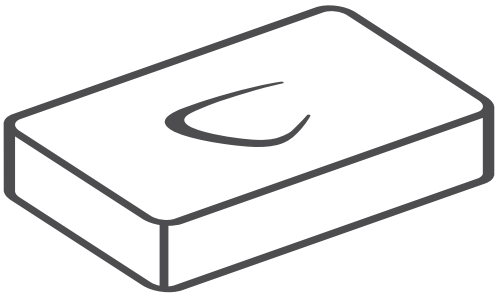


CLC Assembly Cell

User manual

User manual for

CLC Assembly Cell 3.0.1

Windows, Mac OS X and Linux

March 16, 2010

CLC bio
Finlandsgade 10-12
DK-8200 Aarhus N
Denmark

Contents

1 Introduction **4**

1.1 Installation 4

1.1.1 Windows installation 4

1.1.2 Mac installation 5

1.1.3 Linux installation 5

1.1.4 Using a license server 6

1.2 Notation 6

1.3 Overview of Commands 6

2 System requirements **8**

2.1 Operating system platforms 8

2.2 Supported Intel CPU architectures 8

2.3 Supported AMD CPU architectures 9

2.4 How do I determine my CPU type? 9

2.4.1 CPU info: Windows XP 9

2.4.2 CPU info: Mac OS X 9

2.4.3 CPU info: Linux 10

2.5 Disk space 11

3 Cas File Format **12**

3.1 Sequence Data 12

3.2 Binary Format 12

3.3 Contained Data 12

3.4 Limitations 13

4 Command Line Options **14**

4.1 Input Files . . . 14
4.2 Paired Ends . . . 15
4.3 Interleaved Read Files for Paired Ends . . . 16

5 Reference Assembly **18**
5.1 Non-specific matches . . . 18
5.2 Placement of Read Pairs . . . 19
5.3 Scoring Schemes . . . 19
5.4 Short Read Reference Assembly . . . 20
5.5 Long Read Reference Assembly . . . 21

6 Color space **22**
6.1 Sequencing . . . 22
6.2 Error modes . . . 23
6.3 Assembly in color space . . . 23
6.3.1 Score limit . . . 26
6.4 File formats . . . 26

7 *De novo* assembly **28**
7.1 How it works . . . 28
7.2 Specific characteristics of CLC bio's algorithm . . . 30
7.3 SOLiD data support in de novo assembly . . . 31
7.4 Other options . . . 31

8 Working with Assemblies **33**
8.1 The sequence_info Program . . . 33
8.2 The assembly_table Program . . . 33
8.3 The assembly_info Program . . . 35
8.4 The filter_matches Program . . . 37
8.5 The sort_pairs Program . . . 37
8.6 The split_sequences Program . . . 37
8.7 The change_assembly_files Program . . . 38
8.8 The join_assemblies Program . . . 38
8.9 The sub_assembly Program . . . 38

8.9.1 Specifying Assembly Files . . . 38
8.9.2 Extracting a Subset of Reference Sequences . . . 38
8.9.3 Extracting a Part of a Single Reference Sequence . . . 39
8.9.4 Extracting a Subset of Read Sequences . . . 39
8.9.5 Other Match Restrictions . . . 39
8.9.6 Output Reference File . . . 39
8.9.7 Output Read File . . . 39
8.9.8 Handling of non-specific matches . . . 40
8.10 The find_variations Program . . . 40
8.11 The unassembled_reads Program . . . 40

9 Assembly Viewer 41

A Options for All Programs 45

A.1 Options for assembly_info . . . 45
A.2 Options for assembly_table . . . 46
A.3 Options for change_assembly_files . . . 46
A.4 Options for clc_assembly_viewer . . . 47
A.5 Options for clc_novo_assemble . . . 47
A.6 Options for clc_ref_assemble_long . . . 48
A.7 Options for clc_ref_assemble_short . . . 50
A.8 Options for filter_matches . . . 51
A.9 Options for find_variations . . . 52
A.10 Options for join_assemblies . . . 53
A.11 Options for sequence_info . . . 53
A.12 Options for sort_pairs . . . 53
A.13 Options for split_sequences . . . 54
A.14 Options for sub_assembly . . . 54
A.15 Options for unassembled_reads . . . 55
A.16 Options for tofasta . . . 56

Chapter 1

Introduction

This document describes the CLC Assembly Cell - CLC bio's command line tools for performing sequence assembly and for basic analysis of such assemblies. If more advanced analyses of assemblies are desired, the CLC Genomics Workbench can be used (see http://www.clcbio.com/genomics). You can either import assembly files to the Workbench or make the assemblies directly within the Workbench. The Workbench uses the same assembly algorithms as the CLC Assembly Cell.

1.1 Installation

1.1.1 Windows installation

1. Download the distribution from http://www.clcbio.com/download_assembly_cell.
2. Unzip the files in the zip-file to a folder on your computer.
3. Double-click the file *host_info.bat*.
4. This program will generate an email to license@clcbio.com with information about your computer. This is used to create a license key.
5. Send the email.
6. When we have received your email, we will generate a license key file which is sent back to you by email.
7. Save the license key file (*.lic) in either the
   - *working directory*,
   - `$ALLUSERSPROFILE\CLC bio\Licenses` or
   - `$APPDATA\CLC bio\Licenses`

   or you can save it in another folder and specify this location in the environment variable called *CLCBIO_LICENSE_PATH*.
8. You are ready to use the CLC Assembly Cell.

1.1.2 Mac installation

1. Download the distribution from http://www.clcbio.com/download_assembly_cell.
2. Unzip the files in the zip-file to a folder on your computer.
3. Double-click the file *host_info*. This will generate one or more 16-digit numbers.
4. Copy the first number into an email and send it to license@clcbio.com. This number is used to create a license key.
5. When we have received your email, we will generate a license key file which is sent back to you by email.
6. Save the license key file (*.lic) in either the
   - *working directory*,
   - /Library/Application Support/CLC bio/Licenses or
   - $HOME/Library/Application Support/CLC bio/Licenses

   or you can save it in another folder and specify this location in the environment variable called *CLCBIO_LICENSE_PATH*.
7. You are ready to use the CLC Assembly Cell.

1.1.3 Linux installation

1. Download the distribution from http://www.clcbio.com/download_assembly_cell.
2. Unzip the files in the zip-file to a folder on your computer.
3. Run the file *host_info*. This will generate one or more 16-digit numbers.
4. Copy the first number into an email and send it to license@clcbio.com. This number is used to create a license key.
5. When we have received your email, we will generate a license key file which is sent back to you by email.
6. Save the license key file (*.lic) in either the
   - *working directory*,
   - /etc/clcbio/licenses or
   - $HOME/.clcbio/licenses

   or you can save it in another folder and specify this location in the environment variable called *CLCBIO_LICENSE_PATH*.
   - If you are using tcsh or a similar shell, the command for setting the environment variable would be *setenv CLCBIO_LICENSE_PATH /path/to/license*
   - If you are using bash or a similar shell, the command for setting the environment variable would be *export CLCBIO_LICENSE_PATH=/path/to/license*
7. You are ready to use the CLC Assembly Cell.

1.1.4 Using a license server

If you are using a license server rather than stand-alone licenses for the CLC Assembly Cell, the licensing steps are a little different: The *host_info* program included in the distribution should be run on the computer where the license server is to be installed (if the license server is running a different operating system, you need to download the full distribution, even though you only need the host_info program). The license that you will receive from CLC bio is this valid for that computer.

In order to make the CLC Assembly Cell contact the license server for a license, you need to create a text file in the *working directory* 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 server set-up.

You can read more about the license server at the bottom of http://www.clcbio.com/usermanuals.

1.2 Notation

We distinguish between *reference assembly* where the target sequences are known and *de novo assembly* where the goal is to find the sequences that the reads came from. Other words for reference assembly used outside this document are alignment and mapping. De novo assembly is sometimes just called assembly, but in this document the general term *assembly* covers both reference assembly and de novo assembly.

To keep notation consistent, the sequences that reads are aligned to are always called reference sequences. This is the case even if the sequences were formed in a de novo assembly process.

1.3 Overview of Commands

The following commands are available for creating assemblies:

clc_ref_assemble_short Short read reference assembly.

clc_ref_assemble_long Long read reference assembly.

clc_novo_assemble De novo assembly.

Assembly files are in a special format called cas files because the extension is .cas. The following commands are available for analyzing these files as well as sequence files:

sequence_info Print overview of fasta file.

assembly_info Print overview of assembly.

assembly_table Print details of assembly.

filter_matches Removes matches of low similarity.

Apart from printing the contents of cas files in different ways, it is also possible to perform various operations on them using these commands:

change_assembly_files Change the sequence file names in an assembly file.

join_assemblies Join a number of assemblies to the same reference.

sub_assembly Extract a part of an assembly

find_variations Find the positions where the reads differ from the reference sequences.

unassembled_reads Extract unassembled reads from an assembly.

For handling special cases in the file formats, there are two dedicated conversion programs:

sort_pairs For converting paired SOLiD csfasta files.

split_sequences Removing linker from 454 paired end data and extracts pairs.

Finally, there is a program to convert the different read file formats into fasta (fastq, sff, csfasta and genbank):

tofasta Converts fastq, sff, csfasta and genbank into fasta.

Chapter 2

System requirements

2.1 Operating system platforms

The system requirements of CLC Assembly Cell are these:

- Windows XP, Windows Vista or Windows 7
- Mac OS X 10.3 or newer
- Linux: Redhat or SuSE
- CPU architectures as described below

2.2 Supported Intel CPU architectures

The Cell uses the SSE2 extension of the Intel CPU instruction set. It was introduced in 2001.

Intel uses a number of different CPU microarchitectures with different performance characteristics. The recent ones are:

- The NetBurst microarchitecture:
  - Pentium 4 (670, 661, 660, 651, 650, 641, 640, 631, 630, 551, 541, 531, 524, 521)
  - Pentium D
  - Xeon (7150N, 7140M, 7140N, 7130M 7130N, 7120M, 7120N, 7110M, 7110N, 7041, 7040, 7030, 7020, 5080, 5063, 5060, 5050, 5030)
- The Pentium M microarchitecture:
  - Pentium M (780, 770, 765, 760, 755, 750, 745, 740, 735, 730, 725, 715, 705, 778, 758, 738, 718, 773, 753, 733J, 733, 723, 713)
  - Pentium Core Solo (T1400, T1300, U1500, U1400, U1300)
  - Pentium Core Duo (T2700, T2600, T2500, T2400, T2300, T2300E, L2500, L2400, L2300, U2500, U2400)

- The Core microarchitecture:
  - Pentium Core 2 Duo (E6700, E6600, E6400, E6300, E4300, T7600, T7400, T7200, T5600, T5500, L7400, L7200)
  - Pentium Core 2 Extreme (X6800, QX6700)
  - Xeon (3070, 3060, 3050, 3040, X3220, X3210, X5355, L5320, L5310, E5345, E5335, E5320, E5310, 5160, 5150, 5148 LV, 5140, 5130, 5120, 5110)

As shown, the Pentium Core processors have the Pentium M microarchitecture, while Pentium Core 2 processers have the Pentium Core microarchitecture.

