Published in final edited form as:

Curr Protoc Bioinformatics. 2010 September ; CHAPTER: Unit-11.5. doi:
10.1002/0471250953.bi1105s31.

Using the Velvet *de novo* assembler for short-read sequencing technologies

Daniel R. Zerbino

Center for Biomolecular Science and Engineering 503A Engineering 2 CBSE/ITI UC Santa Cruz
1156 High Street Santa Cruz, CA 95064 dzerbino@soe.ucsc.edu Phone: 831-459-5232; EMBL-
EBI Wellcome Trust Genome Campus CB10 1SD Cambridge, UK

Abstract

The Velvet *de novo* assembler was designed to build contigs and eventually scaffolds from short read sequencing data. This protocol describes how to use Velvet, interpret its output and tune its parameters for optimal results. It also covers practical issues such as configuration, using the VelvetOptimiser routine and processing colorspace data.

INTRODUCTION

The Velvet *de novo* assembler (Zerbino *et al.*, 2008) can be used to quickly build long continuous sequences, or *contigs*, as well as gapped assemblies of contigs, or *scaffolds*, out of short-read datasets as produced by next-generation sequencing technologies. This function is mainly useful when studying data from a new organism for which a reference genome has not been assembled yet, or when trying to determine the origin of unmapped reads.

In short, Velvet builds a *de Bruijn* graph from the reads and removes errors from the graph (Zerbino *et al.*, 2008). It then tries to resolve repeats, based on the available information, whether long reads or paired-end reads (Zerbino *et al.*, 2009). It finishes by outputting an assembly of the reads, along with various statistics (see Understanding Results).

Velvet is centered around two programs, *velveth* and *velvetg*, which are always used together. They generally require some parameter tuning for each new dataset (see Basic Protocol, and Advanced Parameters). However, depending on the requirements of the experiment, it might be necessary to review some of the compilation parameters (see Support Protocol 1). To automate the process of parameter tuning, the VelvetOptimiser routine (see Internet resources) can be very useful (see Support Protocol 2). Finally, Velvet can be used to analyze colorspace data, although this requires specific settings and the use of adjunct conversion programs (see Support Protocol 3).

BASIC PROTOCOL: ASSEMBLING A SET OF READS WITH VELVET

Introduction

This basic protocol describes the basic Velvet assembly process which takes in short read sequences, and produces an assembly. A single assembly with Velvet typically happens in two steps: hashing and graph building. These steps correspond to the two Velvet executables, *velveth* and *velvetg* respectively. *Velveth* reads sequence files and builds a dictionary of all words of length *k*, where *k* is a user-defined parameter, thus defining exact local alignments between the reads. *Velvetg* then reads these alignments, builds a *de Bruijn* graph from them, removes errors and finally proceeds to simplify the graph and resolve repeats based on the parameters provided by the user.

Materials

Software—*velvetg* and *velveth* (see Support Protocol 1 for installation).

If you are analyzing colorspace reads, see Support Protocol 3 for specific comments.

A Linux/Unix/MacOSX command shell

Files—Sequence files in FastA, FastQ (Cock *et al.*, 2010), SAM or BAM format (Li *et al.*, 2009) (*the chosen format has no impact on results*)

1. Steps and Annotations

1. Preparing paired-end files—Although Velvet can function with single end reads, the use of paired-end reads is strongly recommended to obtain longer contigs, especially in repetitive regions. If all the data consists of single end reads, then this step can be skipped. The following procedure has already been applied to the test dataset which can be found in *data/test_reads.fa* and *data/test_long.fastq*.

Velvet requires that paired-end FASTA and FASTQ datasets come in a single merged file, where each read is paired with the one directly above or the one directly below. However, paired-end datasets are often provided as two separate FASTA or FASTQ files, and the reads are paired by their ordering.

To convert two FASTA (FASTQ) files into one, use the *shuffleReads_fasta.pl* (*shuffleReads_fastq.pl*) script:

```
shuffleReads_fasta.pl reads_1.fa reads_2.fa merged.fa
```

If your paired-end reads are contained in SAM or BAM files, then you must simply order those files by read name. Note that it is not necessary to ensure that reads appear only once or that every read in the file is paired. Velvet detects which reads are unpaired and handles them as such.

