sequencing errors, it is more straightforward; given a reasonable level of coverage, the erroneous variant will be suppressed.

Figure 5.15 shows an example of a data set where the reads have systematic errors. Some reads include five As and others have six. This is a typical example of the homopolymer errors seen with the 454 and Ion Torrent platforms.



```
:AGATGACCAGGGTGTCGATAAAAAATGCCAATCATCTGGAC.
:AGATGACCAGGGTGTCGAT-AAAAATGCCAATCATCTGGAC.
:AGATGACCAGGGTGTCGAT-AAAAATGCCAATCATCTGGAC.
:AGATGACCAGGGTGTCGAT-AAAAATGCCAATCATCTGGAC.
:AGATGACCAGGGTGTCGATAAAAAATGCCAATCATCTGGAC.
:AGATGACCAGGGTGTCGATAAAAAATGCCAATCATCTGGAC.
:AGATGACCAGGGTGTCGAT-AAAAATGCCAATCATCTGGAC.
:AGATGACCAGGGTGTCGAT-AAAAATGCCAATCATCTGGAC.
:AGATGACCAGGGTGTCGAT-AAAAATGCCAATCATCTGGAC.
:AGATGACCAGGGTGTCGAT-AAAAATGCCAATCATCTGGAC.
```

Figure 5.15: *Reads with systematic errors.*

When these reads are assembled, this site will give rise to a bubble in the graph. This is not a problem in itself, but if there are several of these sites close together, the two paths in the graph will not be able to merge between each site. This happens when the distance between the sites is smaller than the word size used (see figure 5.16).

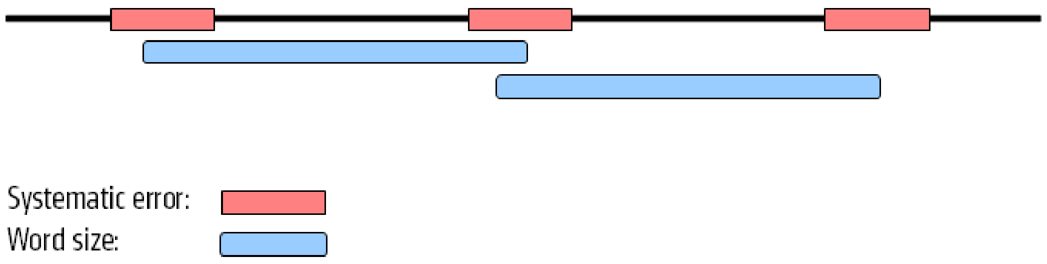


Figure 5.16: *Several sites of errors that are close together compared to the word size.*

In this case, the bubble will be very large because there are no complete words in the regions between the homopolymer sites, and the graph will look like figure 5.17.

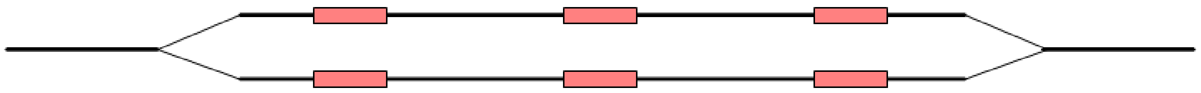


Figure 5.17: *The bubble in the graph gets very large.*

If the bubble is too large, the assembler will have to break it into several separate contigs instead of producing one single contig.

The maximum size of bubbles that the assembler should try to resolve can be set by the user. In the case from figure 5.17, a bubble size spanning the three error sites will mean that the bubble will be resolved (see figure 5.18).

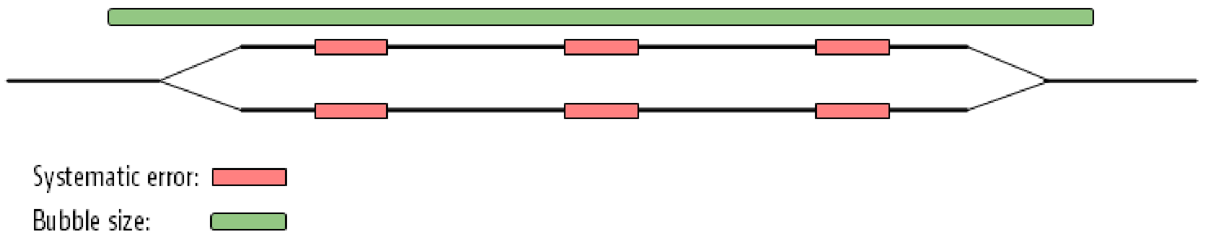


Figure 5.18: *The bubble size needs to be set high enough to encompass the three sites.*

While the default bubble size is often fine when working with short, high quality reads, considering the bubble size can be especially important for reads generated by sequencing platforms yielding long reads with either systematic errors or a high error rate. In such cases, a higher bubble size is recommended. For example, as a starting point, one could try half the length of the average read in the data set and then experiment with increasing and decreasing the bubble size in small steps. For data sets with a high error rate it is often necessary to increase the bubble size to the maximum read length or more. Please keep in mind that increasing the bubble size also increases the chance of misassemblies.

5.3.6 Converting the graph to contig sequences

The output of the assembly is not a graph but a list of contig sequences. When all the previous optimization and scaffolding steps have been performed, a contig sequence will be produced for every non-ambiguous path in the graph. If the path cannot be fully resolved, Ns are inserted as an estimation of the distance between two nodes as explained in section 5.3.3.

5.3.7 Summary

So in summary, the de novo assembly algorithm goes through these stages:

- Make a table of the words seen in the reads.
- Build a *de Bruijn* graph from the word table.
- Use the reads to resolve the repeats in the graph.
- Use the information from paired reads to resolve larger repeats and perform scaffolding if necessary.

- Output resulting contigs based on the paths, optionally including annotations from the scaffolding step.

These stages are all performed by the assembler program.

5.4 Randomness in the results

Different runs of the de novo assembler can result in slightly different results. This is caused by multi-threading of the program combined with the use of probabilistic data structures. If you were to run the assembler using a single thread, the effect would not be observed. That is, the same results would be produced in every run. However, an assembly run on a single thread would be very slow. The assembler should run quickly. Thus, we use multiple threads to accelerate the program.

The main reason for the assembler producing different results in each run is that threads construct contigs in an order that is correlated with the thread execution order, which we do not control. The size and "position" of a contig can change dramatically if you start building a contig from two different starting points (i.e. different words, or k-mers), which means that different assembly runs can lead to different results, depending on the order in which threads are executed. Whether a contig is scaffolded with another contig can also be affected by the order that contigs are constructed. In this case, you could see quite large differences in the lengths of some contigs reported. This will be particularly noticeable if you have an assembly with reasonably few contigs of great length.

For the moment, the output of runs may vary slightly, but the overall information content of the assembly should not be markedly different between runs.

5.5 Specific characteristics of CLC algorithm

There are some advantages and some disadvantages of CLC's algorithm when compared to other programs such as *Velvet* [Zerbino and Birney, 2008] and *SOAPdenovo* [Li et al., 2010]. The advantages are:

- `clc_assembler` does not use as much RAM as other programs
- `clc_assembler` program is quite fast
- `clc_assembler` readily uses data from mixed sequencing platforms (Sanger, 454, Illumina, SOLiD[1], etc).

The reason that we are able to use little RAM compared to other programs is that we have a very strong focus on keeping the data structures very compact. When appropriate, we also use the hard drive for temporary data rather than using RAM.

The speed of the assembly program has been achieved by threading many parts of the program to use all available CPU cores. Also, some parts of the program are done using assembler code including SIMD vector instructions to get the optimal performance.

[1]See how SOLiD is supported in section 5.6

5.6 SOLiD data support in de novo assembly

SOLiD sequencing is done in color space. When viewed in nucleotide space this means that a single sequencing error changes the remainder of the read. An example read is shown in figure 5.19.

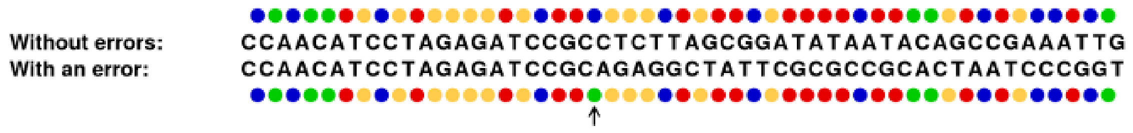

Without errors:
CCAACATCCTAGAGATCCGCCTCTTAGCGGATATAATACAGCCGAAATTG
With an error:
CCAACATCCTAGAGATCCGCAGAGGCTATTCGCGCCGCACTAATCCCGGT

Figure 5.19: *How an error in color space leads to a phase shift and subsequent problems for the rest of the read sequence*

Basically, this color error means that C's become A's and A's become C's. Likewise for G's and T's. For the three different types of errors, we get three different ends of the read. Along with the correct reads, we may get four different versions of the original genome due to errors. So if SOLiD reads are just regarded in nucleotide space, we get four different contig sequences with jumps from one to another every time there is a sequencing error.