The highest performance per GHz is with the Core microarchitecture while Pentium M has a lower performance and NetBurst is slightly lower.

2.3 Supported AMD CPU architectures

AMD introduced the SSE2 extension in 2003, so recent AMD architectures are supported and their performance is generally a little better than Intel Pentium M but not as high as the Intel Core microarchitecture.

2.4 How do I determine my CPU type?

If you do not know the type of your CPU, use this guide to find out:

2.4.1 CPU info: Windows XP

- Click **Start**
- Right-click **My computer**
- Click **Properties**

You will now see a dialog similar to the one shown in figure 2.1:

The red circle indicates the CPU information. Check with the list of CPU types above to see if your CPU is supported. If the CPU is not in the list, please send an email to support@clcbio.com with the information from this dialog.

2.4.2 CPU info: Mac OS X

- Click the **Apple** at the upper left corner of the screen
- Right-click **About This Mac**

You will now see a dialog similar to the one shown in figure 2.1:

The red circle indicates the CPU information. Check with the list of CPU types above to see if your CPU is supported. If the CPU is not in the list, please send an email to support@clcbio.com with the information from this dialog.

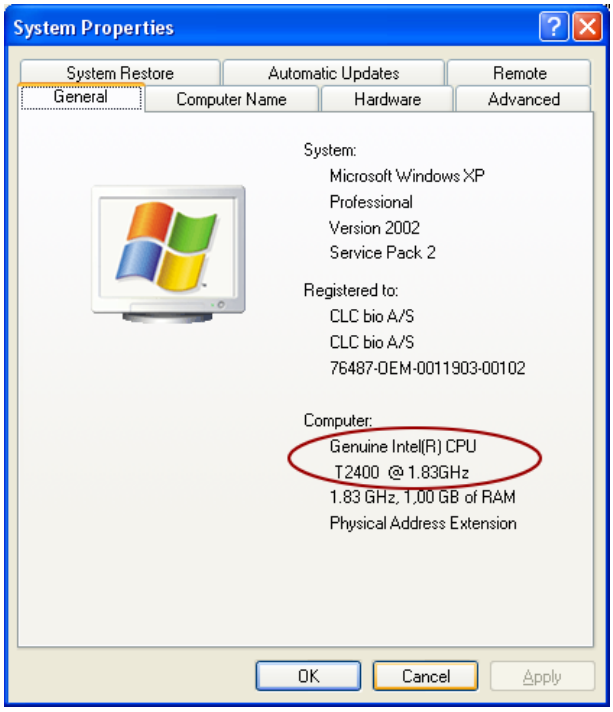


Figure 2.1: *Information about CPU on Windows XP.*

Figure 2.2: *Information about CPU on Mac OS X.*

2.4.3 CPU info: Linux

Enter this:

```
cat /proc/cpuinfo
```

You will now see information about your CPU similar to figure 2.3:

Check with the list of CPU types above to see if your CPU is supported. If the CPU is not in the list, please send an email to support@clcbio.com with the information from this dialog.

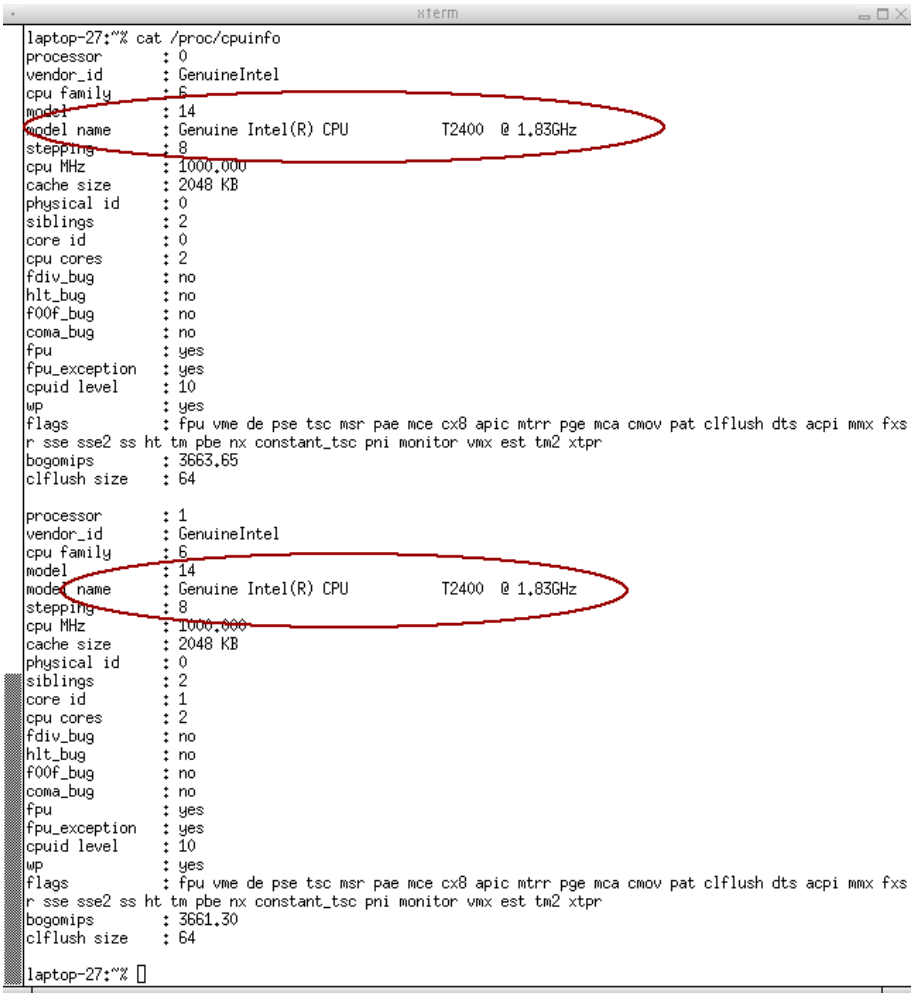


Figure 2.3: *Information about CPU on Linux.*

2.5 Disk space

Data from Next-Generation sequencing machines naturally takes 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 specific in the TMP variable on Windows and TMPDIR on Linux and Mac.

Chapter 3

Cas File Format

With CLC bio's command line assembly tools, the cas file format is used. It is a custom file format made with next generation sequencing data in mind (but works fine for any kind of sequencing data). It is not necessary to know everything about this format to use it, but a few basics will help.

3.1 Sequence Data

The most important thing to notice is that cas files do not contain any sequence data. They only contain data about relations between sequences available in other files. Instead of actual sequence data, the cas files contain the names of the corresponding read and reference sequence files. This approach was chosen to save space. There is no reason to keep all the sequences in two places.

3.2 Binary Format

The cas files are in a binary format. Again, the reason for this is to save space. Due to this design, the size of a cas file is only about 8 bytes per read assembled to the human genome. So a cas file with 100 million Solexa reads of length 35 assembled to the human genome is only about 800 MB in size. This is significantly smaller than assemblies in other file formats.

3.3 Contained Data

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 paired end pair.

3.4 Limitations

As previously noted, cas files do not contain the actual sequences. This means that you have to be careful to include all the files when sending an assembly to someone. You also have to be careful when moving assembly files, since relative file names may not match any more. The program change_assembly_files can be used to change the file names.

There is also a limit of one alignment per read. So a read matching in multiple locations can only have one of these locations described. When assembling short reads to the human genome, some reads may match in over 100,000 locations, so keeping track of all those alignments would be problematic.

If you have a big data set, it would be a good idea to break it up into smaller pieces. The exact limit on when to break up the data depends on the amount of memory on your computer. For an optimal performance on a computer with 32 GB of memory, you should not use more than 100 million reads for one round of assembly. It doesn't mean that you can't assemble more than 100 million reads - it just means that you should do the assembly in several rounds[1]. You can then use the join_assemblies program to join the cas files afterwards, or just parse the output of several cas files.

[1] Usually the sequences come in several files anyway, so it is fairly simple to run the assembly in several rounds

Chapter 4

Command Line Options

This chapter describes some general command line options. More specific options are given in the sections for individual programs (chapters 5–9). Finally, appendix A gives details for all the options for all the programs.

4.1 Input Files

The assembly programs support the following input file formats:

Format	Reads	References
Fasta	+	+
Fastq	+	-
Scarf	+	-
csfasta	+	-
Sff (not paired ends)	+	-
GenBank	-	+

The formats are automatically detected.

The '-d' option indicates that the following files contain reference sequences and the '-q' option indicates that the following files contain read sequences. Both of these options may be used repeatedly. For example:

```
clc_ref_assemble_short -o assembly.cas -d human.gb -q reads1.fasta reads2.fasta
                                                                  -d mito.gb
```

This command assembles the reads in the files `read1.fasta` and `read2.fasta` to the references sequences in the two files `human.gb` and `mito.gb`. The assembly may be done on one read file at a time and then later joined using the join_assembly program.

It is a good idea to include all the reference files in one assembly operation, rather than assembling to different references independently. Consider a reference assembly to the human genome as an example. If reference assembly was performed independently to each chromosome, many reads would not match anything in a given run (because the reads match another chromosome). This results in longer execution time since the reference assembly program then has to look harder for possible matches without any success.

4.2 Paired Ends

It is possible to specify that a read file came from a paired end sequencing experiment. This is specified using the '-p' option which allows any relative orientation of the reads. A typical option would look like this '-p fb ss 100 200', which means the following:

- The first read of a pair is in the forward direction, the second read is in the backward direction ('fb').
- The distance between the reads are measured from the start of the first to the start of the second. Thus, since the second read is reversed, the distance includes both the reads and the sequence between them ('ss').
- The distance between these two starting points is between 100 and 200 positions, both included ('100 200').