For a SAM file, the following is sufficient:

```
sort reads.sam > reads.sorted.sam
```

For a BAM file, use SAMTOOLS (Li *et al.*, 2009):

```
samtools sort -n reads.bam reads.sorted
```

2. Categorizing the reads—Velvet handles reads differently depending on their length, their pairing and their library. The first distinction is made between long and short reads. There is no strict rule to decide what is long and short, but long read alignments are stored on more detailed data structures, which take up more memory, but allow the system to completely reconstruct their path through the assembly. Typically, reads which are longer than 200 bp (e.g. 454 or capillary) would be marked as long, but if memory is insufficient, they can also be considered short.

Velvet then allows the user to separate the short reads in an arbitrary number of categories (see Support Protocol 1 to set the maximum value). This can be useful when wanting to compare read sets from different samples. More commonly, these categories are used to distinguish paired-end libraries of different lengths. Separate runs of paired-end reads made with the same insert length can be categorized together.

3. Choosing a hash length—The hash length is probably the single most important parameter of a Velvet run. See the Critical Parameters section for a detailed discussion of the choice of this value. In the example below the chosen hash length will be 21. In all generality, a good hash length would be between 21 bp and the average read length minus 10 bp.

4. Running *velveth*—Assuming you want Velvet to create an output directory called *directory*, the syntax for running *velveth* is:

```
velveth directory 21 [[<file_format> <file_category> file] ...]
```

In plain, you must provide in order the working directory name, the hash length, and a list of filenames, each preceded by its file format and its read type. The file format markers are: -fasta -fastq -sam -bam. The read category markers (see point 2) are: -long -longPaired -short -shortPaired -short2 -shortPaired2 etc. The CATEGORIES compilation parameter (see Support Protocol 1) determines how many -short* and -shortPaired* categories are available.

If either a file's format or the category of its reads is the same as the previous file, then the corresponding markers are unnecessary. A typical *velveth* command line would therefore look something like:

```
velveth directory 21 -fastq -short unpaired.fastq -shortPaired paired.fastq
```

For example, to process the test dataset the command line would be:

```
velveth directory 21 -fasta -long data/test_long.fa -shortPaired data/test_reads.fa
```

5. Running *velvetg*—For a basic heuristic run of *velvetg* do:

```
velvetg directory -exp_cov auto -cov_cutoff auto
```

The following two steps concern manual optimization of the assembly parameters. Note that the VelvetOptimiser wrapper can automate this procedure for you. See Support Protocol 2 for more details.

6. Optimizing the hash length—Repeat points 4-5 using different hash lengths. Some simple tips are provided in the Critical Parameters section.

7. Optimizing the other parameters—After running point 4 once with the optimal hash length, you can repeat point 5 with different options to determine the optimal parameters for *velvetg*.

In order, optimize the coverage cutoff, the expected insert length, and possibly the insert lengths. As a first approximation, you want to maximize the final N50 without losing too much of the total assembly length (this data is displayed on the screen as *velvetg* exits). See the Critical Parameters section for more information.

The optimal settings depend on a large number of experimental conditions: the length of the sequence being analyzed, its complexity, the number of reads, their quality, their length, etc. For this reason, the optimization step must generally be performed for each dataset.

The different runs of velvetg are independent, meaning that the options used in one run do not affect the next. For each run, the order of the parameters on the command line is indifferent.

As an example, if you want to set the coverage cutoff at 10x and the expected coverage at 30x, the command line is:

```
velvetg directory -cov_cutoff 10 -exp_cov 30
```

Returning to the test dataset, an appropriate command would be:

```
velvetg directory -cov_cutoff 5 -exp_cov 19 -ins_length 100
```

SUPPORT PROTOCOL: INSTALLING VELVET

Velvet is freely available for download under the form of source code that can be compiled then run on practically any system. This compilation stage is very straightforward and quick, but requires the user to set a number of important parameters.

Materials

Hardware—A system with as much physical memory as possible (12GB or more) is recommended. Velvet can in theory function on a 32bit environment, but such systems have memory limitations which might ultimately be a constraint for assembly. A 64bit system is therefore strongly recommended.

Software—Web browser

ANSI compliant C compiler (e.g. *gcc*) and a standard Unix shell such as *bash* or *tcsh*.

Velvet should function on any standard 64bit Linux environment with *gcc*. It has been tested on other systems such as MacOS, Sparc/Solaris or Windows (with Cygwin installed).