Thus, to fully accommodate SOLiD sequencing data, the special nature of the technology has to be considered in every step of the assembly algorithm. Furthermore, SOLiD reads are fairly short and often quite error prone. Due to these issues, we have chosen not to include SOLiD support in the first algorithm steps, but only use the SOLiD data where they have a large positive effect on the assembly process: when applying paired information. Thus, the `clc_assembler` program has a special option ("-p d") to indicate that a certain data set should be used only for its paired information. This option should always be applied to SOLiD data. It is also useful for data sets of other types with many errors. The errors might have the effect of confusing the initial graph building more than improving it. But the paired information is still valuable and can be used with this option.

5.7 Command line options

This section provides details of the command line options available for the clc_assembler command. As with all other programs in the CLC Assembly Cell, full usage information are given in Appendix A and can be viewed by executing the command without any arguments.

Note that you can use the `clc_sequence_info` program with the '-n' option to get statistics on the result of a de novo assembly. This is described in the **Viewing and reporting tools section of the manual.**

5.7.1 Specifying the data for the assembly and how it should be used

The parameters described in this section can be used multiple times within a single clc_assembler command.

Information provided using flags that can be specified multiple times in a single command pertains to **all inputs that follow** until you specify otherwise. For example, if you state you wish reads to be used for guidance only, all datasets entered via the command from that point forward will be used only for guidance, until a point in the command where you choose to indicate that reads should be used at all stages of the assembly. Please see the examples section below for further details on this.

Some of these options are most easily explained by considering the de novo as consisting of four steps:

1. Create fragments from reads.
2. Connect fragments based on all reads to form a graph.
3. Optimize the graph based on all reads.
4. Optimize the graph based on paired reads.

`-p <par> / --paired <par>`

Here, par is a set of parameters, which indicate the pair status of your data, and for paired data, the relative orientations and expected distances between members of the pair. Options after the -p flag are also used to indicate if paired data are in two files and also whether any particular data should be used only for its paired distance information during the final phase of the assembly.

Paired status

Data are assumed to be paired by default. To indicate that the data contain single reads, that is, that the data are not paired, the single string value "no" is given. I.e. "-p no" followed by the name of the read file.

Paired information

For paired data, par consists of four strings: `<mode> <distance_mode> <min_dist> <max_dist>`

`mode` indicates the relative orientation of the reads in a pair set. These can be **ff**, **fb**, **bf** or **bb**. These are used for "forward-forward" reads, "forward-reverse" reads, "reverse-forward" reads and "reverse-reverse" reads[2].

`distance_mode` indicates the point on paired reads from which the distance measure you provide is taken. The options are **ss**, **se**, **es** or **ee**. These mean "start-start", "start-end" reads, "end-start" reads and "end-end".

`<min_dist> and <max_dist>` give the minimum and maximum distance range for the distances between the pairs. Where to take the start and end points of this distance range is what is specified by the `distance_mode` described above.

So, `-p fb ss 180 250` would indicate that the reads are inverted and pointing towards each other, that the distance range includes the sequences of both the reads as well as the fragment between them, and that the distance range is between 180 and 250 bases.

How data should be used in the assembly

`-p d <mode> <distance_mode> <min_dist> <max_dist>` An additional option, **d**, can be added to the -p flag to indicate that the reads should only be used in the fourth step of the assembly, as listed at the top of this section. That is, the paired distance information associated with these reads is used when optimizing the graph. The sequence information itself is not used

[2]The letter "b" in the mode strings refers to the word "backwards", but the more common word to use to describe this relative orientation is "reverse".

towards the assembly result. This is useful for data such as SOLiD, where the reads are quite short and may contain many errors. Such reads would not, by themselves, be of great help in building and optimizing the graph initially, but the paired information associated with them can be valuable during the final graph optimization stage.

Please also refer to the information about the `-g` option in the section below.

`-g <mode>/ --fragmentmode <mode>`

You might choose to use this option with longer reads that are expected to contain many errors, for example, in the case of 454 reads.

Here, mode can be one of two values: "use" and "ignore". The default is that all data is run with `-g 'use'`.

Providing the "-g ignore" before the name of a read set indicates that this read set (and all others after this point in the command, until any point where the "-g use" option is entered in the command) is **not** used in the first step of the assembly, as described above, where fragments are generated from the reads. In other words, read sets following "-g ignore" in the command, are used in steps 2, 3 and 4 as listed above, where the graph is determined and optimized.

`-n / --no-scaffolding` Pair distance information is used for the creation of contig sequences, but no scaffolding (i.e. making associations between contigs) is performed.

`-q / --reads`

This flag indicates that the information that follows are read filenames. Read files can be in fasta, fastq or sff format.

`-i <file1> <file2> / --interleave <file1> <file2>`

To input paired read data that is in two files, where one read of each pair is in one file and the other of each pair is in the second file, the pair of file names should be provided after the -i flag.

Read files for paired data entered without the -i flag are assumed to contain interleaved pairs. That is, the first sequence is the first member of a pair, the second sequence is the second member of that same pair, the third sequence is the first member of the second pair, the fourth sequence is the second member of the second pair, and so on.

Mixed files, that is, single input files that contain both paired and unpaired reads, cannot be used as input with the clc_assembler command, unless the intention is to treat all reads as single reads.

5.7.2 Specifying information for the assembly

Options in this section can alter the results of the assembly. The **How it works** section of the manual gives further details that are relevant to how these settings may affect an assembly.

`-w <n> / --wordsize <n>`

Set n to be the word size for the de Bruijn graph. The default is based on the size of the input as described in the **How it works** section of the manual.

`-b <n> / --bubblesize <n>`

Set the maximum bubble size for the de Bruijn graph. The default is 50 bases.

`-e <file> / --estimatedistances <file>`

This setting estimates the distances for paired reads as observed within unscaffolded contigs. These distances are then used in the scaffolding step. If multiple sets of paired data have been input, the distances are estimated separately for each data set. The distances calculated will be saved to the file specified as the argument to this parameter.

When this flag is used, the program will aim to identify tight distance intervals from areas containing a substantial number of the mapped reads for each dataset.

There are situations where it is not possible to estimate accurate paired distances from the data, such as:

- No best candidate interval for distance estimation can be specified as the two best candidate intervals to be used for estimating the distances differ only by a factor of two in the number of pairs they contain.
- The best interval suggests a negative average distance for the paired reads in a dataset.
- More than half the reads have the wrong relative orientation.
- The best interval contains less than 1% of the mapped reads in that dataset.

If it is not possible to estimate an accurate distance from the data for any particular paired read set, then the original paired distance entered as part of the parameter settings associated with the **-p flag** will be used. Errors and warnings associated with such situations will be written to the file specified with the -e parameter.

5.7.3 Specifying information about the outputs

`-o <file> / --output <file>` Give the name for the file that will contain the contigs, in fasta format, that are assembled.

This parameter is required.

`-m <n> / --min-length <n>` Set the minimum length for contigs to be output from the assembly process. The default value is 200 bases.

`-f <file> / --feature_output <file>`

Providing this option indicates that the annotations associated with scaffolding should be output.

The output can be in GFF format (the default), or in AGP format. The file suffix you provide specifies the output format. For AGP format, use '.agp'. For gff format, you can use whatever name you like, but it would be usual to use the filename suffix '.gff'.

5.7.4 Other options

`--cpus <n>` Specify the maximum number of cpus that should be used by the assembly process. If not set explicitly, the process assumes it has access to as much of your computer's cpus as it needs.

`-v / --verbose`: This option specifies that verbose reporting should be turned on. This results in various information being written to the terminal when the process is running, include the word size used if you have chosen to allow that to be determined by the assembly program.

5.7.5 Example commands

The command below is a mixed assembly, where a set of paired reads, available in two files with one member of each pair in each file, is used for all stages of the assembly, and a set of 454 data is used for guidance only. The output file, containing the fasta formatted assembled contigs, will be called myContigs.fasta.

Note the use of the `-p no` before the non-paired, 454 data. This ensures that the assembler knows that the data about to be entered are not paired. It effectively overwrites the earlier -p information provided earlier in the command.

```
clc_assembler -o myContigs.fasta -p fb ss 200 400 -q -i pairedRead-member1.fastq
                           pairedRead-member2.fastq -g ignore -p no -q 454read.sff
```

The above command is equivalent to:

```
clc_assembler -o myContigs.fasta  -g ignore -p no -q 454read.sff -g use
              -p fb ss 200 400 -q -i pairedRead-member1.fastq pairedRead-member2.fastq
```

Examples of undesirable commands

A command like the following would imply that the data in 454reads. sff was paired, in an interleaved file, with the reads having relative orientation of forward-reverse, and a paired distance range of 200 to 400 bases. That is, the earlier information provided to the -p parameter would be used for all the following data entered in the command.

```
clc_assembler -o myContigs.fasta -p fb ss 200 400 -q -i pairedRead-member1.fastq
                                       pairedRead-member2.fastq -g ignore -q 454read.sff
```

A command like the following would fail because all reads are now used in a guidance only role. This leaves no reads being used to create the graph fragments in the initial stage. The reason here is that the `-g ignore` parameter is given early in the command and because not `-g use` parameter is entered later, all read sets are ignored for the building of the fragments.

```
clc_assembler -o myContigs.fasta -g ignore -p no -q 454read.sff -p fb ss 200 400 -q
                                       -i pairedRead-member1.fastq pairedRead-member2.fastq
```

5.8 New long read assembly tools (beta)

The purpose of the last three sections in this chapter on de novo assembly is to introduce two new tools that we have made specifically to assist with long read assembly. Note that both tools are still considered beta quality, and that they have exclusively been tested with PacBio data.