The allowed values for the directions are 'ff', 'fb', 'bf', and 'bb'. They mean the following:

	Read		
Code	First	Second	Description
ff	→	→	Both reads are forward.
fb	→	←	Reads point toward each other.
bf	←	→	Reads point away from each other.
bb	←	←	Both reads are backward.

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. This option is probably not going to be very widely used, but is included for the sake of completeness. The 'fb' option is the most typical.

The next question is how to measure the distance between two reads of a pair. This depends on how the sequencing experiment is done. The distance between two reads should not depend on how long the sequencing reaction was run. So if 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 reads 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 end 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_ref_assemble_short -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 end reads.

Note that the sort_pairs and split_sequences program can be used to convert data from SOLiD and 454 systems, respectively, into an intelligible format.

4.3 Interleaved Read Files for Paired Ends

In general, paired end data are expected to be in a single file in the form of two sequences from one pair, then two sequences from the next pair, etc. Some sequencing technologies use separate files for the paired reads. In this case, the '-i' option (for interleaved) can be used followed by the two separate files, one with the first reads of the pairs and one with the second reads.

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

This is identical to:

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

Chapter 5

Reference Assembly

When the reads come from a set of known sequences with relatively few variations, reference assembly is often the right approach to assembling the data. CLC bio offers two programs for reference assembly: clc_ref_assemble_short and clc_ref_assemble_long, which are for short and long reads, respectively.

The short read program can be used for reads of length 55 and less. For short reads, it is possible to make reference assembly with a guarantee of finding all alignment locations for all the reads, given a certain quality threshold. Such a threshold can for example be to find all reads with at most two mismatches. The short read assembly program works under the assumption that many alignments of reads to the reference sequences are without gaps. By default, gapped alignments are also found, but only after ungapped alignment has been tried. Gapped alignments can be completely turned off for improved speed ('-u' option).

The long read program is used when the requirements of the short read program are not met. For long reads, the alignment quality threshold is given as a certain fraction of the read that must match in a certain fraction of its positions. E.g., the threshold may be set at 90 % identity over 50 % of the read length. The long read assembly program works under the assumption that many alignments have gaps, so gapped alignment is always performed.

By default, reference assembly is done with local alignment of reads to a set of reference sequences. The advantage of performing local alignment instead of global alignment is that the ends are automatically removed if there are sufficiently many sequencing errors there. If the ends of the reads contain vector contamination or adapter sequences, local alignment is also desirable.

Note that you can specify also to use global or local alignment for both short and long reads.

The following sections contain some general information about options for reference assembly. This is followed by sections on specific options for short and long read assembly.

5.1 Non-specific matches

In some cases it may not be possible to uniquely assign a read to a specific optimal position in a reference sequence. This for example happens when a part of a sequence is repeated a number of times among the references. A read that falls entirely within the repeat sequence is impossible to place uniquely. Using longer reads or paired end sequencing alleviates the problem, but if the

repeat is long enough, some reads will still be impossible to place uniquely.

The reference assembly programs allow two options for how to treat these non-specific matches: They can either be randomly placed or not placed at all. This is controlled by the '-r' option which has random placement as default. Since non-specific matches can always be removed later, there is usually little reason to change this option.

Note that it is not possible to record all the positions of the reads since this would sometimes lead to very large amounts of results.

5.2 Placement of Read Pairs

Many sequencing technologies allow paired end sequencing of reads. In such experiments, the reads come in pairs with certain restrictions on their relative placement and orientation.

The approach taken for determining the placement of read pairs is the following:

- First, all the optimal placements for the two individual reads are found.
- Then, the allowed placements according to the paired end options are found.
- If both reads can be placed independently but no pairs satisfy the paired end criteria, the reads are treated as independent and not marked as a pair.
- If only one pair of placements satisfy the criteria, the reads are placed accordingly and marked as uniquely placed even if either read may have multiple optimal placements.
- If several placements satisfy the paired end criteria, the read is treated according to the above described option for ambiguously placed reads. The number of places for the reads are reported as the possible number of placements of the whole pair, not the individual reads.

5.3 Scoring Schemes

For both reference assembly programs, the alignments are scored using Smith Waterman alignment with a linear gap cost. A linear gap cost means that an insertion or deletion of length two costs twice as much as an insertion or deletion of length one. This corresponds to individual insertion and deletion events occurring independently, even if adjacent.

The parameters are:

Parameter	Option	Restrictions
Match score	-	Always 1
Mismatch cost	'-x'	Between 1 and 3. Default is 2
Gap cost	'-g'	Between 1 and 3. Default is 3[1]

It is the relative scores and costs that determine an alignment, so multiplying all the scores by a common factor would give the same alignment. Thus, having the match score fixed to one does not significantly reduce the flexibility in the scoring scheme since the other values can be adjusted. An ambiguous nucleotide aligned to any other nucleotide including the same ambiguous type is treated as a mismatch.

The limitations in the scoring scheme allows more efficient algorithms to be used which is important considering the large data sets being assembled.

5.4 Short Read Reference Assembly

Given a certain quality threshold, it is possible to guarantee that all optimal ungapped alignments are found for each read. Alignments of short reads to reference sequences usually contain no gaps, so the short read assembly operates with a strict scoring threshold to allow the user to specify the amount of errors to accept.

With other short read mapping programs like Maq and Soap, the threshold is specified as the number of allowed mismatches. This works because those programs do global alignment. For local alignments it is a little more complicated.

The default alignment scoring scheme for short reads is +1 for matches and -2 for mismatches. The limit for accepting an alignment is given as the alignment score relative to the read length. For example, if the score limit is 8 below the length, up to two mismatches are allowed as well as two ending nucleotides not assembled (remember that a mismatch costs 2 points, but when there is a mismatch, a potential match is also lost). Alternatively, with one mismatch, up to 5 unaligned positions are allowed. Or finally, with no mismatches, up to 8 unaligned positions are allowed. See figure 5.1 for examples. The default setting is exactly this limit of 8 below the length.

```
CGTATCAATCGATTACGCTATGAATG          CGTATCAATCGATTACGCTATGAATG
   ||||||||||||||||||||      20          ||||||||||||||||||||      19
   ATCAATCGATTACGCTATGA                TTCAATCGATTACGCTATGA

CGTATCAATCGATTACGCTATGAATG          CGTATCAATCGATTACGCTATGAATG
   ||||||||| ||||||||||||      17          |||||||| ||||||||||||      16
   ATCAATCGGTTACGCTATGA                TTCAATCGGTTACGCTATGA

CGTATCAATCGATTACGCTATGAATG          CGTATCAATCGATTACGCTATGAATG
    |||||||| ||||||||||       15          |||||| || ||||||||||||      14
   CTCAATCGGTTACGCTATGA                ATCAACCGGTTACGCTATGA

CGTATCAATCGATTACGCTATGAATG          CGTATCAATCGATTACGCTATGAATG
    |||||||| |||| ||||||       13          ||||||||||| ||||          12
   TTCAATCGGTTACCCTATGA                ATCAATCGATTGCGCTCTTT

CGTATCAATCGATTACGCTATGAATG          CGTATCAATCGATTACGCTATGAATG
    |||||||| |||| |||||       12              ||||||||||||          12
   TTCAATCGGTTACCCTATGC                AGCTATCGATTACGCTCTTT
```

Figure 5.1: Examples of ungapped alignments allowed for a 20 bp read with a scoring limit of 8 below the length using the default scoring scheme. The scores are noted to the right of each alignment. For reads this short, a limit of 5 would typically be used instead, allowing up to one mismatch and two unaligned nucleotides in the ends (or no mismatches and five unaligned nucleotides).

Note that if you choose to do global alignment, the default setting means that up to two mismatches are allowed (because "unaligned positions" at the ends are counted as mismatches as well).

The match score is always +1. If the mismatch cost is changed, the default score limit will also

change to:

$$score\ limit = 3 \times (1 + mismatch\ cost) - 1$$

The default mismatch *score* of -2 equals a mismatch *cost* of 2 and a score limit of 8 below the read length, as stated above. For any mismatch cost, the default score limit allows any alignment scoring strictly better than 3 mismatches.

The maximum score limit also depends on the mismatch cost:

$$max\ score\ limit = 4 \times (1 + mismatch\ cost) - 1$$

Gapped alignment is also allowed for short reads. Contrary to ungapped alignments, it is very difficult to guarantee that all gapped alignments of a certain quality are found. The scoring limit discussed above applies to both gapped and ungapped alignments and there is a guarantee that there are no ungapped exceeding the limit, but there is is no such guarantee for gapped alignments. This being said, the program does a good effort to find the best gapped alignments and usually succeeds.

5.5 Long Read Reference Assembly

For long read assembly, there is no option to perform ungapped alignment because gaps occur easier for longer reads. Because of this, there is no inherent guarantees of finding the optimal alignments according to some scheme. To guarantee finding all optimal alignments, full Smith Waterman alignment would have to be carried out against the whole set of reference sequences. This would take too much computation time to be practical for most data sets.

Instead, a best effort is done to find all the best alignments and this usually succeeds. The quality threshold is determined as a certain fraction of the read matching over a certain identity threshold. The default is that at least half the read must match in at least 90 % of its positions.

Chapter 6

Color space

6.1 Sequencing

The SOLiD sequencing technology from Applied Biosystems is different from other sequencing technologies since it does not sequence one base at a time. Instead, two bases are sequenced at a time in an overlapping pattern. There are 16 different dinucleotides, but in the SOLiD technology, the dinucleotides are grouped in four carefully chosen sets, each containing four dinucleotides. The colors are as follows:

Base 1	**Base 2**			
	A	C	G	T
A	●	●	●	●
C	●	●	●	●
G	●	●	●	●
T	●	●	●	●

Notice how a base and a color uniquely defines the following base. This approach can be used to deduce a whole sequence from the initial nucleotide and a series of colors. Here is a sequence and the corresponding colors.