2. Steps and Annotations

1. Download the latest version of Velvet at www.ebi.ac.uk/~zerbino/velvet_latest.tgz
2. Unpack the tar file and move into the Velvet directory:

```
tar -xvzpf velvet_latest.tgz
```

```
rm velvet_latest.tgz
```

```
cd velvet*
```

3. Determine the longest hash length you intend on using.

If this is the first time you are using Velvet, simply use the default value, 32 bp. If you later realize that you are constrained by this parameter (cf. Basic Protocol, Critical Parameters), you can then re-compile Velvet with a higher value. It is generally most efficient to choose the lowest multiple of 32 that is above your maximum hash length. In the example below, the maximum hash length will be 32 bp.

4. Determine how many short-read paired-end libraries you are likely to analyze simultaneously. (cf. Basic Protocol).

Bear in mind that the higher you set this parameter, the more memory Velvet will require, so try to set it sparingly. In the example below, the number of paired-end

libraries is set at 2. Read pairs from different runs but with the same insert length distribution (either because they were produced from the same fragment library, or from different libraries selected for the same fragment size) can be considered as part of the same library.

5. Compile Velvet (if you intend on assembling colorspace SOLiD reads, refer to Support Protocol 2):
make 'MAXKMERLENGTH=32' 'CATEGORIES=2'
6. For convenience you can copy the newly created files *velveth* and *velvetg* into */usr/local/bin* (system administrator rights are necessary to do this) or alternatively set you *PATH* environment variable to point to the Velvet directory.

SUPPORT PROTOCOL: USING VelvetOptimiser

To simplify the search for optimal parameters, Simon Gladman and Torsten Seeman developed a script, VelvetOptimiser (see Internet Resources), which automatically scans the parameter space to produce the best possible assembly.

Materials

The requirements for this procedure are identical to that of the Basic Protocol. In addition, you will need:

Software

VelvetOptimiser (bundled with the Velvet package)

Perl version 5.8.8 or later.

BioPerl version 1.4 or later.

Basic Unix shell with *grep*, *sed*, *free* and *cut*.

Steps and annotations

1. Determine a range of *k*-mers to test. See Critical parameters for an estimate of appropriate values to test. In the example below, the range of *k*-mers will be from 16 to 31 bp.
2. Determine the appropriate file description line for *velveth* (cf. Basic Protocol, point 4). In the example below, this line will be “-short -fasta reads.fa”.
3. *Optional*: If you have an estimate of the genome size in megabases, you can obtain an estimate of the required memory. *Assuming the genome is around 50Mbp long*:
VelvetOptimiser.pl -s 16 -e 31 -f “-short -fasta reads.fa” -g 50
4. Run VelvetOptimiser:
VelvetOptimiser.pl -s 16 -e 31 -f “-short -fasta reads.fa”
5. Collect data. Upon exiting, VelvetOptimiser prints out the directory in which it left the output of its final Velvet assembly. This directory contains the standard Velvet output files.

SUPPORT PROTOCOL: PROCESSING COLORSPACE DATA

Users of SOLiD machines must be aware that the data produced by these machines is provided in a very specific format, colorspace. This difference is not just typographical but

information theoretic. Colospace files have very different properties in term of strandedness and of error rates. This is why it is recommended using colospace-compatible software throughout and only converting the data to conventional sequence-space at the very end of the pipeline. This unit describes the specific steps which users must take when installing and running Velvet to ensure compatibility with colospace data.

Materials

Software—Perl version 5.8.8 or later.

The ABI *de novo* tools (see Internet Resources), including `denovo_preprocessor_solid_v2.2.1.pl` and `denovo_postprocessor_solid_v1.6.pl`.

Optionally, if combining colospace reads with other data, the *Corona Lite* package (see Internet Resources)

Files—A set of colospace reads in csfasta format (*cf.* Internet resources). In the examples below, a paired-end dataset is represented by files `reads_1.csfasta` and `reads_2.csfasta`.

3. Steps and Annotations

1. Compile the colospace version of Velvet.

Follow the procedure described in Support Protocol 1, except that at stage 5, use the instruction:

```
make color 'MAXKMERLENGTH=32' 'CATEGORIES=2'
```

This produces two executable files, `velveth_de` and `velvetg_de`, which will be used instead of the conventional Velvet executables. You can optionally install these executables in `/usr/local/bin` (system administrator rights are necessary to do this).