The two tools work in tandem:

- The `clc_correct_pacbio_reads` tool (optionally) error-corrects PacBio reads, leading to a much improved subsequent PacBio assembly.
- The `clc_assembler_long` is our long read de novo assembler.

In the next two subsections, we will show you how to use the two tools.

5.9 The clc_correct_pacbio_reads tool (beta)

The `clc_correct_pacbio_reads` tool performs error-correction of PacBio reads.

IMPORTANT NOTICE: This tool relies on certain methods that are the intellectual property of Pacific Biosciences. Consequently, the use of this tool with any data other than data generated on a Pacibic Biosciences instrument constitutes a violation of the end-user license agreement that users of the CLC Assembly Cell agree to during installation.

A typical PacBio run produces a wide range of different read lengths. All other things being equal, the longer a read is, the more useful it is for de novo assembly. The primary reason for this is the ability of long reads to span longer repeats and connect with more unique sequence surrounding the repeat, clearly delimiting that repeat region in the final assembly.

Raw PacBio reads exhibit a much higher rate of (random) errors than short read technologies, such as Illumina. Unaddressed, this added noise would confuse the assembler, leading to a poor assembly.

The `clc_correct_pacbio_reads` tool performs optional, but highly recommended, pre-processing of the raw reads that alleviates this problem. It takes as input the raw reads and produces a new FASTA file[3] containing error-corrected versions of those long reads.

The error correction itself is a step-wise process. Somewhat simplified, it consists of three steps:

- **Step 1.** The raw reads are split into two classes: long reads (or *seed reads*) and short reads (or *correction reads*). The way the split is made is governed by the `--fraction` parameter. By default, the longest reads, corresponding to 30% of the total read length, become the seed reads.
- **Step 2.** The short reads are then mapped against the long reads. Typically, several short reads end up covering each position of the long read.
- **Step 3.** Finally, each long read is considered along with the corresponding short reads to form a consensus sequence by per-position majority vote. Because the noise in the raw sequences is close to random, much of that noise is eliminated by this process, and we can consider the consensus sequence a corrected version of the read.

[3]The corrected reads will not have quality scores, as it is unclear how to calculate these in any meaningful way. However, this is not a problem, as quality scores are not used in the subsequent assembly process.

5.9.1 Interlude: Converting PacBio's BAM to FASTA

PacBio reads are now output from their sequencing instruments in BAM format (http://pacbiofileformats.readthedocs.org/en/3.0/BAM.html). As our tools only work with FASTA or FASTQ, we recommend that you convert the reads to FASTA format using samtools (https://github.com/samtools/samtools). Once you have samtools up and running, the following command does the trick:

```
samtools fasta -0 raw_reads.fa original_pacbio_reads.bam
```

5.9.2 Basic usage

To specify the raw input reads, use the `-q/--reads` parameter followed by one or more FASTA or FASTQ files.

The resulting corrected reads are output to a FASTA file whose name is specified after the `-o/--output` parameter.

5.9.3 Advanced parameters

We currently expose four parameters that allow you to tweak the behavior of the algorithm.

As briefly mentioned above, you can use the `-f/--fraction` parameter (default value is 30) to control how the distinction between *long reads* and *short reads* is made. By default, the longest reads corresponding to 30% of the total read length are considered "long".

Once all the short reads have been mapped to the long reads, you have the option of adjusting three filtering criteria. If short-on-long coverage falls beneath the value provided after the `-m/--min-coverage` parameter, those parts of the corrected long read are excised, resulting in several, shorter reads. Those shorter reads are then filtered according to the following two parameters:

- The `-l/--min-read-length` parameter allows you to set a minimum read length (default is 1000). Corrected reads shorter than this are discarded.
- The `-a/--min-average-coverage` parameter specifies a lower bound on the average coverage across a corrected read (default is 15). If a read's average coverage is lower, that read will be discarded.

Here is a complete example, where we correct the PacBio reads contained in `raw_reads.fa` and, producing a single file containing the corrected long reads `corrected_reads.fa`:

```
clc_correct_pacbio_reads -q raw_reads.fa -o corrected_reads.fa
                         -f 30 -m 10 -l 1000 -a 15
```

5.10 The clc_assembler_long tool (beta)

In this section we give a brief introduction to the new long read assembler.

We discuss the method employed, basic usage, and provide a complete example including error correction for PacBio reads.

Note that the long read assembler was designed to be used for the assembly of microbial genomes and small eukaryotic genomes.

5.10.1 Method

Like our old assembler, the long read assembler builds a kind of *de Bruijn graph* to do the assembly.

Since the method is k-mer based, a (fixed) word size k, should be chosen and provided to the assembler using the `-w/--wordsize` parameter. If left out, the assembler will do its best to decide on a word size based on the size of the input data.

When the assembler is run, we first count all the k-mers occurring in the reads, provided as one or more FASTA or FASTQ files containing the reads. The k-mers are identified with their reverse-complements as we have no way of knowing which strand gave rise to a particular read.

Once all the k-mers have been counted, we throw away all k-mers that occur fewer than a user-specified number of times, since unique, or rarely occurring, k-mers can often be attributed to sequencing errors.

In the next step, any k-mers that always occur next to each other, i.e. *only* have $k - 1$ overlap with one another, are joined together, iteratively, into longer *fragments*[4].

Next, we do a second pass through the read data; mapping all the reads back to the fragments. Reads that span/link multiple fragments give rise to connections between the fragments, and together, the fragments (nodes) and those connections (edges) form a graph structure. If there is a gap between two fragments, that gap is replaced by a majority consensus sequence of the reads spanning it.

From this point, the assembly graph represents our knowledge of the genome that we aim to assemble. However, it is often a very messy graph, and it is not immediately possible to read off the chromosome, or chromosomes, that we are trying to assemble. Luckily, we have a lot of information available, such as information about how well the individual edges are supported. We use this information along with the graph topology itself to reduce noise and tease out the actual information content of the graph. This simplification process consists of many, quite technical, steps and will not be covered in any detail here.

Once the graph has been simplified into a number of longer contigs, as an optional step, it is possible to *polish* the contigs by mapping the reads back onto the contigs, replacing them with the respective consensus sequences. In the case of error-corrected PacBio reads, you will get the best result by using the original raw reads, instead of the error-corrected ones, to do the polishing.

Finally, the resulting assembly/contigs are output to a FASTA file.

[4]Given a set of reads, word size k, and minimum coverage c; the resulting fragments are unambiguously defined.

5.10.2 Input parameters

The input reads must be provided as one or more FASTA or FASTQ files following the `-q/--reads` parameter.

5.10.3 Graph construction

We expose two optional parameters that greatly influence the way the fragments, and subsequently the assembly graph, gets constructed.

The `-w/--wordsize` parameter allows you to set the fixed k-mer length used during fragment construction. If left out, the assembler tries to estimate a reasonable word size k based on the number of bases in the input data.

The `-c/--min-coverage` parameter specifies a lower bound on the number of times, we want to see any given k-mer for it to play a part in fragment construction (default 3).

Should you want to inspect the resulting fragments, you can choose to save them to a FASTA file using the `-f/--savefragments` parameter.

You also have the option of saving the constructed graph using the `--savegraph` parameter. That way you can experiment with different post-processing steps without continuously rebuilding the graph. Afterwards, you simply reuse the graph by adding the `--loadgraph` parameter in subsequent runs.

5.10.4 Graph post-processing

The `-a/--anchor-length` specifies a minimum length for fragments to become part of the final assembly graph (default 100). Raising this helps to eliminate entangled, noisy structures in the graph. On the flip side, you may need a lower value to correctly resolve complex regions.

5.10.5 Contig post-processing

As explained above, you can polish the final contigs by mapping the reads back to them, replacing them with the resulting consensus sequence. If you want to do this, simply provide the read file, or files, to use for polishing following the `-r/--polishingreads` parameter.

5.10.6 Output parameters

As the output of the assembly process is a number of contigs/sequences, you must provide the name of the FASTA file they should be written to. This is done using the `-o/--output` parameter.