Sequence	T	A	C	T	C	C	A	T	G	C	A
Colors	●	●	●	●	●	●	●	●	●	●	

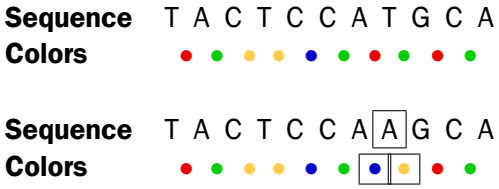
The colors do not uniquely define the sequence. Here is another sequence with the same list of colors:

Sequence	A	T	G	A	G	G	T	A	C	G	T
Colors	●	●	●	●	●	●	●	●	●	●	

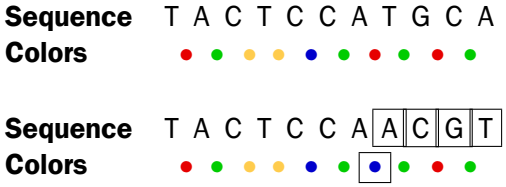
But if the first nucleotide is known, the colors do uniquely define the remaining sequence. This is exactly the strategy used in SOLiD sequencing: The first nucleotide is known from the primer used, and the remaining nucleotides are deduced from the colors.

6.2 Error modes

As with other sequencing technologies, errors do occur with the SOLiD technology. If a single nucleotide is changed, two colors are affected since a single nucleotide is contained in two overlapping dinucleotides:



Sometimes, a wrong color is determined at a given position. Due to the dependence between dinucleotides and colors, this affects the remaining sequence from the point of the error:



Thus, when the instrument makes an error while determining a color, the error mode is very different from when a single nucleotide is changed. This ability to differentiate different types of errors and differences is a very powerful aspect of SOLiD sequencing. With other technologies sequencing errors always appear as nucleotide differences.

6.3 Assembly in color space

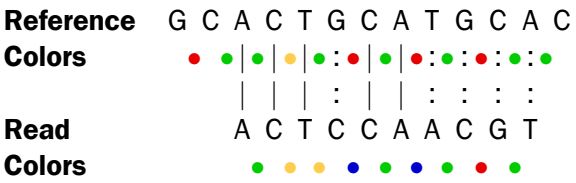
Reads from a SOLiD sequencing run may exhibit all the same differences to a reference sequence as reads from other technologies: mismatches, insertions and deletions. On top if this, SOLiD reads may exhibit color errors, where a color is read wrongly and the rest of the read is affected. If such an error is detected, it can be corrected and the rest of the read can be converted to what it would have been without the error.

Consider this SOLiD read:

The first nucleotide (T) is from the primer, so is ignored in the following analysis. Now, assume that a reference sequence is this:

Here, the colors are just inferred since they are not the result of a sequencing experiment.

Looking at the colors, a possible alignment presents itself:



In the beginning of the read, the nucleotides match (ACT), then there is a mismatch (G in reference and C in read), then two more matches (CA), and finally the rest of the read does not match. But, the colors match at the end of the read. So a possible interpretation of the alignment is that there is a nucleotide change in position four of the read and a color space error between positions six and seven in the read. Such an interpretation can be represented as:

Here, the * represents a color error. The remaining part of the displayed read sequence has been adjusted according to the inferred error. So this alignment scores nine times the match score minus the mismatch cost and a color error cost. This color error cost is a new parameter that is introduced when performing reference assembly in color space.

Note that a color error may be inferred before the first nucleotide of a read. This is the very first color after the known primer nucleotide that is wrong, changing the whole read.

Here is an example from a set of real SOLiD data that was reference assembled by taking color space into account using ungapped global alignments. The assembly_table program with the '-a' option reports:

```
444_1840_767_F3 has 1 match with a score of 35:

    1046535 GATACTCAATGCCGCCAAAGATGGAAGCCGGGCCA 1046569    reference
            ||||||||||||||||||||||||||||||||||
            GATACTCAATGCCGCCAAAGATGGAAGCCGGGCCA            reverse read

444_1840_803_F3 has 0 matches

444_1840_980_F3 has 1 match with a score of 29:

    2620828 GCACGAAAACGCCGCGTGGCTGGATGGT*CAAC*GTC 2620862    reference
            ||||||||||||||||||||||||||||*||||*|||
            GCACGAAAACGCCGCGTGGCTGGATGGT*CAAC*GTC            read

444_1840_1046_F3 has 1 match with a score of 32:

    3673206 TT*GGTCAGGGTCTGGGCTTAGGCGGTGAATGGGGC 3673240    reference
            ||*|||||||||||||||||||||||||||||||||
            TT*GGTCAGGGTCTGGGCTTAGGCGGTGAATGGGGC            reverse read

444_1841_22_F3 has 0 matches

444_1841_213_F3 has 1 match with a score of 29:
```

```
    1593797 CTTTG*AGCGCATTGGTCAGCGTGTAATCTCCTGCA 1593831    reference
            |||||*|||||||| ||||||||||||||||||||
            CTTTG*AGCGCATTAGTCAGCGTGTAATCTCCTGCA            reverse read
```

The first alignment is a perfect match and scores 35 since the reads are all of length 35. The next alignment has two inferred color errors that each count is -3 (marked by * between residues), so the score is *35 - 2 x 3 = 29*. Notice that the read is reported as the inferred sequence taking the color errors into account. The last alignment has one color error and one mismatch giving a score of *34 - 3 - 2 = 29*, since the mismatch cost is 2.

Running the same reference assembly without allowing for color errors, the result is:

```
444_1840_767_F3 has 1 match with a score of 35:

    1046535 GATACTCAATGCCGCCAAAGATGGAAGCCGGGCCA 1046569    reference
            |||||||||||||||||||||||||||||||||||
            GATACTCAATGCCGCCAAAGATGGAAGCCGGGCCA            reverse read

444_1840_803_F3 has 0 matches

444_1840_980_F3 has 0 matches

444_1840_1046_F3 has 1 match with a score of 29:

    3673206 TTGGTCAGGGTCTGGGCTTAGGCGGTGAATGGGGC 3673240    reference
              |||||||||||||||||||||||||||||||
            AAGGTCAGGGTCTGGGCTTAGGCGGTGAATGGGGC            reverse read

444_1841_22_F3 has 0 matches

444_1841_213_F3 has 0 matches
```

The first alignment is still a perfect match, whereas two of the other alignment now do not match since they have more than two errors. The last alignment now only scores 29 instead of 32, because two mismatches replaced the one color error above. This shows the power of including the possibility of color errors when aligning: many more matches are found.

The reference assembly program in the CLC Assembly Cell does not directly support alignment in color space only, but if such an alignment was carried out, sequence 444_1841_213_F3 would have three errors, since a nucleotide mismatch leads to two color space differences. The alignment would look like this:

```
444_1841_213_F3 has 1 match with a score of 26:

    1593797 CTTTG*AGCGCATT*G*GTCAGCGTGTAATCTCCTGCA 1593831    reference
            |||||*||||||||*|*||||||||||||||||||||
            CTTTG*AGCGCATT*G*GTCAGCGTGTAATCTCCTGCA            reverse read
```

So, the optimal solution is to both allow nucleotide mismatches and color errors in the same program when dealing with color space data. This is the approach taken by the assembly program in the CLC Assembly Cell.

To invoke color space assembly, use the '-c' option. The cost of color errors is set using '-y' (range 1-3, default is 3). Note that the limit is also affected by the color space error cost:

6.3.1 Score limit

When using color space, there are additional constraints to setting the score limit. The limit is then calculated this way:

m = mismatch cost

c = color error cost

e = min(m + 1, c)

score limit = 3 x e - 1 (just one point short of three errors)

6.4 File formats

The `.csfasta` file format is often used for color space data. That format looks like this:

```
#picked reads from data/reads/SHIRAZ_20080320_MP_2_Sample1_F3.csfasta.original, panel ra
09
>600_50_31_F3
T22220021133003221321122 31
>600_50_63_F3
T23301332121301332210331 10
>600_50_100_F3
T01300011310123102010001 01
>600_50_170_F3
T10023121030331213212331 03
>600_50_174_F3
T03300223303320003230311 21
>600_50_241_F3
T21031031031002121230300 11
>600_50_256_F3
T03011310102333112002233 32
>600_50_329_F3
T13032110331123013032200 00
>600_50_342_F3
T21000030122120003101301 11

...
```

So, it is very similar to the fasta file format. It does, however, allow one or more lines starting with `#` before the first sequence. The sequences are specified as a nucleotide followed by the colors encoded as numbers where 0 is blue, 1 is green, 2 is yellow, and 3 is red. So the sequence:

Sequence	T	A	C	T	C	C	A	T	G	C	A
Colors	●	●	●	●	●	●	●	●	●	●	

Would be coded like this in a `.csfasta` file:

```
>sequence
T3122013131
```

The T is the nucleotide that is known from the primer and the numbers indicate the colors. Because the T came from the primer, it is not part of the sequenced DNA molecule. Thus,

this letter should be ignored when analyzing the read. So this sequence would look like this in `.fasta` format:

```
>sequence
ACTCCATGCA
```

So there is one nucleotide for each experimentally determined color (i.e. the numbers in the `.csfasta` file).

The `.csfasta` does not contain any significant information that is not also present in a standard fasta file of the same sequences. The only extra information is the last nucleotide of the primer, which is not useful in later analyses.

So from the viewpoint of software programs analyzing read data, color space is just yet another file format for reads along with `.fasta`, `.fastq`, `.sff`, etc. Thus, in the Assembly Cell programs, color space options for assembly have no connection to file formats. You can choose to assemble SOLiD data in `.csfasta` format without using the color space options for assembly and you can also choose to assemble reads in a normal `.fasta` file using color space assembly options.

Chapter 7

De novo assembly

The clc_novo_assemble program performs assembly of reads without a known reference. The input is a number of files containing reads and the output is a fasta file of contig sequences. Any number of read files can be used, and short and long reads can also be used together.

The '-p' option can be used to set approximate minimum and maximum distances between pairs. All the paired-end options are the same as for reference assembly as described above.

7.1 How it works

CLC bio's de novo assembly algorithm works by using de Bruijn graphs. This is similar to how most new de novo assembly algorithms work. 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 a bacterial genome and 27 for a human genome.

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 7.1.

Backward neighbors	Starting word	Forward neighbors
AACGTAGCTAGCGCAT	ACGTAGCTAGCGCATG	CGTAGCTAGCGCATGA
CACGTAGCTAGCGCAT		CGTAGCTAGCGCATGC
GACGTAGCTAGCGCAT		CGTAGCTAGCGCATGG
TACGTAGCTAGCGCAT		CGTAGCTAGCGCATGT

Figure 7.1: *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.*

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 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 7.2 shows an example where one node has two forward neighbors:

Figure 7.2: *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 7.3.

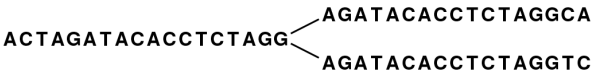


Figure 7.3: *The five nodes are compacted into three. Note that the first node is now 18 bases and the second nodes are each 17 bases.*

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 length 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 as shown in figure 7.4.

