Velvet Manual - version 0.7

Daniel Zerbino

August 29, 2008

Contents

1 For impatient people

```
> make
> ./velveth
> ./velvetg

> ./velveth sillyDirectory 21 -shortPaired data/test_reads.fa
> ./velvetg sillyDirectory
(Final graph has 16 nodes and n50 of 24184 max 44966)
> less sillyDirectory/stats.txt

> ./velvetg sillyDirectory -cov_cutoff 5 -read_trkg yes -amos_file yes
(Final graph has 1 nodes and n50 of 99975 max 99975)
> less sillyDirectory/velvet_asm.afg

> ./velvetg sillyDirectory -exp_cov 19 -ins_length 100
(Final graph has 12 nodes and n50 of 99975 max 99975)

> ./velveth sillyDirectory 21 -short data/test_reads.fa -long data/test_long.fa
> ./velvetg sillyDirectory -exp_cov 19
(Final graph has 2 nodes and n50 of 99893 max 99893)
```

2 Installation

2.1 Requirements

Velvet should function on any standard 64bit Linux environment with gcc. A good amount of physical memory (12GB to start with, more is no luxury) is recommended.

It can in theory function on a 32bit environment, but such systems have memory limitations which might ultimately be a constraint for assembly.

2.2 Compiling instructions

From a GNU environment, simply type:

```
> make
```

3 Running instructions

3.1 Running velveth

Velveth helps you construct the dataset for the following program, velvetg, and indicate to the system what each sequence file represents.

If, on the command line, you forget the syntax, you can print out a short help message:

```
> ./velveth
```

Velveth takes in a number of sequence files, produces a hashtable, then outputs two files in an output directory (creating it if necessary), Sequences and Roadmaps, which are necessary to velvetg. The syntax is as follows:

```
> ./velveth output_directory hash_length
            [[-file_format][-read_type] filename]
```

The hash length, also known as k-mer length, corresponds to the length, in base pairs, of the words being hashed. See 5.1 for a detailed explanation of how to choose the hash length.

Supported file formats are:

fasta (default)

fastq

fasta.gz

fastq.gz

eland

gerald

Read categories are:

short (default)

shortPaired

short2 (same as short, but for a separate insert-size library)

shortPaired2 (see above)

long (for Sanger, 454 or even reference sequences)

longPaired

For concision, options are stable. In other words, they are true until contradicted by another operator. This allows you to write as many filenames as you wish without having to re-type identical descriptors. For example:

```
> ./velveth output_directory/ 21 -fasta -short solexa1.fa solexa2.fa solexa3.fa -long
capillary.fa
```

In this example, all the files are considered to be in FASTA format, only the read category changes. However, the default options are "fasta" and "short", so the previous example can also be written as:

```
> ./velveth output_directory/ 21 solexa*.fa -long capillary.fa
```

3.2 Running velvetg

Velvetg is the core of Velvet where the de Bruijn graph is built then manipulated. Note that although velvetg saves some files during the process to avoid useless recalculations, the parameters are **not** saved from one run to the next. Therefore:

```
> ./velvetg output_directory/ -cov_cutoff 4
> ./velvetg output_directory/ -min_contig_lgth 100
```

. . . is different from:

```
> ./velvetg output_directory/ -cov_cutoff 4 -min_contig_lgth 100
```

This means you can freely play around with parameters, without re-doing most of the calculations:

```
> ./velvetg output_directory/ -cov_cutoff 4
> ./velvetg output_directory/ -cov_cutoff 3.8
> ./velvetg output_directory/ -cov_cutoff 7
> ./velvetg output_directory/ -cov_cutoff 10
> ./velvetg output_directory/ -cov_cutoff 2
```

On the other hand, within a single velvetg command, the order of parameters is not important.

Finally, if you have any doubt at the command line, you can obtain a short help message by typing:

```
> ./velvetg
```

3.2.1 Single reads

Initally, you simply run:

```
> ./velvetg output_directory/
```

This will produce a fasta file of contigs and output some statistics. Experience shows that there are many short, low-coverage nodes left over from the intial correction. Determine as you wish a coverage cutoff value (cf. 5.2), say 5.2x, then type:

```
> ./velvetg output_directory/ -cov_cutoff 5.2
```

On the other hand, if you want to exclude highly covered data from your assembly (e.g. plasmid, mitochondrial, and chloroplast sequences) you can use a maximum coverage cutoff:

```
> ./velvetg output_directory/ -max_coverage 300 (... other parameters ...)
```

3.2.2 Adding long reads

Reminder: you must have flagged your long reads as such when running velveth (cf. 3.1).

If you have a sufficient coverage of short reads, and any quantity of long reads (obviously the more the coverage and the longer the reads, the better), you can use the long reads to resolve repeats in a greedy fashion.