If you want to filter out short contigs before they are written to the output file, you can specify a cut-off using the `-m/--min-length` parameter (default 500). Contigs below that length will not be output.

5.10.7 An example including error-correction for PacBio reads

Say we start out having a BAM file with raw reads from a PacBio instrument, `raw_reads.bam`. First thing we need to do is extract the reads in FASTA format using samtools:

```
samtools fasta -0 raw_reads.fa raw_reads.bam
```

We now have the reads in the FASTA file `raw_reads.fa` and will no longer be needing the original BAM file.

Next, we error-correct the raw PacBio reads:

```
clc_correct_pacbio_reads -q raw_reads.fa -o corrected_reads.fa
```

We are now ready to start the assembly. We want to use the corrected reads (`corrected_reads.fa`) for the assembly itself and use the raw reads (`raw_reads.fa`) for polishing.

```
clc_assembler_long -q corrected_reads.fa -r raw_reads.fa -o contigs.fa
```

Let us assume that we have inspected the result in `contigs.fa` and are unhappy with the quality of the result; perhaps we would like to see fewer contigs or would like a total contig length closer to that of our expected genome length. We can then re-run the same assembly with slightly different parameters and opt to save the graph, in case we need to do more runs:

```
clc_assembler_long -q corrected_reads.fa -r raw_reads.fa -o contigs.fa
                   --min-coverage 4 --savegraph raw_reads.graph
```

Say we inspect the result again and are still not quite happy. Maybe we want to raise the anchor length because we expect there to be some unwanted noise in the graph. Because we have saved the graph, we can cut run-time significantly on this run by re-loading the saved graph:

```
clc_assembler_long -q corrected_reads.fa -r raw_reads.fa -o contigs.fa
                   --min-coverage 4 --loadgraph raw_reads.graph
                   --anchor-length 200
```

If the data is good and the sample genome is tractable, we should end up with a decent result after a few iterations of this procedure.

Chapter 6

Reporting tools

6.1 The clc_sequence_info Program

The clc_sequence_info program gives some basic information about the sequences in a fasta file:

```
File                           data/paired.fasta

Number of sequences                 47356

Residue counts:
  Total                          11114027

Sequence length:
  Minimum                             170
  Maximum                             240
  Average                             234.69
```

Using the '-r' options include counts of the different types of nucleotides, with all ambiguous nucleotides counted as N's. The '-a' option used together with the '-r' option does the counts for amino acids.

The lengths of the sequences can be printed or summarized using the '-l' and '-k' options, respectively.

It is also possible to get various sequence length statistics. Using the '-n' option, the N50 value of the sequences is calculated. The N50 value means that the sum of sequences of this length or longer is at least 50% of the total length of all sequences. This is useful to get a quick quality overview of a de novo assembly.

Use the '-c' option to disregard all sequences under a certain length from being considered in the statistics. This is sometimes useful for analyzing de novo assembly results, where short sequences may not be of interest.

Further details are available in Appendix A.

6.2 The clc_mapping_table Program

The clc_mapping_table program takes a single cas file as input and prints assembly information for each read. By default, clc_mapping_table makes a table with one read per row. The columns are:

- Read number (starting from 0).
- Read name (enable using the '-n' option).
- Read length.
- Read position for alignment start.
- Read position for alignment end.
- Reference sequence number (starting from 0).
- Reference position for alignment start.
- Reference position for alignment end.
- Whether the read is reversed (0 = no, 1 = yes).
- Number of optimal locations for the read.
- Alignment score (enable using the '-s' option).

If a read does not match, all columns except the read number and name are '-1'. If a read is reverse, the read positions for the alignment start and end are given after the reversal of the read. The sequence positions start from 0 indicating before the first residue and end at the sequence length indicating after the last residue. So a read of length 35, which matches perfectly will have an alignment start position of 0 and an alignment end position of 35.

Here is part of an example output using both the '-n' and the '-s' option:

```
208       SLXA-EAS1_89:1:1:622:715/1       35    0    35    0      89385      89420  0   1  35
209       SLXA-EAS1_89:1:1:622:715/2       35    0    35    0      89577      89612  1   1  35
210       SLXA-EAS1_89:1:1:201:524/1       35    0    32    0       4829       4861  0   1  29
211       SLXA-EAS1_89:1:1:201:524/2       -1   -1    -1   -1         -1         -1 -1  -1  -1
212       SLXA-EAS1_89:1:1:662:721/1       35    0    35    0      38254      38289  1   1  35
213       SLXA-EAS1_89:1:1:662:721/2       35    0    35    0      38088      38123  0   1  32
214       SLXA-EAS1_89:1:1:492:826/1       35    0    35    0      81872      81907  1   1  35
215       SLXA-EAS1_89:1:1:492:826/2       35    0    35    0      81685      81720  0   1  35
```

As the read names indicate, the data are from a paired experiment. Read 211 does not match at all and only the first 32 out of the 35 positions in read 210 matches. The score for this read is 29, indicating that a mismatch is also present (31 - 2 = 29). Read 213 also has a mismatch while the rest of the sequences match perfectly. We can also see that the pairs are located close together and on opposite strands.

Use the '-a' option to get a very detailed output ('-n' and '-s' are without effect here):

```
SLXA-EAS1_89:1:1:622:715/1 has 1 match with a score of 35:
      89385 TTGCTGTGGAAAATAGTGAGTCATTTTAAAACGGT 89419      coli
            |||||||||||||||||||||||||||||||||||
            TTGCTGTGGAAAATAGTGAGTCATTTTAAAACGGT            read

SLXA-EAS1_89:1:1:622:715/2 has 1 match with a score of 35:

      89577 AAACTCCTTTCAGTGGGAAATTGTGGGGCAAAGTG 89611      coli
            |||||||||||||||||||||||||||||||||||
            AAACTCCTTTCAGTGGGAAATTGTGGGGCAAAGTG            reverse read

SLXA-EAS1_89:1:1:201:524/1 has 1 match with a score of 29:

       4829 ATCCAGGCGAATATGGCTTGTTCCTCGGCACC    4860      coli
            ||||||||||||||||||| ||||||||||||
            ATCCAGGCGAATATGGCTTTTTCCTCGGCACCCCG            read

SLXA-EAS1_89:1:1:201:524/2 has 0 matches

SLXA-EAS1_89:1:1:662:721/1 has 1 match with a score of 35:

      38254 AGGGCATTCGATACGGTGGATAAGCTGAGTGCCTT 38288      coli
            |||||||||||||||||||||||||||||||||||
            AGGGCATTCGATACGGTGGATAAGCTGAGTGCCTT            reverse read

SLXA-EAS1_89:1:1:662:721/2 has 1 match with a score of 32:

      38088 ACTGAGTGATTGATTCGCGAGCCACATACTGTGGA 38122      coli
            |||||||||||||||||||||||||||||| ||||
            ACTGAGTGATTGATTCGCGAGCCACATACTCTGGA            read

SLXA-EAS1_89:1:1:492:826/1 has 1 match with a score of 35:

      81872 GCATCCAGCACTTTCAGCGCCTGGGTCATCACTTC 81906      coli
            |||||||||||||||||||||||||||||||||||
            GCATCCAGCACTTTCAGCGCCTGGGTCATCACTTC            reverse read

SLXA-EAS1_89:1:1:492:826/2 has 1 match with a score of 35:

      81685 TTCTGGTTGCTGGTCTGGTGGTAAATGTTCCCACT 81719      coli
            |||||||||||||||||||||||||||||||||||
            TTCTGGTTGCTGGTCTGGTGGTAAATGTTCCCACT            read
```

Note! The positions in the standard output assumes the reference sequence starts at 0. However, the '-a' option assumes that the reference starts at 1. This is due to the fact that the '-a' option is intended to produce human-readable output whereas the standard option is intended to be used by computer programs.

If multiple hit positions are recorded in the cas file (using the -t option when running the assembly), running the assembly_ table with the '-m', the output looks like this:

```
480         35    0   35    0     19625     19660  0   1
481         35    0   35    0     19815     19850  1   1
482         35    0   35    0   2512501   2512536  1   3
-           35    0   35    0    607436    607471  1   3
-           35    0   35    0     15593     15628  1   3
483         35    0   35    0   2512321   2512356  0   3
-           35    0   35    0    607256    607291  0   3
-           35    0   35    0     15413     15448  0   3
484         35    0   35    0     14374     14409  1   1
485         35    0   35    0     14194     14229  0   1
```

Reads 482 and 483 map in three places and they are all printed. The order is random, which has the advantage that using the first match (according to output order) is the same as using a random match. For paired data (like these), the matches are in the same order for two *paired* reads. So the first match for 482 belongs with the first match for 483, etc.