Figure 7.4: *A bubble caused by a 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, we would see that one path through the bubble will only be words seen a single time. On the other hand if this was a SNP we would see both paths represented more or less equally. 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.

If we have a *repeat sequence* that is present twice in the genome, we would get a graph as shown in figure 7.5.

Figure 7.5: *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 length 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 extra nucleotide in the words, 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 of the word length is 20. If the word length is 25, we get only 11 words from each read.

To strike a balance, CLC bio's de novo assembler chooses a word length based on the amount of input data: the more data, the longer the word length. The word size can also

be specified manually using the `-w" option. The range of word sizes is 12-24 on 32-bit computers and 12-31 on 64-bit computers. Using the -v" (verbose) option, you can see the word size that is automatically calculated by the assembler`A simple de novo assembly result would be to output the sequence of each reduced node. The bubbles described above from SNPs and sequencing errors as well as the repeats will make this quite a bad result with many short contigs. Instead, we can try to resolve the repeats with reads that span from a node before the repeat to a node after the repeat. Small bubbles can be resolved by choosing the path with the most coverage. Thus, by using the information from the full length reads, we are able to produce much longer contigs.

Furthermore, when paired reads are available, we can use this information to resolve even larger repeat regions that may not be spanned by individual reads, but are spanned by read pairs. This results in even longer contigs.

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

- Make a table of the words seen in the reads.
- Build de Bruijn graph from the word table.
- Use the reads to resolve the repeats.
- Use the information from paired reads to resolve larger repeats.
- Output resulting contigs based on the paths.

These stages are all performed by the assembler program.

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, further complicating the graph. Sometimes a repeat is longer than the read length (or the paired end distance when pairs are available) and then it becomes impossible to resolve 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. This means that no matter how much coverage we have, we will still get a number of separate contigs as a result.

7.2 Specific characteristics of CLC bio's algorithm

There are some advantages and some disadvantages of CLC bio's algorithm when compared to other programs such as *Velvet* [**?**] and *SOAPdenovo* [**?**]. The advantages are:

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

One of the disadvantages is that the use of paired information in `clc_novo_assemble` is not quite optimal. The problem with `clc_novo_assemble` is that it does not use paired end

[1] See how SOLiD is supported in section 7.3

information to connect two nodes if it cannot resolve the path from one node to the other. This may occur if there is a spot with no coverage or if there is a very complex repeat region spanned by paired reads, but not by individual reads. Connecting nodes without knowing exactly what is between them is typically called scaffolding. We are working on an updated assembly program which includes this scaffolding.

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.

7.3 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 7.6.

Figure 7.6: *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_novo_assemble` program has a special option ("-p d") to indicate that a certain data set should be used only for their 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.

7.4 Other options

By default a contig has to contain at least 200 nucleotides to be reported, but the '-m' can be used to change this to a different number.

Note that you can use the `sequence_info` program described below with the '-n' option to get

statistics on the result of a de novo assembly.

The output of the `clc_novo_assemble` 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 reference assembly programs described above with the newly created contig sequences as references. This will create a cas file with this information.

See full usage including examples in section A.

Chapter 8

Working with Assemblies

8.1 The sequence_info Program

The 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 small sequences may not be of interest.

8.2 The assembly_table Program

The assembly_table program takes a single cas file as input and prints assembly information for each read. By default, assembly_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 end 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.

8.3 The assembly_info Program

Whereas assembly_table outputs detailed information about individual matches, the assembly_info program instead gives an overview:

```
General info:

  Program name         clc_ref_assemble_short
  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 end distances using the assembly_info program. This is done with the standard '-p' option and results in output like this:

```
Paired end 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 end analysis assembly_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 end 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 sub_assembly to make an assembly with only the paired end data before analyzing.

When a data set contains paired end data of unknown distances, a good approach is to make an

initial reference assembly without using paired end information. Then the assembly_info program can be used to investigate the paired end distance properties of the data using wide limits for the distances. Finally, a reference assembly run can be performed with the estimated paired end 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 end distance properties of the results, but a smaller fraction of the reads might match.

8.4 The filter_matches Program

The filter_matches program removes matches of low similarity from a cas file. The limits for low similarity is 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.

8.5 The sort_pairs Program

A SOLiD paired end data set usually comes in two `.csfasta` files, but unlike Illumina paired end data the sequences are not necessarily all paired. This means that one cannot assume that sequence one from file one pairs with sequence one from file two, and sequence two from file one pairs with sequence two from file two, etc. Instead, only the names of the sequences are used to indicate which sequences form pairs.

The sort_pairs program takes two SOLiD read files as input and outputs a file with unpaired reads and a file with paired reads. These files are then ready for further analysis, e.g. using clc_ref_assemble_short. Note that the output format is fasta, but no information is lost relative to .csfasta format as discussed in the color space section.

8.6 The split_sequences Program

The 454 sequencing technology allows paired end reads by having two paired read fragments in 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 contain a pair.

The split_sequences program finds this linker sequence and creates two new files, one with unpaired reads and one with paired reads. The '-m' option specifies the minimum read length to report. 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 the fragment is below the specified minimum length, it is discarded along with the linker. The remaining part of the read is reported as unpaired.

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. If only very few nucleotides of the linker overlaps with the read, it can be difficult to determine if the short overlap is in fact a linker or just some nucleotides that are identical to the small linker fragment. The '-o' option sets the limit for when to discard nucleotides at the start and end of the read because they might be the end of a linker. The default is two nucleotides.

8.7 The change_assembly_files Program

This program allows you to change the file names in an assembly file. It is useful if you have moved the sequence files after the assembly was made. Or if you for example made the assembly with relative file names and want to change the file names to absolute names (or vice versa). It is also possible to change the file format, for example from fasta to GenBank format if you wish a richer representation of the sequence. For the operation to be a success, however, the actual sequences and their order must remain unchanged.

With the change_assembly_files program, file names are specified like they are when making the original reference assembly, i.e. using the '-d', '-q', and '-i' options. The output assembly file is specified with the '-o' option and the input assembly file is specified with the '-a' option.

To make the change in place, use the same assembly file name for input and for output. It is of course slightly safer to use different file names, so a backup of the original is kept.

By default, the program compares the sequence files to make sure they contain the same data. This takes some time, so the '-n' option is included to avoid this check. The '-n' option is also useful if the old sequence files does not exist any more.

8.8 The join_assemblies Program

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

8.9 The sub_assembly Program

The sub_assembly program allows the user to make a new assembly containing only part of the original assembly.

8.9.1 Specifying Assembly Files

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

8.9.2 Extracting a Subset of Reference Sequences

The '-s' option is used for making a new assembly with only matches to a single reference sequence. The '-d' option makes a new assembly 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 assembly. You can use assembly_info to see the contents of the input assembly is needed.

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

8.9.3 Extracting a Part of a Single Reference Sequence

If a single reference sequence is specifies using the '-s' option or if the input assembly contains only a single reference sequence, the '-b' option may be used to specify a position range to extract. The output assembly 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.

8.9.4 Extracting a Subset of Read Sequences

Using the '-q' option, you can make an assembly with only the reads from one of the read files. The read file is specified by its number in the input assembly. If reads are interleaved, the output assembly 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 even is the full assembly contains reads from several experiments.

8.9.5 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.

8.9.6 Output Reference File

By default, the output assembly refers to one or all of the reference files in the input assembly. 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 assembly. The new assembly 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 assembly only contains the relevant reference sequence instead of many references with no matches. It makes the output assembly easier and faster to work with.

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

8.9.7 Output Read File

By default, the output assembly refers to one or all of the read files in the input assembly. 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 assembly automatically refers to this new read file instead of the originals.

This is very useful when making a sub assembly 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 assembly, making subsequent analyses more efficient.

When the reads are from a paired end experiment, the assembly analysis programs expect 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 sub_assembly program, the paired end read order is disrupted. Because of this, the '-p' option should be used when the reads are from a paired end experiment. It works by retaining reads that do not match the sub_assembly 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.

8.9.8 Handling of non-specific matches

If an assembly contains non-specific match reads and a sub assembly 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 sub_assembly program is meant to make it simpler to study a small region of a large assembly, so the original characteristics of the larger assembly are kept.

8.10 The find_variations Program

This program makes a new reference sequence file containing all the original data but with changes made so the references reflect the read sequences of an assembly. The new reference file is always in fasta format. It is also possible to run the program so it only prints a list of differences instead of actually making a new file.

There is an option '-c' to determine minimum coverage for read differences to be reported.

The find_variations program is used after reference assembly to get an estimate of the actual sequences that was studied in the sequencing experiment.

If you wish to see the reads matched to the new reference sequences, a new round of reference assembly has to be performed. The reason for this is that the changes to the references may significantly change the optimal locations of the reads in the changed regions. So a complete new reference assembly is necessary. Sometimes the new read alignments may suggest a few more changes to the reference sequences, so another run of find_variations may be in order.

8.11 The unassembled_reads Program

This program extracts the unassembled read sequences from an assembly. They are output in fasta file. By default the only output sequences are the ones that does not match at all. Using the options it is also possible to output the unaligned ends of reads. A minimum length of unassembled sequecnce 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 assembly. Sometimes, performing de novo assembly on these unassembled reads may be useful to determine their source. It could, for example, be mitochondrial DNA or vector sequence contamination.

Chapter 9

Assembly Viewer

The assembly viewer program shows assemblies in a text based terminal window. It is useful for getting a quick overview of the data and for investigating interesting places.

The program takes one or more assembly files as parameters. For large assemblies, it may take a little while to start since the reads have to be sorted for viewing. The key bindings are as follows:

Key	Description
Arrows	Move view.
0–9	Any (possibly multi digit) number followed by any other key: move to that position. Follow by 'K' to multiply by 1,000, or 'M' to multiply by a million.
z	Center vertical position on reads.
v	Scroll left to interesting part and center horizontally.
b	Scroll right to interesting part and center horizontally.
c	Toggle color scheme.
m	Toggle position marks.
e	Toggle how to show unaligned ends.
r	Toggle between contigs.
j	Toggle joint read view.
p	Move to same position as for last contig.
h	Show help screen.
s	Search for a sequence in the reference.
q	Quit.

Using shift together with one of the toggle keys ('C', 'E', 'R' and 'M') cycles the other direction. Using shift with one of the movement keys (including arrows) makes the movement faster. This also applies to the 'K' and 'M' keys for sequence positions. Figures 9.1–9.4 show some screen shots and examples.