2. If combining colospace reads with other datasets, you should convert these other reads to colospace. For this, the Corona Lite package contains the `encodeFasta.py` script which converts FastA files. To convert file `reads.fa`, do:

```
encodeFasta.py reads.fa > reads.csfasta
```

3. Convert the reads to double encoding.

Although double-encoded files contain sequences of A's, T's, C's and G's, it is important to distinguish them from standard FASTA files, since the letters do not correspond to nucleotides. Confusion between sequence files and double encoded files could lead to serious errors in the pipeline.

For unpaired reads run:

```
denovo_preprocessor_solid_v2.2.1.pl --run fragment -f3
```

```
reads_1.csfasta --dir double_encoded_files
```

For paired-end reads run:

```
denovo_preprocessor_solid_v2.2.1.pl --run mates -f2
```

```
reads_1.csfasta -r3 reads_2.csfasta --dir double_encoded_files
```

4. Run the protocols described in this unit, using the executable files `velveth_de` and `velvetg_de` instead of `velveth` and `velvetg` respectively.

5. When satisfied with the assembly results, run `velvetg` with the same options as your last run, but appending “`-amos_file yes`” at the end of the command line.

6. Finally convert the Velvet output into nucleotide sequences. If the Velvet working directory is *directory*, do:

```
denovo_postprocessor_solid_v1.6.pl -afgfile
directory/velvet_asm.afg -csfasta
reads_1.csfasta --csfasta reads_2.csfasta -output
sequence_space_contigs.fa
```

GUIDELINES FOR UNDERSTANDING RESULTS

Files

In its output directory (as specified on the command line of *velveth* and *velvetg*) Velvet produces a number of files, including:

- *contigs.fa* : Contig sequences in FASTA format,
- *stats.txt* : a tab-separated table with statistics on the contigs,
- *velvet_asm.afg* : Assembly file (compatible with AMOS, see the Advanced Parameters section for more information on how to create and use it).
- Sequences : a modified fasta file which contains the original sequence names (as they appear in the input files) and the corresponding Velvet read ID numbers.

The contigs contained in the FASTA file are in fact scaffolds and contain variable length gaps represented by sequences N's. The length of the sequence corresponds to the estimated gap length. However, for compatibility issues with the NCBI database, all gaps are represented as at least 10 bp long, even if the distance estimate is shorter. The AFG file, because its format is more flexible, contains all the scaffolding information explicitly.

The AFG file contains information on the inferred mapping of the reads onto the contigs. Because Velvet constructs, whenever possible, contigs through repeated regions, it sometimes cannot reliably assign reads to their respective repeat copies. This is why the coverage of repeated regions can drop to zero within a contig, and the average contig coverage depths drop accordingly.

The de Bruijn graph structure allows a read to be fragmented into several *k*-mers, which are separately mapped onto different contigs. A single read can therefore be mapped onto several contigs, essentially connecting the contigs. Normally, Velvet would try to merge the two contigs, but it sometimes leaves such connections untouched, in the absence of sufficient evidence.

Units

Velvet measures and reports lengths in overlapping *k*-mers. Although not intuitive at first sight, this unit system allows for consistency throughout Velvet's output. Therefore, if a contig is reported as being *L* *k*-mers long, its sequence in the *contigs.fa* file is in fact *L + k - 1* basepairs long.

Similarly, statistics derived from lengths are also subjected to this transformation. If the median coverage of an assembly is reported as C_k read *k*-mers per contig *k*-mer it is corresponds in fact to roughly $C_k L / (L + k - 1)$ read basepair per contig basepair (assuming that contigs are significantly longer than the hash length).

The coverage distribution

The distribution of average contig (or node) coverage, represented in the stats.txt file, provides a quick initial glimpse into the content of an assembly.

The coverage of individual unique k -mers should normally obey a Poisson distribution, as shown on Figure 1. Velvet does not allow you to directly measure this k -mer coverage, but you can use *tallymer* (Kurtz *et al.*, 2008) to produce this distribution. In practice, the observed distribution differs in three ways, also shown on Figure 1. First, because of cloning bias (typically associated to GC content) the variance of the observed distribution is larger than expected. Second, occasional random errors create a large number of words which are only observed very few times. This creates a sharp peak of k -mers with very low coverage. Third, genomes generally contain some repeated sequences. This produces secondary peaks whose mean coverage are multiples of the main peak's mean. The height of these peaks generally decreases sharply with repeat multiplicity.