For alignment output in clc_mapping_table with "-m", it looks like this:

```
SLXA-EAS1_89:1:1:980:945/1 has 1 paired match with a score of 35: (alignment 1)

      19626 AGCTCCCCCAAAGTTAAGGTGGGGAGATAGATTA 19660      coli
            ||||||||||||||||||||||||||||||||||
            AGCTCCCCCAAAGTTAAGGTGGGGAGATAGATTA            read

SLXA-EAS1_89:1:1:980:945/2 has 1 paired match with a score of 35: (alignment 1)

      19816 GATAGTGTTTTATGTTCAGATAATGCCCGATGACT 19850      coli
            ||||||||||||||||||||||||||||||||||
            GATAGTGTTTTATGTTCAGATAATGCCCGATGACT            reverse read

SLXA-EAS1_89:1:1:307:821/1 has 3 paired matches with a score of 35: (alignment 1)

    2512502 CGGCCCCGGGGGGATGTCATTACGTGAAGTCACTG 2512536    coli
            ||||||||||||||||||||||||||||||||||
            CGGCCCCGGGGGGATGTCATTACGTGAAGTCACTG            reverse read

SLXA-EAS1_89:1:1:307:821/1 has 3 paired matches with a score of 35: (alignment 2)

     607437 CGGCCCCGGGGGGATGTCATTACGTGAAGTCACTG 607471     coli
            ||||||||||||||||||||||||||||||||||
            CGGCCCCGGGGGGATGTCATTACGTGAAGTCACTG            reverse read

SLXA-EAS1_89:1:1:307:821/1 has 3 paired matches with a score of 35: (alignment 3)

      15594 CGGCCCCGGGGGGATGTCATTACGTGAAGTCACTG 15628      coli
            ||||||||||||||||||||||||||||||||||
            CGGCCCCGGGGGGATGTCATTACGTGAAGTCACTG            reverse read

SLXA-EAS1_89:1:1:307:821/2 has 3 paired matches with a score of 35: (alignment 1)

    2512322 GGTATTACGCCTGATATGATTTAACGTGCCGATGA 2512356    coli
            ||||||||||||||||||||||||||||||||||
            GGTATTACGCCTGATATGATTTAACGTGCCGATGA            read
```

Options for the clc_mapping_table are described in Appendix A.

6.3 The clc_mapping_info Program

Whereas clc_mapping_table outputs detailed information about individual matches, the clc_mapping_info program instead gives an overview:

```
General info:

  Program name          clc_mapper
  Program version       1.00.31043
  Program parameters   -o tmp.cas -d data/paired.fasta -q data/paired_reads.fasta -m

  Contig files:
    data/paired.fasta

  Read files:
    data/paired_reads.fasta

Read info:

  Contigs                        1
  Reads                     108420
    Unassembled reads         1506
    Assembled reads         106914
      Multi hit reads            0

Alignment info:

  Number of inserts             13
  Number of deletes             42
  Number of mismatches        9253

Coverage info:

  Total sites               100000

  Average coverage              37.29
  Sites covered 0 times          0
  Sites covered 1 time           0
  Sites covered 2 times          3
  Sites covered 3+ times     99997

Contig info:

  Contig     Sites     Reads   Coverage
       1    100000    106914      37.29
```

It is possible to make an analysis of paired distances using the clc_mapping_info program. This is done with the standard '-p' option and results in an output like this:

```
Paired reads info:

  Pairs                     2478655
    Average distance            215.44
    99.9 % of pairs between       175 - 253
    99.0 % of pairs between       191 - 241
    95.0 % of pairs between       197 - 234

  Not pairs                  143727
    Both seqs not matching    21946
    One seq not mathing       62938
    Both seqs matching        58843
      Different contigs           0
      Wrong directions        40524
      Too close                 663
      Too far                 17656
```

Note that for paired analysis clc_mapping_info assumes that read one pairs with read two, read three with read four, etc. Thus, it is crucial that the reads are from a paired experiment and that they are assembled in the right order, possibly using the interleaved option for creating the assembly. If an assembly has a mixture of paired and unpaired data, use clc_submapping to make an assembly with only the paired data before analyzing.

When a dataset contains paired data of unknown distances, a good approach is to make an initial reference assembly without using paired information. Then the clc_mapping_info program can be used to investigate the paired distance properties of the data using wide limits for the distances. Finally, a reference assembly run can be performed with the estimated paired distances at a suitable distance interval. To get a quicker result, the initial reference assembly run may be done on only a part of the data, using ungapped alignments, and/or using stricter scoring criteria. These factors will usually not affect the paired distance properties of the results, but a smaller fraction of the reads might match.

Further details can be found in Appendix A.

Chapter 7

Assembly post-processing tools

This chapter covers tools included in CLC Assembly Cell that can be used to further process cas assembly files.

7.1 The clc_change_cas_paths Program

Cas files contain information about the files containing the reference and read data used in the mapping. This includes the paths to those files. The clc_change_cas_paths program allows you to change the file names and paths for the read and reference data files referred to in a cas file. This is useful if you have moved the sequence data files, and also can be useful when sharing cas files and the constituent data (reads and references) with others.

In addition to changing the locations of the data files, this tool can also be useful for changing relative file paths to absolute paths or vice versa.

The data file information is provided to this tool using the same parameters used with the clc_mapper tool. That is, using the '-d', '-q', and '-i' options. The input cas file is specified with the '-a' option and the cas file to be generated, containing the updated file locations, is specified with the '-o' option.

By default, the `clc\_change\_cas\_paths` program compares the sequence files to make sure they contain the same data. If the original read files (in their original location) no longer exist, or if you are certain you are working with the correct data files and wish to skip this check, you can use the n flag.

For the `clc\_change\_cas\_paths` command to succeed, you must provide the (new) locations of the data in the same order as they were used when creating the cas file originally. If you do not already know what this order is, then you can find out by running the `clc\_mapping\_info` program on the cas file. Providing the s flag to that tool skips checks of the contigs and thus can save some time here.

```
clc_mapping_info -s myassembly.cas
```

The order of the reads and reference files listed in that output is the same order they should be provided to the `clc\_change\_cas\_paths` tool.

Generally speaking it is a good idea to use different file names for the input cas file and the output cas file so the original is retained as backup. However changes can be made in place if the same cas file name is used for both the input and output.

For a full list of parameters, please refer to Appendix A.

7.2 The clc_filter_matches Program

The clc_filter_matches program removes matches of low similarity from a cas file. The limits for low similarity are expressed as a minimum sequence similarity required over a minimum fraction of the read length. These parameters are set using the '-s' and '-l' options, respectively. The limits work just like for clc_ref_assemble_long. For paired reads, if only one read fails to pass the filter criteria, only this one read is removed by default. Use the '-p' option to only include paired reads in the output in such cases.

For further details see Appendix A.

7.3 The clc_extract_consensus Program

This program makes it possible to extract a consensus sequence from a read mapping. It operates on a cas file produced by the reference assembly programs.

The clc_extract_consensus program makes a consensus sequence file containing all the original data but with changes made to the references to reflect the information recovered by the read mapping. The program can furthermore be run such that it outputs a list of regions in the reference with zero coverage in the read mapping. The following options control how the consensus sequence is constructed.

The -r option determines how conflicts between reads should be resolved in the consensus sequence. The default is a simple vote (the majority of the read bases determine the consensus base), but it is also possible to output ambiguity characters (this will cause sequencing errors to be reflected in the consensus sequence) or to report the positions as containing unknown bases (using N's).

The -c option can be used to specify the minimum coverage that will make a difference with respect to the reference valid and report it. The default for this option is 2.

The -z option controls how positions with zero coverage are reported in the consensus sequence. The default is to report the base from the reference sequence, but it is also possible to report the position as an unknown base (N) or to simply remove the position.

Using the -i option will make the program ignore all indels completely while constructiong the consensus sequence.

The following three options control the input and output of the program

Use the -a option to specify the input cas file to be analyzed. This option is a required option.

Use the -o option to specify where the output fasta file should be placed. This option is a required option.

Finally, use the -w option if you want a list of zero coverage regions output to sceeen.

See Appendix A for further details.

7.4 The clc_join_mappings Program

Using this program, it is possible to join two or more cas mapping files into one. It is sometimes convenient to perform read mappings on different sets of reads as independent runs. These runs can then be joined later with the clc_join_mappings program. It is a requirement that the mappings have exactly the same reference sequence files in the same order to join them.

Options for clc_join_mappings can be found in Appendix A.

7.5 The clc_submapping Program

The clc_submapping program allows the user to make a new mapping containing only part of the original maping.

Options for clc_submapping can be found in Appendix A.

7.5.1 Specifying Mapping Files

The '-a' options specifies the input assembly and the '-o' option specifies the output assembly.

7.5.2 Extracting a Subset of Reference Sequences

The '-s' option is used for making a new mapping with only matches to a single reference sequence. The '-d' option makes a new mapping with only matches to the reference sequences of a single file. The sequence or file must be specified as its number in the list of reference sequences or files in the input. You can use clc_mapping_info to see the contents of the input mappings is needed.

These options are useful when working with a large mappings such as the human genome. Extracting sub mappings for each chromosome may make it easier to work with.