To do this, Velvet needs to have a reasonable estimate of the expected coverage *in short reads* of unique sequence (see 5.1 for a definition of k-mer coverage). The simplest way to obtain this value is simply to observe the distribution of contig coverages (as described in 5.3), and see around which value the coverages of nodes seem to cluster (especially the longer nodes in your dataset). Supposing the expected coverage is 19x, them you indicate it with the *exp_cov* marker:

```
> ./velvetg output_directory/ -exp_cov 19 (... other parameters ...)
```

3.2.3 Paired-ends reads

Reminder: you must have flagged your reads as being paired-ends when running velveth (cf. 3.1).

To activate the use of read pairs, you must specify two parameters: the *expected* (i.e. average) insert length (or at least a rough estimate), and the expected *short-read k-mer coverage* (see 5.1 for more information). If you expect your insert length to be around 400bp, and your coverage to be around 21.3x, you would type:

```
> ./velvetg output_directory/ -ins_length 400 -exp_cov 21.3
          (... other parameters ...)
```

If you happen to have hashed paired long reads and you ordered them as explained in 4.1 you can also tell Velvet to use this information for scaffolding by indicating the corresponding insert length (remember that you still need to indicate the short-read k-mer coverage):

```
> ./velvetg output_directory/ -exp_cov 21 -ins_length_long 40000
 (... other parameters ...)
```

Standard deviations This is a more subtle point which you can ignore if you have only one dataset of paired-end reads or if the standard deviation (SD) of the insert lengths is roughly proportional to the expected length (e.g. if the insert-lengths are described as $length \pm p\%$).

Velvet does not use the absolute values of the insert-length SDs, but their relative values. Therefore, you do not need to spend too much time on the estimation of the SDs, as long as you are consistent. You can then enter your own *a priori* SD's. To do so simply indicate them as follows:

```
> ./velvetg output_directory/ -exp_cov 21
-ins_length 200 -ins_length_sd 20
```

```
-ins_length2 20000 -ins_length2_sd 5000
-ins_length_long 40000 -ins_length_long_sd 1000
 (... other parameters ...)
```

3.2.4 Controlling Velvet's output

Selecting contigs for output By default, Velvet will print out as many contigs as possible. This has the drawback of potentially flooding the output with lots of unwanted very short contigs, which are hardly useable in a significant way. If you wish, you can request that the contigs in the contigs.fa file be longer than a certain length, say 100bp:

```
> ./velvetg -min_contig_lgth 100 (... other parameters ...)
```

Using read tracking Velvet's read tracking can be turned on with the read-tracking option. This will cost slightly more memory and calculation time, but will have the advantage of producing in the end a more detailed description of the assembly:

```
> ./velvetg output_directory/ -read_trkg yes  (... other parameters ...)
```

Producing an .afg file If you turn on the read tracking, you might also want to have all the assembly information in one datastructure. For this purpose Velvet can produce AMOS files (cf 4.2.3). Because the .afg files tend to be very large, they are only produced on demand:

```
> ./velvetg output_directory/ -amos_file yes  (... other parameters ...)
```

Using multiple categories You can be interested in keeping several kinds of short read sets separate. For example, if you have two paired-end experiments, with different insert lengths, mixing the two together would be a loss of information. This is why Velvet allows for the use of 2 short read channels (plus the long reads, which are yet another category).

To do so, you simply need to use the appropriate options when hashing the reads (see 3.1). Put the shorter inserts in the first category. Supposing your first readset has an insert length around 400bp and the second one a insert length around 10,000bp, you should type:

```
> ./velvetg output_directory/  -ins_length 400 -ins_length2 10000
(... other parameters ...)
```

Note: Increasing the amount of categories is possible. It's simply a bit more expensive memory-wise.

Note: In the stats.txt file, you will find all three categories (long, short1 and short2) treated separately.

3.3 Advanced parameters: Tour Bus

Caveat Emptor

> The following parameters are probably best left untouched. If set unwisely, Velvet's behaviour may be unpredictable.
>
> Nonetheless, some users are curious to control the way in which Tour Bus (cf. 6) decides whether to merge polymorphisms or not.
>
> Before we go into the actual details, it is worth discussing the pros and cons of bubble smoothing. The original idea is that a few SNPs, in the case of diploid assembly, should not prevent the construction of an overall contig. Detecting them post assembly is just a matter of scanning the assembly files and detecting discrepancies between the consensus sequence and the reads.
>
> On the other hand, if you have two copies of a repeat in a haploid genome, you want to reduce the merging to a minimum, so that later analysis with paired-end reads or long reads may allow you to retrieve both individual copies, instead of just one artificial "consensus" sequence.
>
> Hopefully, these issues will eventually be resolved by further thought and experiment. In the mean time, Velvet allows direct access to these parameters for those who want to play around, or maybe tailor Velvet to specific needs (e.g. multi-strain sequencing).