The statistics provided in Velvet's stats.txt file correspond to average contig coverages. Because of this operation, the initial Poisson distribution of the k -mer coverage converges towards a normal distribution with the same mean, but with a variance which decreases proportionally to the inverse of the contigs' length. As the contigs get longer, the width of the peaks should get narrower, as shown on Figure 1.

The first thing to look for is the separation of the peak of genuine k -mer coverage from that of false k -mers. If it is as clean as shown on the figure, then the run was probably very successful and a simple coverage cutoff would presumably be very efficient in clearing out any uncorrected error. On the contrary, if the expected coverage is very low, or the error rate very high, then this separation will be blurred, and the assembly will suffer.

Anomalies in the coverage distribution can be indicative of perturbations of the sequencing process. For example, contamination would create extra peaks. Uneven DNA concentrations in the sample (for example nuclear DNA vs. mitochondrial DNA) would create separate sets of peaks. Unusually high cloning bias would increase the variance of the peaks.

COMMENTARY

Background Information

For many years, sequence assembly algorithms were mainly designed around the overlap-layout-consensus (Pevzner *et al.*, 2001). This approach builds an assembly from the pairwise alignment of all the reads. With capillary reads, this approach was very successful, and produced many high quality genome assemblies.

However, assembling short reads with these programs proved very costly. The redundancy of these datasets caused the quadratic number of read alignments to become prohibitively costly to compute, and the short length of the reads reduced the statistical significance of overlaps between reads.

This spurred the development of another category of assemblers, based on the use of the de Bruijn graph, among which *EULER* (Pevzner *et al.*, 2001; Chaisson *et al.*, 2009), *ALLPATHS* (Butler *et al.*, 2008), *ABYSS* (Simpson *et al.*, 2009), *SOAPdenovo* (Li *et al.*, in press) and *Velvet* (Zerbino and Birney, 2008). To cut down on computation time, these approaches simply search for exact word matches, and only at a later stage try to separate false positive alignments.

Critical Parameters

The main parameter which the user must set is the hash length. Velvet can generally derive all the other parameters once this one is given. One approach is to use scripts such as the *VelvetOptimiser* (cf. Support Protocol 2) to scan a set of possible values. However it is possible to determine a near-optimal hash length with some simple estimates.

The hash length, sometimes referred to as k -mer length or word length, corresponds to the length of the words which are stored and compared in the construction of the de Bruijn graph. The hash length must be an odd number which is shorter than most reads in the dataset. All reads shorter than the hash length are simply ignored throughout the assembly. Finding the optimal hash length is essentially striking a compromise between sensitivity and specificity.

On one hand, the longer the hash length, the less likely words will be exactly repeated in the genome being sequenced. Similarly, spurious errors are less likely to overlap with other sequences if the word length is increased, making them easier to detect and remove. A large hash length therefore leads to a simpler de Bruijn graph with fewer tangles and fewer errors.

On the other hand, the longer the hash length, the fewer times each word will be observed in the dataset. Neglecting errors and repeats, the expected number of times a k -mer is observed in a sequencing dataset, or k -mer coverage C_k , can be derived from the common coverage depth C , the average read length L , and the hash length k , using the following formula:

$$C_k = C \frac{L - K}{L}$$

As the hash length k gets close to L , C_k can drop quickly, thus creating more coverage gaps in the assembly. A short hash length therefore gives the assembler more sensitivity to fill in gaps.

Experience shows that optimal results are obtained if k is adjusted so that C_k is close to 15x. The function which associates a contig N50 to each hash length generally has a very sharp bell shape. It is therefore generally quite easy to find the optimum value with a few iterations.

It is quite common to observe that the N50 rises steadily as the hash length reaches the maximum length allowed by Velvet. If that is the case, you simply need to re-compile Velvet with a higher MAXKMERLENGTH parameter (cf. Support Protocol 1).

Two more important parameters are the expected coverage and coverage cutoff. They can be determined automatically by Velvet. This behavior is triggered by adding the following options to the *velvetg* command line (see Basic Protocol):

```
velvetg directory (...) -exp_cov auto -cov_cutoff auto
```

Generally, this process works smoothly, but it is worth keeping an eye on Velvet's estimates. If for some reason its estimates are flawed, then the user should override them manually.