7.5.3 Extracting a Part of a Single Reference Sequence

If a single reference sequence is specified using the '-s' option or if the input mapping contains only a single reference sequence, the '-b' option may be used to specify a position range to extract. The output mapping will then only contain matches to this specific region. If a match is partially located in the region, only the part of the match inside the region is kept.

This option is useful for studying a particular section of a long reference sequence. It could, for example, be a single gene in the whole human genome.

7.5.4 Extracting Only Long Contigs (Useful for De Novo Assembly)

If you map reads against contigs created by de novo assembly, it can be useful to extract the mappings of the longest contigs only. This can be done using the '-r', specifying the minimum length of the reference sequence.

7.5.5 Extracting a Subset of Read Sequences

Using the '-q' option, you can make a mapping file with only the reads from one of the read files. The read file is specified by its number in the input mapping file. If reads are interleaved, the output file will refer to the two interleaved files instead of just one file.

This is for example useful if you wish to study how the reads from a particular experiment behaved although the full mapping contains reads from several experiments.

7.5.6 Other Match Restrictions

The '-u' option ensures that only uniquely placed matches are kept. The '-l' option specifies a minimum length of a read sequence that must be part of its match alignment for it to be kept. Mismatches within the alignment does not affect the length measurement.

7.5.7 Output Reference File

By default, the output mapping refers to one or all of the reference files in the input cas file. It refers to just one of the files when it has been selected using the '-d' option or when a single reference sequence has been selected with the '-s' option.

If the '-g' option is used, an output file is made with only the reference sequences of the output mapping. The new mapping automatically refers to this reference sequence file. This is typically useful when selecting only a single reference sequence and the input alignment contains many reference sequences in the same file. That way the output only contains the relevant reference sequence instead of many references with no matches. It makes the output easier and faster to work with.

If a position range was specified, the output reference file only contains these positions.

7.5.8 Output Read File

By default, the output refers to one or all of the read files in the input. It refers to just one of the files when it has been selected using the '-q' option.

Using the '-f' option, a new read file is made instead containing only the reads that match. The output automatically refers to this new read file instead of the originals.

This is very useful when making a sub mapping that only covers a small part of the original reference sequences. That way a much smaller number of reads come into play when working with the sub mapping, making subsequent analyses more efficient.

When the reads are from a paired experiment, the read mapper expects read one to pair with read two, read three to pair with read four, etc. If one read out of a pair is removed with the clc_submapping program, the paired read order is disrupted. Because of this, the '-p' option should be used when the reads are from a paired experiment. It works by retaining reads that do not match the clc_submapping criteria if the counterpart does match the criteria. Without the '-p' option, the read file will contain no unassembled reads, but with this option some reads may be unassembled because the other member of their pair is part of the assembly.

7.5.9 Handling of non-specific matches

If an assembly contains non-specific match reads and a sub mapping is made from it, the non-specific matches will still be marked as such even if there is only a single place they match in the chosen subset of the reference sequences. The reason for this is that the clc_submapping program is meant to make it simpler to study a small region of a large mapping, so the original characteristics of the larger mapping are kept.

7.6 The clc_unmapped_reads Program

This program extracts the unmapped read sequences from a mapping. They are output in fasta format. By default the only output sequences are the ones that do not match at all. Using the options it is also possible to output the unaligned ends of reads. A minimum length of unmapped sequences can also be specified.

This program is useful for investigating the sequences that were not part of the expected reference sequences used in a previous mapping. Sometimes, performing de novo assembly on these unmapped reads may be useful to determine their source. It could, for example, be mitochondrial DNA or vector sequence contamination. See Appendix A for further details.

7.7 The clc_unpaired_reads Program

Create a file containing the reads that mapped, but where the read pair did not meet the requirements to be considered an intact pair within the mapping. Reasons for failing to be considered an intact pair include:

- reads of the pair mapped with incorrect relative orientations
- reads of the pair mapped at a distance outside the expected range
- reads of the pair mapped to different reference sequences
- one of the two reads of the pair did not map to any of the references

Further details can be found in Appendix A.

7.8 The clc_agp_join Program

When using the -f option in the de novo assembler for outputting scaffold annotations in AGP format, scaffolded contigs are output as individual contigs and not as a single scaffold with N's inserted in between contigs. The AGP file which is generated contain information on scaffolded contigs and the size of gaps that separate contigs in a scaffold. This program takes a list of contigs and an AGP file as input and output a list of contigs where each contig represents a scaffold where contigs are separated with a number of N's corresponding to the gap size. That is, the output of this program is identical with the default output of the de novo assembler.

Further details can be found in Appendix A.

Chapter 8

Sequence preparation tools

8.1 The clc_adapter_trim Program

Trims adapters from sequences.

Many sequencing technologies may leave whole or partial adapter or linker sequences in the reads for various reasons. The clc_adapter_trim program is used to find and remove such adapters from the reads.

The clc_adapter_trim tool identifies likely adapters in reads and removes them. To account for sequencing errors known adapter sequence are aligned with each read. Matching positions in these alignments score 1, while each mismatch costs 2 and each gap costs 3. By default, a region that aligns with a score of at least 10 is considered a possible adapter region. The -c option can be used to change the default score threshold of 10.

By default, the clc_adapter_trim tool trims bases towards the 3´ end of reads using the following approach:

- For any read with only one region scoring equal to or higher than 10, that region is considered the be an adapter. The adapter region and all bases towards the 3´ end are removed.
- For any read where there is no alignment to a known adapter sequence scoring 10 or greater, the 3´ end of the read is checked for any possible sign of adapter. If found, such bases will be removed. For example if a single nucleotide at the end of a read is identical to the first nucleotide of the adapter it will be removed since it may have come from an adapter. The end match is defined as the longest match at the end of the read having a non-negative score when aligned to the adapter.
- For any read with more than one region aligning with a score equal to or higher than 10, the region closest to the 3´ end is considered to be an adapter and removed along with any bases towards the 3´ end.

With the -e option it is possible to change the behavior so the reads are trimmed towards the 5´ end of reads, rather than the 3´ end. In this case, the conditions described above are the same, with the directionality of the actions reversed.

The clc_adapter_trim program allows fine control over the behavior of the tool. For example,

- Should read sequences before or after the adapter be kept. The default action is to keep the sequence before the adapter, but this can be altered using the -e option.
- Which reads should be kept. For example, reads where adapter was found are kept by using the -t or -f options, and reads where the adapter was not found are kept by using the -u or -g options.
- Which adapter sequences should be searched for. One or several adapter sequences can be used, using the -a and -d options, and for paired data, different adapters can be used for the first and second reads in the pairs by using the -j and -k options.

For adapter sequences given with the -a, -j or -k options, the reverse complement of the adapter sequences is automatically added to the list of adapters to search for.

Further details can be found in Appendix A.

8.2 Quality trimming

The clc_quality_trim program is used to trim sequencing reads for low quality. The idea is to trim the reads at one or both ends so that only a region of high-quality bases are left. This is done by specifying a threshold value (using the '-c' option) for low-quality base calls. The default value is 20, which means that quality scores below 20 are marked as low quality. Since it is often not desirable to discard a high-quality region because of one isolated low-quality base, you can specify the fraction of low-quality bases allowed in a region using the '-b' option. The default value is 0.1 meaning that up to 10 % low-quality bases are allowed. The trim algorithm will then, for each read, find the longest region that fulfills these thresholds. Note that in some situations the full read will be discarded if no good quality regions can be found.

For paired data, two separate files are specified as output: one for the intact pairs (use the -p option for this output file) and one for the single reads whose mate was discarded during trimming (use the -o option for this output file).

There are other options to refine the quality trimming even more (see Appendix A).

8.2.1 Fastq quality scoring

The clc_quality_trim program requires an ASCII offset value for the quality scores in fastq formatted files. The default behavior is to automatically detect this offset from the input data, but in case that this fails it is also possible to set the offset manually. To set the offset correctly you need to know the pipeline used to generate the original data and then set the offset accordingly using the -f option. Some examples of standard offset values for Illumina pipelines are:

- NCBI/Sanger & Illumina Pipeline 1.8 and later: 33
- Illumina Pipeline 1.2 and earlier: 64
- Illumina Pipeline 1.3 and 1.4: 64
- Illumina Pipeline 1.5 to 1.7: 64

For example, the following command would stipulate a minimum quality value of 10 and an offset of 33. The program will return the longest region for each read that fulfills these criteria. Reads that do not have regions that make the criteria cutoffs will be discarded.

```
quality_trim -r smallfile.fastq -c 10  -f 33 -o smallfile_trimmed.fasta
```

Note that the cutoff can be a negative value for the Solexa pipeline.

8.3 The clc_remove_duplicates Program

clc_remove_duplicates is designed to filter out duplicate reads, retaining just a single representative of any set of identical reads in the output. It is designed for use in situations where certain regions are represented by a higher numbers of reads than reflects the relative underlying biological abundance, such as can occur using PCR amplification. The challenge is to achieve this without removing reads that represent the true biological situation, e.g. repeat regions.