Maximum branch length Partly for engineering issues and partly to avoid aberrant transformations, there is a limit as to how long two paths must before simplification. By default, it is set a 100bp. This means that Velvet will not merge together two sequences which are sufficiently divergent so as not to have any common k-mer over 100bp. If you want to allow greater simplifications, then you can set this length to, say, 200bp:

```
> ./velvetg output_directory/ -max_branch_length 200 (...other parameters...)
```

Maximum indel count Before comparing two sequences, Velvet compares their lengths, and, if they are too different, aborts the inspection. By default, this parameter is set to 3bp, but you can change it to, for example, 6bp:

```
> ./velvetg output_directory/ -max_indel_count 6 (...other parameters...)
```

Maximum divergence rate After aligning the two sequences with a standard dynamic alignment, Velvet compares the number of aligned pairs of nucleotides to the length of the longest of the two sequences. By default, Velvet will not simplify two sequences if they are more than 20% diverged. If you want to change that limit to 33%:

```
> ./velvetg output_directory/ -max_divergence 0.33 (...other parameters...)
```

Maximum gap count After aligning the two sequences with a standard dynamic alignment, Velvet compares the number of aligned pairs of nucleotides to the length of the longest of the two sequences. By default, Velvet will not simplify to sequences if more than 3bp of the longest sequence are unaligned.

```
> ./velvetg output_directory/ -max_gap_count 5 (...other parameters...)
```

3.4 Advanced parameters: Tour Bus

Minimum read-pair validation Velvet will by default assume that paired end reads are perfectly placed. With experimental data this assumption can be contradicted by occasional mis-pairings or by incorrect mappings of the reads because of errors. To avoid being misled by random noise, and therefore avoid missassemblies, Velvet requires that a connection between two contigs be corroborated by at least 10 mate pairs. If you want to change this cutoff to, say, 20, simply type:

```
> ./velvetg output_directory/ -min_pair_count 20 (...other parameters...)
```

4 File formats

4.1 Input sequence files

Velvet works mainly with fasta and fastq formats.

For paired-end reads, the assumption is that each read is next to its mate read. In other words, if the reads are indexed from 0, then reads 0 and 1 are paired, 2 and 3, 4 and 5, etc.

If for some reason you have forward and reverse reads in two different FASTA files but in corresponding order, the bundled Perl script shuffleSequences.pl will merge the two files into one as appropriate. To use it, type:

```
> ./shuffleSequences.pl forward_reads.fa reverse_reads.fa output.fa
```

Concerning read orientation, Velvet expects paired-end reads to come from opposite strands facing each other, as in the traditional Sanger format. If you have paired-end reads produced from circularisation (i.e. from the same strand), it will be necessary to replace the first read in each pair by its reverse complement before running velveth.

4.2 Output files

After running Velvet you will find a number of files in the output directory:

4.2.1 The contigs.fa file

This fasta file contains the sequences of the contigs longer than $2k$, where k is the word-length used in velveth. If you have specified a *min_contig_lgth* threshold, then the contigs shorter than that value are omitted.

Note that the length and coverage information provided in the header of each contig should therefore be understood in k-mers and in k-mer coverage (cf. 5.1) respectively.

4.2.2 The stats.txt file

This file is a simple tabbed-delimited description of the nodes. The column names are pretty much self-explanatory. Note however that node lengths are given in *k-mers*. To obtain the length in nucleotides of each node you simply need to add $k - 1$, where k is the word-length used in velveth.

The *in* and *out* columns correspond to the number of arcs on the 5' and 3' ends of the contig respectively.

The coverages in columns *short1_cov*, *short1_Ocov*, *short2_cov*, and *short2_Ocov* are provided in k-mer coverage (5.1).

Also, the difference between $*$*_cov* and $*$*_Ocov* is the way these values are computed. In the first count, slightly divergent sequences are added to the coverage tally. However, in the second, stricter count, only the sequences which map perfectly onto the consensus sequence are taken into account.

4.2.3 The velvet_asm.afg file

This file is mainly designed to be read by the open-source AMOS genome assembly package. Nonetheless, a number of programs are available to transform this kind of file into other assembly file formats (namely ACE, TIGR, Arachne and Celera). See http://amos.sourceforge.net/ for more information.

The file describes all the contigs contained in the contigs.fa file (cf 4.2.1).

If you are overwhelmed by the size of the file, two bundled scripts provided by Simon Gladman can help you out:

- asmbly_splitter.pl breaks down the original .afg file into individual files for each contig,
- snp_view.pl allows you to print out a simple ASCII alignment of reads around a given position on a contig.