The expected coverage is crucial, as it allows Velvet to determine which contigs correspond to unique regions of the genome and which contigs correspond to repeats. As explained above, the expected coverage can be determined using a probabilistic formula. However, it does not take into account the error rate which decreases the effective coverage. It is

generally more reliable to observe the coverage distribution after a first assembly then enter it manually in the *velvetg* command line.

The coverage cutoff is a simple but effective way of removing many basic errors which passed the previous filters. It simply removes contigs with an average cutoff below a given threshold. It must therefore be set so as to remove more errors than it removes genuine sequence. If the coverage threshold is too high, then correct contigs are removed from the assembly, potentially creating misassemblies. If determined automatically, this cutoff is simply set to half the expected coverage.

Troubleshooting

The main obstacle encountered by Velvet users is lack of memory. Depending on the system, the computer will either slow down significantly as it starts swapping memory onto the hard drive, or Velvet will simply stop, printing a short message on the standard output. A number of steps can be used to fit a dataset onto the available hardware.

The first step consists in eliminating unnecessary redundancy in the dataset. Velvet requires typically 15x *k*-mer coverage to function properly. However, higher coverage is not necessarily beneficial to the assembly stage. It is therefore possible to temporarily discard part of the dataset, and only assemble the remaining reads. The entire dataset can then be aligned to the assembled contigs to obtain variation data.

The second step consists in eliminating low quality information. Erroneous sequences do not generally overlap with each other, so each individual error creates its own independent data structures. The error rate therefore has a significant impact on the memory footprint of Velvet. If memory is an issue but coverage is sufficient, you can trim the reads according to quality, to conserve a smaller but more reliable dataset.

Finally, it is useful to take into account the effect of the hash length on the memory usage. The closer the hash length is to the optimum value, the more compact the de Bruijn graph, the lower the memory consumption. Therefore, a proper setting of the hash length can help fit a dataset into the computer.

Another issue is misassemblies. In many cases, this can be linked to the parameterization. For example, the automatic measurement of coverage or insert length can fail on highly fragmented datasets, in which case it is necessary to enter the corresponding manually (check the standard output on the screen for the relevant information). Another issue is the specificity of long reads or paired-end information. At high coverage, it may be necessary to raise the *-min_pair_count* or *-long_mult_cutoff* thresholds to filter out spurious connections between contigs. These parameters determine how many short read pairs or long reads are necessary to connect two contigs.

Advanced Parameters

By default, Velvet automatically estimates the insert length of the different libraries present in the assembly. This approach relies on paired reads which land on the same contigs, thus introducing an interval censoring bias. In other words, if the expected insert length is longer than most contigs, then only read pairs with a short insert length will be observed on the same contigs. This is why it is recommended to check the estimates as they are printed in the standard output. If you have prior knowledge which indicates that these estimates are wrong, then you can simply override them by setting the average insert length manually. For example to set the insert length of the third library to 3200 bp:

```
velvetg directory (...) -ins_length3 3200
```

If you manually provide this insert length, then Velvet uses this value, and assumes that the standard deviation is 10% of the expected value. If you are using several insert libraries, and you know that the standard deviation of one library's insert length is more important relative to its expected value, then you can indicate this to Velvet. For example, if the above library has a insert length standard deviation of 1000 bp, then the command line becomes:

```
velvetg directory (...) -ins_length3 3200 -ins_length3_sd 1000
```

You can provide as many insert lengths and standard deviations as there were libraries (or categories) provided to *velveth*, changing the flag's number accordingly: *-ins_length2*, *-ins_length5_sd*, etc.

If you are using pre-assembled contigs as long reads, then it is necessary to lower the *-long_mult_cutoff* parameter to 0. This is because to reduce noise Velvet requires a minimum number (by default 2) of long reads to connect two contigs before they are merged. In the case of pre-computed contigs, these "long reads" do not overlap, and the above constraint prevents Velvet from using them to resolve repeats.

You can also request extra outputs from Velvet to suit your pipeline. For example, if you wish to know which reads were not used in the assembly, then add the following flag to the command line:

```
velvetg directory (...) -unused_reads yes
```

This will produce an *UnusedReads.fa* file in the working directory, containing all the desired reads.