When to use and when not to use this tool

We believe this tool should be rarely needed in practice and we recommend it is only used in situations where there is a strong suspicion that duplicate reads are present and that they would affect the quality of a de novo assembly. We note that NGS reads often contain leftover adapters and sequencing artifacts, and that these can cause a massive increase in both memory and time consumption of this tool. We thus do not recommend that this tool in included as a fixed step in assembly pipelines.

This tool is also not recommended for use with for any data where the start of a large number of reads is expected to be the same location on a genome, for example exome data, amplicon data, RNA-Seq data, or Chip-Seq data. It should also not be used with data that will be used for variant detection downstream.

Recommendations when running this tool

- clc_remove_duplicates should be run after adapters have been trimmed from the reads.
- Data should *not* be trimmed based on quality before running this tool.

clc_remove_duplicates initially looks for identical words at the beginning of reads. Thus, if adapters are present, many reads will be falsely identified as duplicates. Conversely, if reads have been trimmed based on quality, true duplicates may not be identified as they may have different start points after trimming.

What is a duplicate read?

An example of true duplicate reads can be seen in figure 8.1, where the reads have been mapped to a reference sequence. The duplicates can be easily seen: they map to the same position on the reference with the same mapping orientation, and there is a sudden rise in coverage at that point.

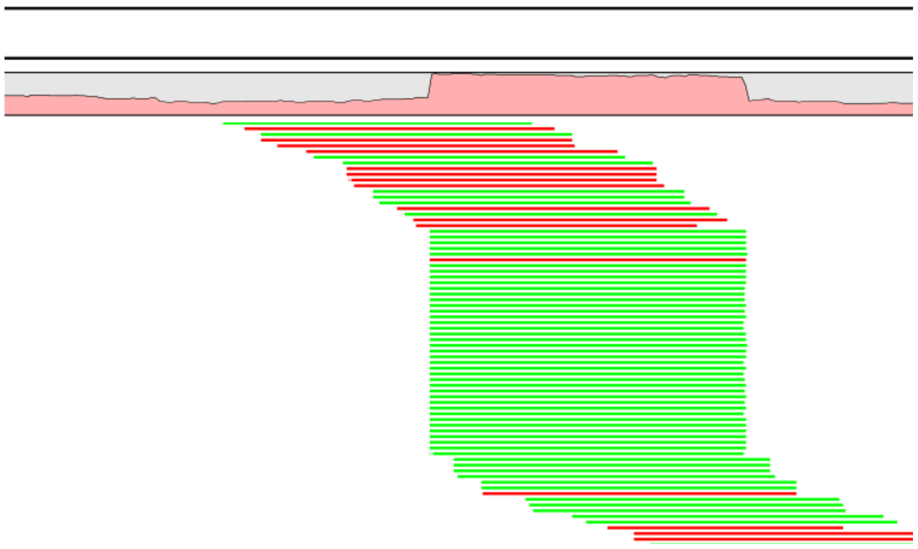


Figure 8.1: *Duplicate reads in this mapping are easily identified: they map with the same start position and in the same orientation (all colored green, showing they map to the forward strand of the reference). The region they map to has an unpectedly high coverage.*

In a data set without duplicate reads, there are still fluctuations in coverage, but a large number of reads do not start at exactly the same position with identical orientation. An example is shown in figure 8.2.

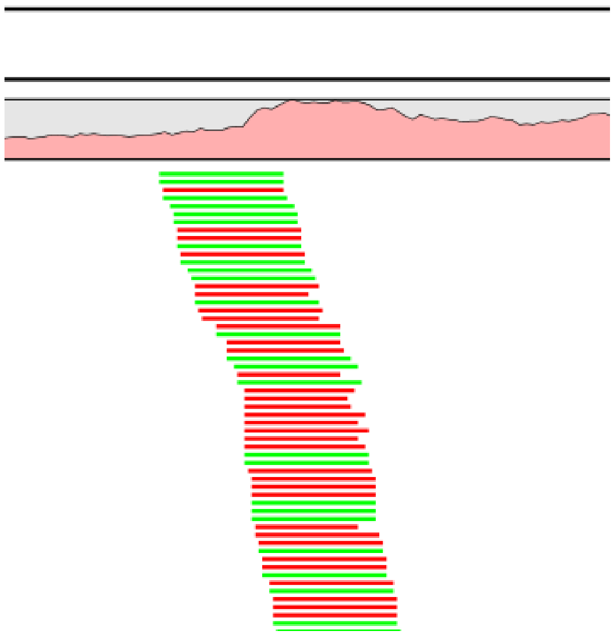


Figure 8.2: *A rise in coverage is seen in this region, but this does not appear to be due to duplicate reads.*

Looking for neighbors

clc_remove_duplicates works directly on the sequencing reads to identify duplicates by looking for "neighboring" reads, that is, reads that share most of the sequence but with a small offset. These are used to determine whether there is generally high coverage for this sequence. If there is not, the read in question will be marked as a duplicate.

For certain sequencing platforms such as 454, reads have varying lengths. This is taken into account by the algorithm.

Sequencing errors in duplicates

clc_remove_duplicates accounts for sequencing errors when it identifies duplicate reads. This is done by defining the limits of how different reads can be and still be considered duplicates.

Reads can be considered duplicates if:

- They share enough common sequence: single reads share a common sequence of at least 20 bases at the start, or at any of four other regions distributed evenly across the read. For paired reads, the length of common sequence required is 10 bases.
- The remainder of the read has an alignment score above 80% of the optimal score, where the optimal score is what a read would score if it aligned perfectly to the consensus for that group of duplicates.

An illustration of the problem these checks are addressing: If a set of reads contained 100 duplicates of a particular 100bp read, and ther was a random 0.1 % probability of a sequencing error, 10 of those duplicates would, on average, contain an error. Without accounting for this, only the 90 identical reads would be removed, leaving the 10 duplicate reads with errors in the output.

Paired data

For paired reads, we check whether the sequences in different pairs mapping in a given region have their first 10 base pairs identical. If they do, they are marked as potential duplicate reads. (We have found this route works better than a statistical route in the case of paired data.) We then check our proposed list of duplicates to look for false positives. We also do a check for closely related variants of those sequences we now believe to represent duplicate reads.

The algorithm also takes sequencing errors into account when filtering out paired data.

Known limitations clc_remove_duplicates has a limitation when there are duplicate reads representing several alleles. The algorithm will identify if there are duplicate reads to be removed, but it is not able to distinguish between sequencing errors and true variation in the reads. So if you have a heterozygous SNP in such an area, you risk that only one of the alleles will be represented in the data after running this tool.

8.3.1 Example of duplicate read removal

The following command outputs all reads to `coli_reads_nodup.fa` that are not identified as duplicates from the paired reads contained in `coli_reads_1/2.fa` and `coli_reads_2/2.fa`.

```
clc_remove_duplicates -p -r -i coli_reads_1/2.fa coli_reads_2/2.fa
                   -o coli_reads_nodup.fa
```

The program runs only in a single thread and for large data set it would be convenient to run multiple instances at the same time for each data file.

8.4 The clc_sample_reads Program

This tool extracts a subset of reads where the size of the subset is a percentage of the input size. Sampling is done in a pseudo-random way, which does not guarantee that the extracted subset comprises an exact percentage of the input reads. The input reads can be provided in both interleaved and non-interleaved format and reads marked as paired are kept together.

Read sampling can be useful for reducing coverage of datasets with a very high coverage (>500x coverage) in preparation for a de novo assembly. A reduction in coverage makes the assembly run faster and reduces the chance of having overlapping errors in the reads, thus increasing the assembly quality.

See Appendix A for full usage information.

8.5 The clc_sort_pairs Program

The clc_sort_pairs program takes two SOLiD read files, or two Ion Torrent read files, as input and generates as output a file containing paired reads and a file containing unpaired reads. Here the read names are used to sort the reads. This tool is necessary because pairing of the reads in these cases is based on the read names rather than just the position within the file, as would be the case for Illumina data. That is, paired reads for these data types need to be sorted, and paired reads separated from single reads.

To properly handle the input sequence data, the read names within the files must match certain patterns. These are:

Ion Torrent: [anytextinfo]:[number]:[number]

Solid: [number]_[number]_[number]_[R3/F3/F5/F5-P2/F5-BC]

In the case of Solid data, reads in one file of the pair should end in one of the patterns above, and reads in the other file of the pair should end with one of the other patterns. For example, one file might contain reads with names ending in R3, while reads in the other file have names ending in F5, or reads in one file might contain names ending in F3, while reads in the other file have names ending in F5, and so on.

Reads within a given file must be named consistently. That is, if a read has a name ending with a particular pattern, for example F5, then all reads in that file must have names ending in F5.

