**Velvet de novo Assembler for Short-Read Sequencing Technology**

User manual

**How to get Velvet**

Velvet is available for free download on Github

> git clone <https://github.com/dzerbino/velvet.git>

**Or download the latest version**

[www.ebi.ac.uk/~zerbino/velvet\_latest.tgz](http://www.ebi.ac.uk/~zerbino/velvet_latest.tgz" \t "pmc_ext)

**Requirements**

Velvet should work on any 64bit Linux environment with gcc. A physical memory of 12GB is recommended for best performance.

**To compile velvet**

cd to velvet repository and type:

> make

To use colorspace Velvet, type:

> make color

The git repository includes a manual in the source directory, if you can’t find it, you can compile it by typing:

> make doc

**Assembling a set of reads with Velvet**

**Basic protocol**

Takes in short read sequences, and produces an assembly. A single assembly happenes in two steps: hashing and building a graph. *Velveth* reads sequence files and creates a dictionary of all words of length k, k is user-defined parameter, and therefore the user can define exact local alignments between reads. *Velvetg* reads the alignments, then builds a de Bruijn graph, removes error and repeats then simplify the graph based on user-defined parameters.

**Files**: Sequence files in FastA, FastQ, SAM OR BAM format (the chosen format doesn’t change the results)

**1. Steps and annotations**

**1.1) Preparing paired-end files**

Velvet can function with single end reads, but using paired-end reads is strongly recommended to obtain longer contigs, especially when having repetition in some regions. Velvet requires that paired-end FASTA and FASTQ datasets to come in one merged file. In this case “ColE9-J.fastq” is a single file so we can just use it.

**1.2) 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. The “ColE9-J.fastq” contains 2500 reads each of 150bp so they will be considered short reads.

**1.3) Choosing a hash length**

The hash length is probably the single most important parameter of a Velvet run. In the example below the chosen hash length will be 21. In all generality, a good hash length would between 21bp and the average read length minus 10bp.

**1.4) Running *velveth***

Assuming you want Velvet to create an output directory called directory, syntax used is:

velveth directory 21 [[<file\_format><file\_category> file] …]

File formats are: -fasta –fastq –sam –bam

File categories are: -long –longPaired –short –shortPaired –short2 –shortPaired2 etc

Example:

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

**1.5) Running velvetg**

For a basic heuristic run of velvetg do:

velvetg directory –exp\_cov auto –cov\_cutoff auto

**1.6) Optimizing parameters**

In order to optimize the coverage cutoff, the expected insert length and possibly the insert lengths. 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

**Support protocol:** Using VelvetOptimizer

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.
2. *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

1. Run VelvetOptimiser:

VelvetOptimiser.pl –s 16 –e 31 –f “-short –fasta reads.fa”

1. 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.

Guideline to understanding results

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

* contigs.fa: Contig sequences in FASTA format
* stats.txt: a tab-separated table with statistics on the contigs
* velvet\_asm.afg: Assembly file
* Sequences: A modified fasta file which contains the original sequence names (as they appear in the input file) and the corresponding Vevlet read ID number

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 10bp 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 though 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 *Ck* read *k*-mers per contig *k*-mer it is corresponds in fact to roughly *CkL*/(*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.

**Critical parameters**

The main parameter that the user set is the hash length. Velvet can derive all other parameters once this one is given. 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 ignored throughout the assembly.

\* If you tried to use even k-mer length velvet will print a message like this:

[0.000001] Velvet can't work with even length k-mers, such as 14. We'll use 13 instead, if you don't mind.