If you wish to visualize the assembly, you can request the production of an AFG assembly file called *velvet_asm.afg*, which can be handled or converted by a number of tools (see Internet Resources).

```
velvetg directory (...) -amos_file yes
```

Literature

- Butler J, MacCallum I, Kleber M, Shlyakhter IA, Belmonte MK, Lander ES, Nusbaum C, Jaffe DB. ALLPATHS: De novo assembly of whole-genome shotgun microreads. *Genome Research*. 2008; 18:810–820. [PubMed: 18340039]
- Cock PJ, Fields CJ, Goto N, Heuer ML, Rice PM. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *N.A.R.* 2010; 38(6):1767–1771.
- Chaisson MJ, Brinza D, Pevzner PA. De novo fragment assembly with short mate-paired reads: Does the read length matter? *Genome Research*. 2009; 19:336–346. [PubMed: 19056694]
- Li H, Handsaker B, Wysoker A, Fennel T, Ruan J, Homer N, Marth G, Abecasis G, Durbin R, the 1000 Genome Project Data Processing Subgroup. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*. 2009; 25(16):2078–9. [PubMed: 19505943]
- Li R, Zhu H, Wang J. *De novo* assembly of human genomes with massively parallel short read sequencing. *Genome Res*. In press.
- Kurtz S, Narechania A, Stein J, Ware D. A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics*. 2008; 9(1):517. [PubMed: 18976482]
- Pevzner PA, Tang H, Waterman MS. An Eulerian path approach to DNA fragment assembly. *Proc. Nati. Acad. Sci. USA*. 2001; 98(17):9748–9753.
- Simpson JT, Wong K, Jackman SD, Schein JE, Jones SJ, Birol I. ABySS: A parallel assembler for short read sequence data. *Genome Research*. 2009; 19(6):1117–23. [PubMed: 19251739]

4. Key References

- Zerbino DR, Birney E. Velvet: algorithms for *de novo* short read assembly using de Bruijn graphs. *Genome Res.* 2008; 18:821–829. [PubMed: 18349386] This first publication mainly described the implementation of de Bruijn graphs within Velvet and the error correction algorithm, *TourBus*.
- Zerbino DR, McEwen GK, Margulies EH, Birney E. Pebble and Rock Band: Heuristic Resolution of Repeats and Scaffolding in the Velvet Short-Read *de Novo* Assembler. *PLoS ONE.* 2010; 4(12):e8407. [PubMed: 20027311] This follow-up paper describes how Velvet resolves complex repeats using long reads or paired-end read information.

5. Internet Resources

- Velvet website** Code and information on Velvet can be downloaded at:
www.ebi.ac.uk/~zerbino/velvet
- Gladman, Simon; Seeman, Torsten. VelvetOptimiser. This wrapper software scans different parameters of Velvet to produce an optimal assembly, as described in Support Protocol 2.
<http://bioinformatics.net.au/software.shtml>
- Cummings, Craig; Sheth, Vrunda; Brinza, Dima. Colospace *de novo* pipeline. These scripts allow you to do all the appropriate colospace conversions described in Support Protocol 3 (a registration is required, but the software is free): <http://solidsoftwaretools.com/gf/project/denovotools/> The Coronal Lite package can also be found on that server:
<http://solidsoftwaretools.com/gf/project/corona/>
- the AMOS Consortium. AMOS suite. This suite of tools allows the user to manipulate, convert or analyze AFG assembly files: <http://sourceforge.net/apps/mediawiki/amos/>
- Colospace documentation. Applied Biosystems This documents describes colospace, and the csfasta format in particular:
http://tools.invitrogen.com/content/sfs/manuals/SOLiD_SAGE_SoftwareGuide.pdf

Figure 1. Schematic representation of the coverage distribution

The solid curve represents the expected k -mers coverage distribution in an idealized sequencing project of a repeat-free genome, i.e. a Poisson distribution. The dashed curve schematically represents the k -mer coverage distribution in a typical experimental situation. The variance of the main peak is increased, and new peaks are created by erroneous k -mers (left-most peak with minimal coverage) and repeated coverage (smaller peaks on the right). Finally, the dotted curve schematically represents the distribution after a successful assembly. The width of the peaks is tightened because of the averaging over whole contigs.