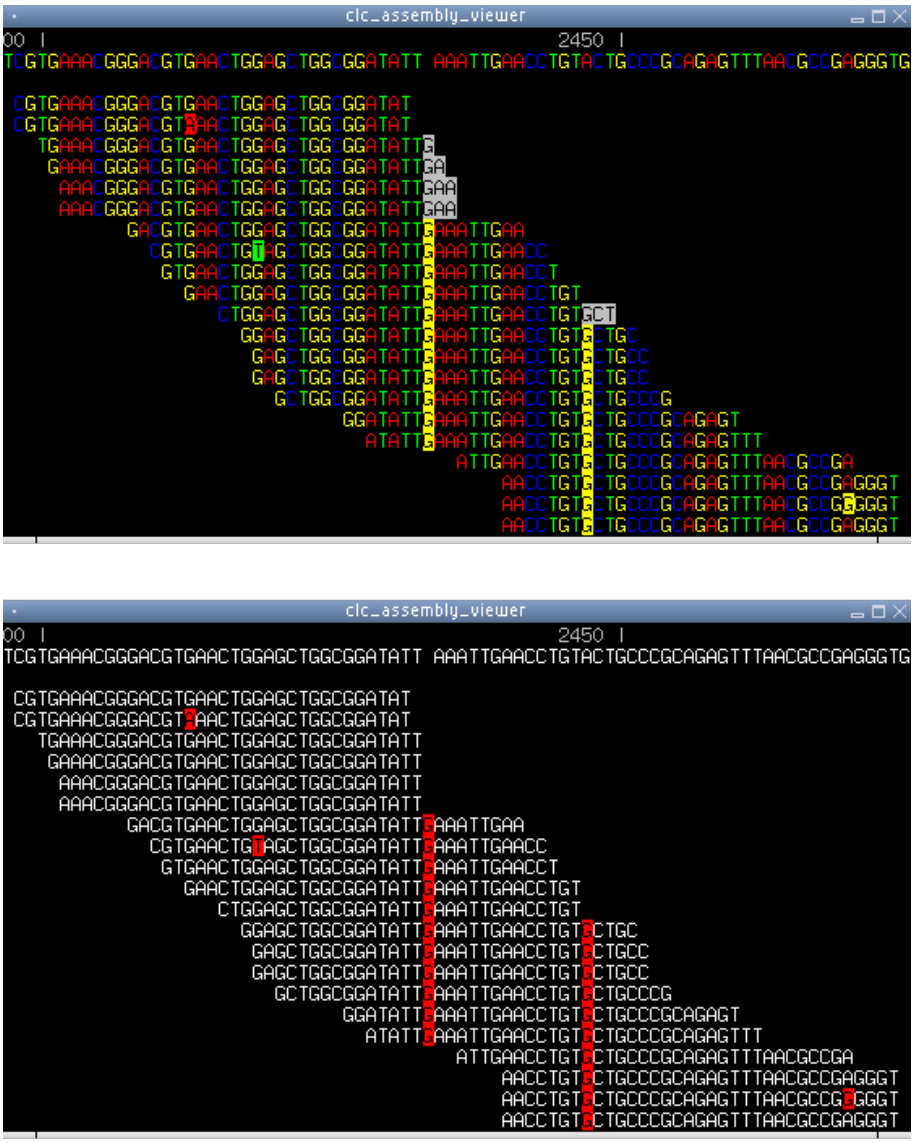


Figure 9.1: *Two screen shots from the assembly viewer. Top) Residue coloring. Residues differing from the reference are highlighted. The first column of highlighted G's is an insertion, the second is a mutation (the reference residue is A in that position). The reversed gray residues at the end of some of the reads are not aligned. Bottom) Another color scheme, where differences are easier to spot. Here the unaligned residues have also been turned off.*

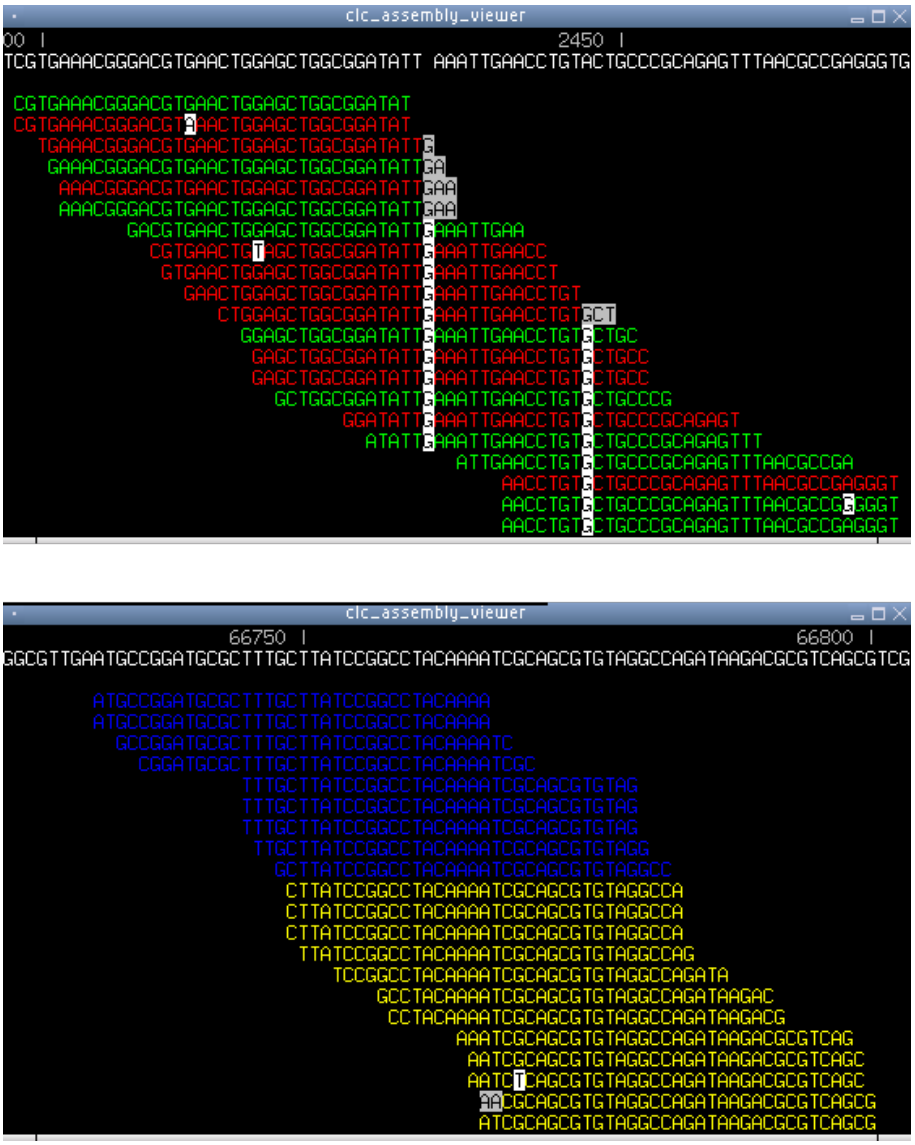


Figure 9.2: *Another screen shot from the assembly viewer. Here, the color scheme is according to the direction of the reads. Green is forward, red is reverse.*

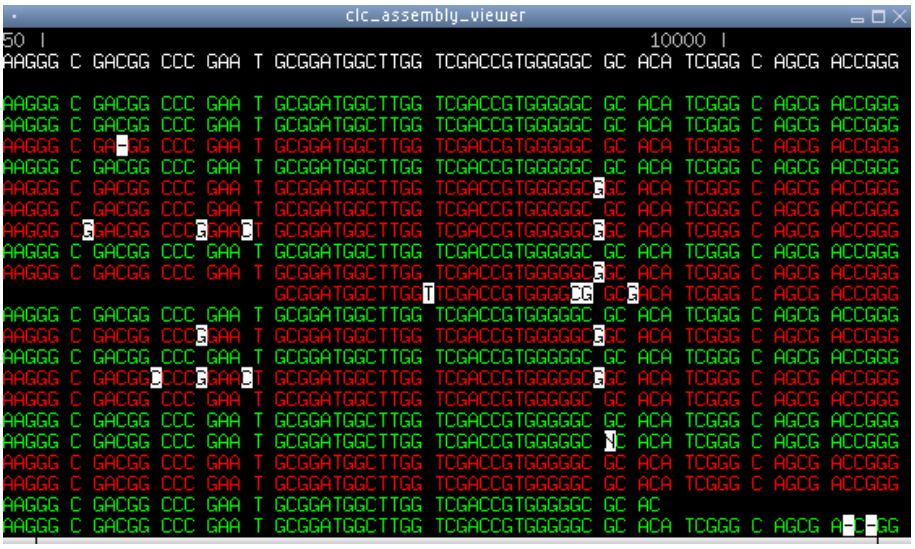


Figure 9.3: *A screen shot with 454 sequencing data. The directional color scheme is useful for recognizing a particular type of sequencing error with the 454 technology. Notice the position with five inserted G's. They are sequencing errors arising from the stretch of five G's to their left, before the C. These errors tend to occur before a stretch of identical residues, which is why they are only seen in the reverse reads in this case.*

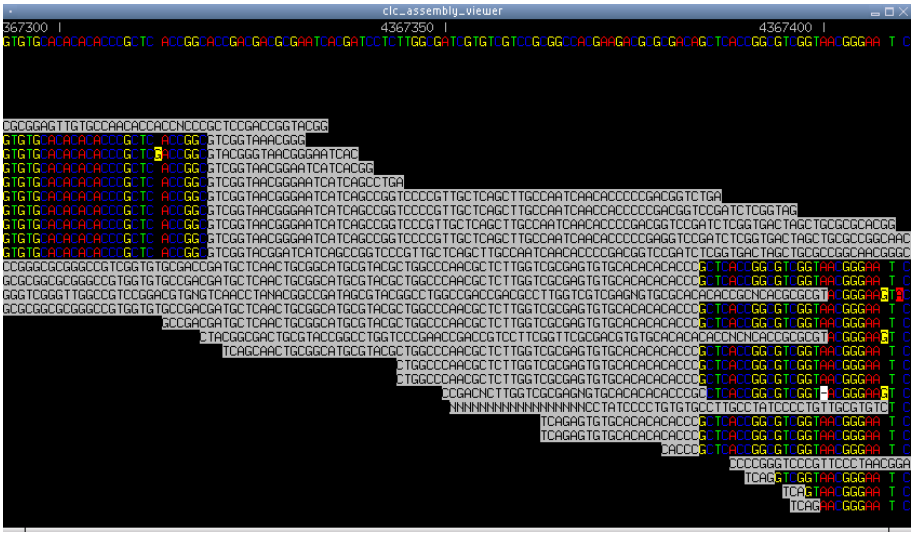


Figure 9.4: *A screen shot with 454 sequencing data. This is how a genomic rearrangement looks in a reference assembly. Suddenly the reads do not match any more, and later another set of reads abruptly start matching. These reads may actually be very distant in the real genome (as opposed to the reference).*