Please note that in the case of Solid data, the following combinations for the read names in a pair of files are **not allowed**:

F3 + F3

R5 + F5
R3 + R3
R3 + F5
F5 + R3

As mentioned in the Input Data section earlier in the manual, the full sequence of any read containing one or more . symbols, present in a .csfasta format file, will be converted to contain only N characters.

Further details are provided in Appendix A.

8.6 The clc_split_reads Program (for 454 paired data)

The 454 sequencing technology can produce paired read files where the two paired read fragments are contained within the same read, separated by a linker sequence. The linker may be placed anywhere in the read or even outside the read, so not all the reads will necessarily contain a pair. The clc_split_reads program finds the linker sequence and creates two new files, one with paired reads and one with unpaired reads.

Like adapter regions, linker regions may contain sequencing errors. With this in mind, the clc_split_reads tool identifies likely linker sequences by initially carrying out an alignment between the known linker sequence and each read. The alignment is global in terms of the linker and local in terms of the reads. That is, the whole linker must align to part of the read. Matching positions in these alignments score 1, while each mismatch costs 2 and each gap costs 3. For alignments found at the ends of reads, any non-matching linker bases that extend beyond the end of the read are not penalized.

By default, a region that aligns with a score of at least 10, or the length of the linker if less than 10, is considered a good enough match to identify a linker region.

If a match to the linker sequence with a score between 0 and 9 is found, the read will still be split, but in this situation the following will happen:

- The two parts of the read that have just been split are put into the singles list (not the paired list). The reasoning behind this is that since the linker did not match with a good enough score, the match location identified might not have been correct. If this was the case, marking such split sequences as a pair would mean that the paired distance information would be used in a de novo assembly, and this could end up being misleading within the assembly process.
- Any linker matches identified at the end of the read will also be trimmed. This extra trimming stage is carried out due to the possibility that the internal linker match identified might not have been correct.

The following situations are particularly detrimental to de novo assembly, and the clc_split_reads program tries to ensure they are avoided:

- Reads contain some remaining section of the linker sequence
- Reads are categorized as a pair when they should not be

In some cases, the start or end of a read is in the middle of the linker. In such cases, the linker sequence is still removed, and the read is put into the file with unpaired reads. If only very few nucleotides of the linker overlap with the read, they are also removed, even though they may not come from the linker. In the case where only a single nucleotide at the start and/or end of the read may come from a linker, it is removed. The rationale is that it is better to discard a few nucleotides and be sure there is no adapter sequence left, since remaining linker sequence is problematic for de novo assembly.

The '-m' option can be used to specify the minimum read length. Only reads this long or longer will be reported. The default value is 15. This becomes important when the linker is close to the start or end of the read, and only a small fragment is left on one side of the linker. If that small fragment is below the specified minimum length, it is discarded along with the linker. The remaining part of the read will be written to the unpaired file.

Further details of the options for this tool are provided in Appendix A.

8.7 The clc_overlap_reads Program

In cases where paired end library preparation methods use a relatively short fragment size, some read pairs will overlap. These overlapping reads can be handled as standard paired-end data.

However, in some situations it can be useful to merge the overlapping pairs into a single read. The benefit is that you get longer reads, and that the quality improves (normally the quality drops towards the end of a read, and by overlapping the ends of two reads, the consensus read now reflects two read ends instead of just one).

This joining of overlapping reads can be done using the clc_overlap_reads program. It aligns the ends of each read within pairs to see if there is evidence that they overlap. If the alignment of these read ends is relatively good, the reads are joined into one read and put in an output file for single (joint) reads. If there is no evidence of the reads overlapping, the original pair of reads is put in an output file for paired reads.

The nucleotides in the overlapping region of a joint read are assigned a quality score of 40 (very high quality) if the two reads agree on the nucleotide. Otherwise the nucleotide with the highest quality is chosen and its quality score is retained.

By default, the alignment between the ends of two reads must have a minimum length of 10 positions and a minimum similarity of 90% for the reads to be considered overlapping. These parameters can be adjusted using the various options for the program. The default is that the first read of each pair is a forward read and the other one is a backward read. This can also be adjusted.

In cases where reads contain an adapter sequence at the read ends, the adapter sequence needs to be removed (see figure 8.3). The option *–autoadapter* enables free gaps at the read ends and leaves out unaligned read ends, i.e. any adapter sequences will not be included in the result.

Further details of the overlap reads options are provided in Appendix A.

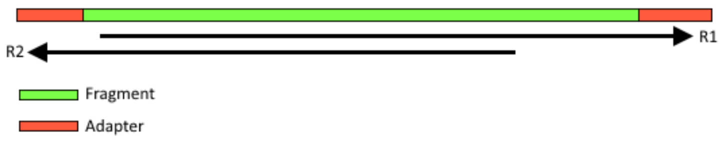


Figure 8.3: *Illustration of two reads originating from a fragment which is shorter than the read length. Both reads contain adapter sequences in the read ends which needs to be removed when overlapping the reads.*

Chapter 9

Format conversion tools

9.1 The clc_cas_to_sam Program

This tool converts a cas format file to sam or bam format format file.

The **clc_cas_to_sam** program takes a cas file as input and produces a corresponding SAM file or BAM file. The format generated depends on the filename you choose. If you choose an output file name with the suffix **.sam**, the output format will be SAM. If you choose an output file name with the suffix **.bam**, the output format will be BAM.

Please note that the read file(s) that you used in generating the cas file must be present in the same relative location to the cas file as they were when you ran the mapping. This is because, unlike cas format files, SAM and BAM files include all the read data. Thus the read data needs to be present in order to make a valid SAM or BAM file. This also means that the SAM or BAM files created will generally be substantially larger than the cas file they were generated from.

When writing a sam/bam file, each read is given a read mapping quality (MAPQ) score of either 0 or 60. The value 60 is given to reads that mapped uniquely. The value 0 is given to reads that could map equally well to other locations besides the one being reported in the BAM file.

The SAM or BAM file created using the clc_cas_to_sam tool is not sorted or indexed. These steps can be necessary for some types of downstream processing and can be done using the samtools sort program (see `http://samtools.sourceforge.net/`).

Like cas format files, SAM and BAM files do not contain the reference sequence data.

Further details of the command line options for this tool are provided in Appendix A.

Further details about the SAM and BAM formats can be found at `http://samtools.sourceforge.net/`.

9.2 The clc_sam_to_cas Program

This tool converts a read mapping in sam or bam format to a cas format file.

The **clc_sam_to_cas** tool converts SAM or BAM files to cas format. Like cas format files, SAM and BAM files do not contain the reference sequence data, so the reference sequences need to be provided when running this command. This is so that the required information can be generated for the cas file.

You also need to provide a destination for the sequencing read data to be written to when running this program.

Further details of the command line options for this tool are provided in Appendix A.

9.3 The clc_convert_sequences Program

The primary purpose of this tool is to convert sequences to fasta or fastq format. However, it can currently be used for a variety of sequence conversion related functions. These include:

- Convert from one format to a fasta format file or to a fastq format file.
- Merge separate forward and reverse read files into a single, interleaved paired data file.
- Remove sequence names from within a file (to save space).
- Create a fastq format file from (separate) sequence and quality files.

Input formats supported for some or all functionality:

- fasta
- fastq
- genbank
- sff
- csfasta

Export formats are fasta or fastq.

See Appendix A for full usage information.

Appendix A

Options for All Programs

Please find the list of options for all programs in the online version of the user manual at `http://resources.qiagenbioinformatics.com/manuals/clcassemblycell/current/index.php?manual=Options_All_Programs.html`

Appendix B

Third Party Libraries

The CLC Assembly Cell for windows includes a number of third party libraries.

Please consult the files named NOTICE and LICENSE in the assembly cell installation directory for the legal notices and acknowledgements of use.

For the code found in this product that is subject to the Lesser General Public License (LGPL) you can receive a copy of the corresponding source code by sending a request to our support function.

Bibliography

[Gnerre et al., 2011] Gnerre, S., Maccallum, I., Przybylski, D., Ribeiro, F. J., Burton, J. N., Walker, B. J., Sharpe, T., Hall, G., Shea, T. P., Sykes, S., Berlin, A. M., Aird, D., Costello, M., Daza, R., Williams, L., Nicol, R., Gnirke, A., Nusbaum, C., Lander, E. S., and Jaffe, D. B. (2011). High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences of the United States of America*, 108(4):1513–8.

[Li et al., 2010] Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., Li, S., Yang, H., Wang, J., and Wang, J. (2010). De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20(2):265–72.

[Zerbino and Birney, 2008] Zerbino, D. R. and Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res*, 18(5):821–829.

[Zerbino et al., 2009] Zerbino, D. R., McEwen, G. K., Margulies, E. H., and Birney, E. (2009). Pebble and rock band: heuristic resolution of repeats and scaffolding in the velvet short-read de novo assembler. *PloS one*, 4(12):e8407.