Appendix A

Options for All Programs

A.1 Options for assembly_info

```
usage: assembly_info [options] <assembly file>

  Print information about an assembly.

Options:

  -h / --help: Display this message.

  -c / --coverage: Show more detailed coverage information

  -d <file> / --coveragefile <file>: Output coverage as a function of position
     for each reference sequence to different files called <file>.001.dat,
     <file>.002.dat, etc.

  -p <par> / --paired <par>: Set the paired read mode.

     par consists of four strings: <mode> <dist_mode> <min_dist> <max_dist>

     mode is ff, fb, bf, bb and sets the relative orientation of read one and
     two in a pair (f = forward, b = backward).

     dist_mode is ss, se, es, ee and sets the place on read one and two to
     measure the distance (s = start, e = end).

     A typical use would be "-p fb ss 180 250" which means that the reads are
     inverted and pointing towards each other. The distance includes both the
     reads and the sequence between them. The distance may be between 180 and
     250, both included.

     Only read pairs satisfying these creteria are counted in the distance
     statistics.

  -q <file> / --pairedfile <file>: Output file for distance histogram for
     paired end data.

  -f / --fast: No coverage information for a fast result.

  -m / --mismatch: Show counts of mismatches, insertions and deletions.
```

A.2 Options for assembly_table

```
usage: assembly_table [options] <assembly file>

  Print information about each match in an assembly file. The columns are:

    Read number
    Read name (enable using the '-n' option)
    Read length
    Read position for alignment start
    Read position for alignment end
    Reference sequence number
    Reference position for alignment start
    Reference position for alignment end
    Whether the read is reversed (0 = no, 1 = yes)
    Number of matches
    Whether the read is paired with the next one (0 = no, 1 = yes) (enable
                                                          using the '-p' option
    Alignment score (enable using the '-s' option)

Options:

  -h / --help: Display this message.

  -n / --names: Include the read names.

  -s / --scores: Include the alignment scores.

  -p / --paired: Include pair information.

  -a / --alignments: Print the full alignments, including names and scores.
```

A.3 Options for change_assembly_files

```
usage: change_assembly_files <options>

  Change the sequence file names in an assembly file. Can be used for the
  reference files, the read files, or both.

Options:

  -h / --help: Display this message.

  -a <file> / --assembly <file>: Give the assembly file (required).

  -o <file> / --output <file>: Give the output assembly file (required).

  -q / --reads: The files following this option are read files. Fasta, fastq,
     and sff formats are allowed (may be used several times).

  -d / --reference: The files following this option are reference files. Fasta
     and GenBank formats are allowed (may be used several times).

  -i <file1> <file2> / --interleave <file1> <file2>: Interleave the sequences
     in two files, alternating between the files when reading the
     sequences. Only valid for read files (may be used several times).
```

```
  -n / --nocheck: Do not check if the sequence files match. This is useful if
     the old files do not exist any more, or to get a fast result if the files
     are known to match.
```

A.4 Options for clc_assembly_viewer

```
usage: clc_assembly_viewer <assembly files>

  Show a number of assemblies in a text viewer. Type H to show an overview of
  the key bindings.
```

A.5 Options for clc_novo_assemble

```
usage: novo_assemble [options]

  De novo assemble some reads and output contig sequences in fasta format.

Options:

  -h / --help: Display this message

  -q / --reads: The files following this option are read files. Fasta, fastq,
     and sff formats are allowed. (may be used several times)

  -i <file1> <file2> / --interleave <file1> <file2>: Interleave the sequences
     in two files, alternating between the files when reading the
     sequences. Only valid for read files. (may be used several times)

  -o <file> / --output <file>: Give the output fasta file (required)

  -m <n> / --min-length <n>: Set the minimum contig length to output (default =
     200)

 -w <n> / --word-size <n>: Set the word size for the de Bruijn graph (default
    is automatic based on input data size)

 -v / --verbose: Output various information while running.

  -p <par> / --paired <par>: Set the paired read mode for the read files
     following this option. (may be used several times)

     par consists of four strings: <mode> <dist_mode> <min_dist> <max_dist>

     mode is ff, fb, bf, bb and sets the relative orientation of read one and
     two in a pair (f = forward, b = backward)

     dist_mode is ss, se, es, ee and sets the place on read one and two to
     measure the distance (s = start, e = end)

     A typical use would be "-p fb ss 180 250" which means that the reads are
     inverted and pointing towards each other. The distance includes both the
     reads and the sequence between them. The distance may be between 180 and
     250, both included.
```

```
     It is also allowed to insert a "d" before the mode. This indicates that
     the reads in the following file(s) should only be used for their paired end
     information and not to build initial contigs. E.g. "-p d fb ss 180 250".

     To explicitly say that the following reads are not paired, use "no" for
     par, i.e. "-p no".

     For paired end reads split in two files, use the -i option.

Examples:

  De novo assembly of a single file with reads:

    novo_assemble -o contigs.fasta -q reads.fasta

  De novo assembly of two interleaved files with paired end reads:

    novo_assemble -o contigs.fasta -p fb ss 180 250 -q
                                                  -i reads1.fq reads2.fq
```

A.6 Options for clc_ref_assemble_long

```
usage: clc_ref_assemble_long <options>

  Reference assemble some reads to some reference sequences. Mostly used for
  reads longer than 55 or of varying differences.

Options:

  -h / --help: Display this message

  -q / --reads: The files following this option are read files. Fasta, fastq,
     and sff formats are allowed. (may be used several times)

  -d / --reference: The files following this option are reference files. Fasta
     and GenBank formats are allowed. (may be used several times)

  -o <file> / --output <file>: Give the output assembly file (required)

  -i <file1> <file2> / --interleave <file1> <file2>: Interleave the sequences
     in two files, alternating between the files when reading the
     sequences. Only valid for read files. (may be used several times)

  -x <n> / --mismatchcost <n>: Set the mismatch cost (range 1 to 3, default 2)

  -g <n> / --gapcost <n>: Set the gap cost (range 1 to 3, default 3)

  -e <n> / --deletioncost <n>: Set the deletion cost in which case the gap cost
     setting only applies to insertions. (range 1 to 3, default 3)

  -c / --colorspace: Use color space when aligning.

  -y <n> / --colorerrorcost <n>: Set the cost of an error in a color when using
     color space. Can only be used with the "-c" option. (range 1 to 3, default
     3)
```

```
  -r <mode> / --repeat <mode>: Set the behavior for reads that match more than
     once, i.e. ignore such reads or place them randomly among the valid
     locations (ignore / random) (default random)

  -l <n> / --lengthfraction <n>: Set the fraction of the read that must match. A
     real number between 0.0 and 1.0 (default 0.5).

  -s <n> / --similarity <n>: Set the limit for the similarity in the fraction
     of the read that must match (according to "-l" option). A real number
     between 0.0 and 1.0 (default 0.8).

  -p <par> / --paired <par>: Set the paired read mode for the read files
     following this option. (may be used several times)

     par consists of four strings: <mode> <dist_mode> <min_dist> <max_dist>

     mode is ff, fb, bf, bb and sets the relative orientation of read one and
     two in a pair (f = forward, b = backward)

     dist_mode is ss, se, es, ee and sets the place on read one and two to
     measure the distance (s = start, e = end)

     A typical use would be "-p fb ss 180 250" which means that the reads are
     inverted and pointing towards each other. The distance includes both the
     reads and the sequence between them. The distance may be between 180 and
     250, both included.

     To explicitly say that the following reads are not paired, use "no" for
     par, i.e. "-p no".

     For paired end reads split in two files, use the -i option.

  -m <n> / --memory <n>: Set the maximum amount of memory to use as a fraction
     of the available memory (default is 1.0).

  -a <mode> / --alignmode <mode>: Set the alignment mode to one of the
     following:

       local: perform local alignment (default)

       global: perform global alignment

       semi-global: perform semi-global alignment

Examples:

  Reference assembly a single file with reads to a single file with reference
  sequences:

    clc_ref_assemble_long -o assembly.cas -q reads.fasta -d reference.fasta

  Reference assemble reads from two unpaired runs and a paired end run split
  across two files. Use two reference sequences:

    clc_ref_assemble_long -o assembly.cas -q unpaired1.fasta unpaired2.fasta
                          -p fb ss 180 250 -i paired_1.qf paired_2.qf -d
                          reference1.gb reference2.gb
```

A.7 Options for clc_ref_assemble_short

```
usage: clc_ref_assemble_short <options>

  Reference assemble some reads to some reference sequences. Maximum read
  length is 55.

Options:

  -h / --help: Display this message

  -q / --reads: The files following this option are read files. Fasta, fastq,
     and sff formats are allowed. (may be used several times)

  -d / --reference: The files following this option are reference files. Fasta
     and GenBank formats are allowed. (may be used several times)

  -o <file> / --output <file>: Give the output assembly file (required)

  -i <file1> <file2> / --interleave <file1> <file2>: Interleave the sequences
     in two files, alternating between the files when reading the
     sequences. Only valid for read files. (may be used several times)

  -x <n> / --mismatchcost <n>: Set the mismatch cost (range 1 to 3, default 2)

  -g <n> / --gapcost <n>: Set the gap cost (range 1 to 3, default 3)

  -e <n> / --deletioncost <n>: Set the deletion cost in which case the gap cost
     setting only applies to insertions. (range 1 to 3, default 3)

  -c / --colorspace: Use color space when aligning.

  -y <n> / --colorerrorcost <n>: Set the cost of an error in a color when using
     color space. Can only be used with the "-c" option. (range 1 to 3, default
     3)

  -u / --ungapped: Use ungapped alignment (default is gapped alignment)

  -r <mode> / --repeat <mode>: Set the behavior for reads that match more than
     once, i.e. ignore such reads or place them randomly among the valid
     locations (ignore / random) (default random)

  -s <n> / --scorelimit <n>: Set the limit for the score. The limit is defined
     as the number of points below the read length to accept (default is 8 for
     default scoring scheme).

  -n <n> / --movelimit <n>: Move the length limit for short sequences that are
     not aligned. By default it is 22, 26, and 30 for 1, 2, and 3 errors,
     respectively. By using this option, it is lowered by n.

  -p <par> / --paired <par>: Set the paired read mode for the read files
     following this option. (may be used several times)

     par consists of four strings: <mode> <dist_mode> <min_dist> <max_dist>

     mode is ff, fb, bf, bb and sets the relative orientation of read one and
     two in a pair (f = forward, b = backward)

     dist_mode is ss, se, es, ee and sets the place on read one and two to
```

```
     measure the distance (s = start, e = end)

     A typical use would be "-p fb ss 180 250" which means that the reads are
     inverted and pointing towards each other. The distance includes both the
     reads and the sequence between them. The distance may be between 180 and
     250, both included.

     To explicitly say that the following reads are not paired, use "no" for
     par, i.e. "-p no".

     For paired end reads split in two files, use the -i option.

  -m <n> / --memory <n>: Set the maximum amount of memory to use as a fraction
     of the available memory (default is 1.0).

  -a <mode> / --alignmode <mode>: Set the alignment mode to one of the
     following:

       local: perform local alignment (default)

       global: perform global alignment

       semi-global: perform semi-global alignment

Examples:

  Reference assembly a single file with reads to a single file with reference
  sequences:

    clc_ref_assemble_short -o assembly.cas -q reads.fasta -d reference.fasta

  Reference assemble reads from two unpaired runs and a paired end run split
  across two files. Use two reference sequences:

    clc_ref_assemble_short -o assembly.cas -q unpaired1.fasta unpaired2.fasta
                           -p fb ss 180 250 -i paired_1.qf paired_2.qf -d
                           reference1.gb reference2.gb
```

A.8 Options for filter_matches

```
usage: filter_matches <options>

  Remove matches from an assembly if they do not live up to some given criteria.

Options:

  -h / --help: Display this message.

  -a <file> / --assembly <file>: Set the input assembly file (required).

  -o <file> / --output <file>: Set the output assembly file (required).

  -l <n> / --lengthfraction <n>: Set the fraction of the read that must match. A
     real number between 0.0 and 1.0 (required).

  -s <n> / --similarity <n>: Set the limit for the similarity in the fraction
```

```
     of the read that must match (according to "-l" option). A real number
     between 0.0 and 1.0 (required).
```

A.9 Options for find_variations

```
usage: find_variations <options>


 -h / --help: Display this message

 -a <file> / --assembly <file>: Specify the assembly file (required).

 -c <n> / --coverage <n>: Specify minimum coverage to report/apply difference
    (default: 2)

 -o <file> / --output <file>: Specify the output fasta file.

 -q / --quiet: Output no information about the reported sites.

 -v / --verbose: Show more information about the reported sites.

 -l <count> / --limit <count>: Show information when more than a given number
    of reads is different from the consensus. Can only be used with the "-v"
    option.

 -f <fraction> / --limitfraction <fraction>: Show information when more than a
    given fraction of reads is different from the consensus. Can only be used
    with the "-v" option. If used with the "-l" option, both requirements must
    be met.

Examples:

 Find all sites where the reads indicate differences relative to the reference
 sequence:

   find_variations -a assembly.cas

 The differences are printed to stdout. To make a new reference sequence with
 the differences incorporated, write:

   find_variations -a assembly.cas -o new_ref.fasta

 By default, only sites with at least two fold coverage are included in the
 analysis. To set this to five fold coverage, use the "-c" option:

   find_variations -c 5 -a assembly.cas

 With this, differences are only printed for sites with at least five fold
 coverage. If the "-c" and "-o" options are used together, changes are only
 made to the reference sequence when the coverage requirement is met. In
 general, the changes made to the reference sequence when using the "-o"
 option are exactly those changes output to stdout (except when using the "-q"
 option where no output is printed).

 Using the "-l" and/or "-f" options with the "-v" option gives output for
 sites where no change is indicated, but some significant amount of
 differences is still present. For example:
```

```
   find_variations -l 2 -f 0.2 -a assembly.cas

 This outputs information for all sites where at least two reads differ from
 the reference and at least 20% of the reads differ from the reference.

 Note that when using the "-o" option, the new reference sequence is not
 affected by the "-q", "-v", "-l" and "-f" options. The "-c" option,
 however, does affect the new reference sequence.
```

A.10 Options for join_assemblies

```
usage: join_assemblies <options> <input assembly 1> <input assembly 2> ...

  Joins any number of assemblies with identical reference files into one.

Options:

  -h / --help: Display this message.

  -o <file> / --output <file>: Set the output assembly file (required).
```

A.11 Options for sequence_info

```
usage: sequence_info [options] <sequence file>

  Print some information about a sequence file. The accepted formats are fasta,
  fastq, sff, and GenBank.

Options:

  -h / --help: Display this help

  -l / --lengths: Print length of each sequence

  -k / --lengthcounts: Print number of sequences of each length

  -n / --n50: Calculate the N50 value

  -c <n> / --cutoff <n>: Ignore all sequences below a minimum sequence length

  -r / --residues: Include residue counts

  -a / --aminoacids: Residue are amino acids (only relevant when including
     residue counts)
```

A.12 Options for sort_pairs

```
usage: sort_pairs [options]

  Split the sequences in a file according to their names to produce a file
  with paired end sequences and one with unpaired sequences.
```

```
Options:

  -h / --help: Display this help

  -i <file1> <file2> / --input <file1> <file2>: Two input sequence files, first
     the forward read file and then reverse (required)

  -s <file> / --singleoutput <file>: Output fasta file for single reads
     (required)

  -p <file> / --pairedoutput <file>: Output fasta file for paired reads
     (required)
```

A.13 Options for split_sequences

```
usage: split_sequences [options]

  Split the sequences in a file according to a linker sequence to produce a file
  with paired end sequences and one with unpaired sequences.

Options:

  -h / --help: Display this help

  -i <file> / --input <file>: Input sequence file (required)

  -s <file> / --singleoutput <file>: Output file for single reads (required)

  -p <file> / --pairedoutput <file>: Output file for paired reads (required)

  -l <seq> / --linker <seq>: Set the linker sequence (default is the 454 paired
     end linker: GTTGGAACCGAAAGGGTTTGAATTCAAACCCTTTCGGTTCCAAC)

  -m <n> / --minlength <n>: Set the minimum sequence length to output (default
     is 15)

  -o <n> / --minoverlap <n>: Set the minimum length of the overlap between the
     linker and the read (default is 2)
```

A.14 Options for sub_assembly

```
usage: sub_assembly <options>

  Extract part of an assembly into a new assembly file.

Options:

  -h / --help: Display this message

  -a <file> / --assembly <file>: Set the input assembly file (required).

  -o <file> / --output <file>: Set the output assembly file (required).
```

```
  -d <n> / --reffile <n>: Restrict matches to a single reference file denoted
     by its number.

  -s <n> / --refseq <n>: Restrict matches to a single reference sequence
     denoted by its number.

  -q <n> / --readfile <n>: Restrict matches to a single read file (or two if
     interlaced) denoted by its number.

  -b <m-n> / --subsequence <m-n>: Restrict matches to a position range. The
      positions start from 1. The '-s' option must also be specified if more
      than one reference sequence is present in the assembly.

  -u / --unique: Restrict to uniquely placed matches.

  -l <n> / --minlength <n>: Restrict to matches where a minimum of n read
     positions are aligned (but not necessarily matching).

  -f <file> / --readoutput <file>: Output file for reads (fasta or fastq format
     depending on file name). Only matching reads are output. With this option
     the output assembly refers to this read file instead of the original read
     files. When both paired and unpaired reads are output, use the -e option to
     speicify the name of the paired read file.

  -e <file> / --pairreadoutput <file>: Output file for paired reads when both
     paired and unpaired reads are output. I.e. when the assembly has both paired
     and unpaired reads, the -f option is used, and the -p and -q options are not
      used.

  -g <file> / --refoutput <file>: Output file for references. With this option
     the output assembly refers to this reference file instead of the original
     reference files.

  -p / --paired: Keep read pairs together when making read output file. Should
     be used when the reads are from a paired end experiment, but were assebled
     as unpaired. If asssembled as paired, the pairs will automatically be kept
     toegether. May only be used with the '-f' option.

Examples:

  Make an assembly containing only reference sequence two of an existing
  assembly. Also make a new file for the reads matching this sequence:

    sub_assembly -a assembly.cas -o new.cas -s 2 -f new_reads.fasta

  The same but only the first 100,000 positions of the reference sequence and
  also make a file for the new partial reference sequence

    sub_assembly -a assembly.cas -o new.cas -s 2 -b 1-100000 -f new_reads.fasta
                                                             -g new_ref.fasta

  Make an assembly without ambiguously placed reads:

    sub_assembly -a assembly.cas -o new.cas -u
```

A.15 Options for unassembled_reads

```
usage: unassembled_reads <options>

  Make a fasta file with the unassembled reads from an assembly.

Options:

  -h / --help: Display this message.

  -a <file> / --assembly <file>: Specify the assembly file (required).

  -o <file> / --output <file>: Specify the output fasta or fastq file
     (required).

  -l <n> / --minlength <n>: Output only sequences with a certain minimum
     length.

  -u / --unaligned: For matching reads with sufficiently long unaligned parts,
     output these parts as individual sequences. Two parts may be output if
     both ends are long enough. Must be used with the '-l' option.

  -p / --paired: Always treat the reads as paired, so if one read of a pair
     qualifies for reporting, report both reads. Cannot be used with the "-u"
     option.

Example:

  Make a fasta file with all the unassembled reads along with all read parts
  that were unaligned and has a length of at least 100 bp:

    unassembled_reads -a assembly.cas -o unassembled.fasta -l 100 -u
```

A.16 Options for tofasta

```
usage: tofasta <sequence file>

  Convert a sequence file to fasta format.

Options:

  -h / --help: Display this help
```

Bibliography

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

[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.

Index

AMD architectures, system requirements, 9

Intel architectures, 8

Linux, 8

Mac OS X, 8

NetBurst microarchitecture, 8

Pentium, system requirements, 8
Platforms supported, 8

Supported CPU architectures, 8
System requirements, 8

Windows, 8