4.2.4 The LastGraph file

This file describes in its entirety the graph produced by Velvet, in an idiosyncratic format which evolved with my PhD project. The format of this file is briefly as follows:

- One header line for the graph:

```
$NUMBER_OF_NODES    $NUMBER_OF_SEQUENCES    $HASH_LENGTH
```

- One block for each node:

  ```
  NODE    $NODE_ID    $COV_SHORT1    $O_COV_SHORT1  $COV_SHORT2  $O_COV_SHORT2
  $ENDS_OF_KMERS_OF_NODE
  $ENDS_OF_KMERS_OF_TWIN_NODE
  ```

  Note that the ends of k-mers correspond to the last nucleotides of the k-mers in the node. This means that the two sequences given above are not reverse-complements of each other but reverse complements shifted by k nucleotides. The common length of these sequences is equal to the length of the corresponding contig *minus* $k-1$.

  See 4.2.2 for an explanation of O_COV values.

- One line for each arc:

  ```
  ARC $START_NODE    $END_NODE    $MULTIPLICITY
  ```

  Note: this one line implicitly represents an arc from node A to B and another, with same multiplicity, from -B to -A.

- For each long sequence, a block containing its path:

  ```
  SEQ $SEQ_ID
  $NODE_ID $OFFSET_FROM_START $START_COORD $END_COORD $OFFSET_FROM_END
  $NODE_ID2 etc.
  ```

  The offset variables are distances from the edges of the nodes whereas the start and end coordinates are correspond to coordinates within the read sequence.

- If short reads are tracked, for every node a block of read identifiers:

  ```
  NR $NODE_ID $NUMBER_OF_SHORT_READS
  $READ_ID $OFFSET_FROM_START_OF_NODE $START_COORD
  $READ_ID2 etc.
  ```

5 Practical considerations / Frequently asked questions

5.1 Choice of hash length k

The hash length is the length of the k-mers being entered in the hash table.

Firstly, you must observe three technical constraints:

- it must be an odd number, to avoid palindromes. If you put in an even number, Velvet will just decrement it and proceed.
- it must be below or equal to 31, because it is stored on 64 bits
- it must be strictly inferior to read length, otherwise you simply will not observe any overlaps between reads, for obvious reasons.

Now you still have quite a lot of possibilities. As is often the case, it's a trade-off between specificity and sensitivity. Longer kmers bring you more specificity (i.e. less spurious overlaps) but lowers coverage (cf. below)... so there's a sweet spot to be found with time and experience.

We like to think in terms of "k-mer coverage", i.e. how many times has a k-mer been seen among the reads. The relation between k-mer coverage C_k and standard (nucleotide-wise) coverage C is $C_k = C * (L - k + 1)/L$ where k is your hash length, and L you read length.

Experience shows that this kmer coverage should be above 10 to start getting decent results. If C_k is above 20, you might be "wasting" coverage. Experience also shows that empirical tests with different values for k are not that costly to run!

5.2 Choice of a coverage cutoff

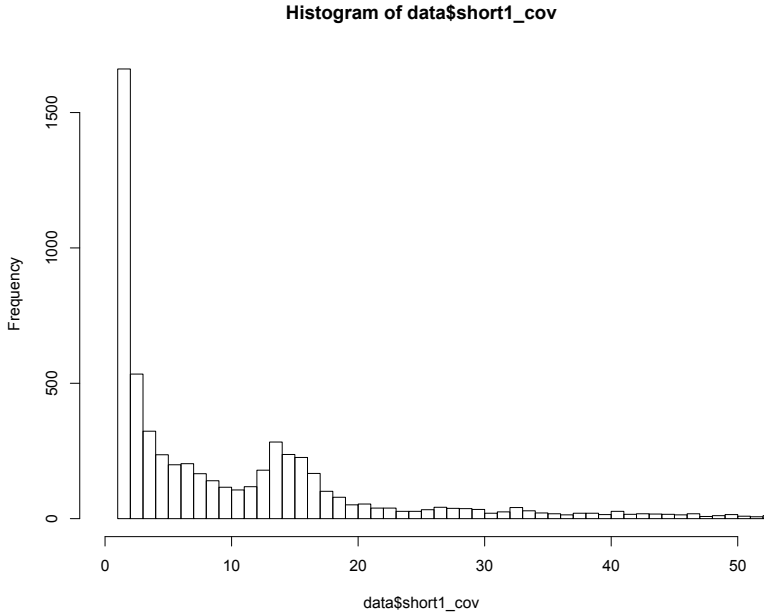
Velvet was designed to be explicitly cautious when correcting the assembly, to lose as little information as possible. This consequently will leave some obvious errors lying behind after the Tour Bus algorithm (cf. 6) was run. To detect them, you can plot out the distribution of k-mer coverages (5.1), using plotting software (I use R).

The examples below are produced using the S. suis P1/7 data available from the Sanger Institute (www.sanger.ac.uk/Projects/S_suis/) (Note: a simple script is necessary to convert the sequence files to FastA). I used a k-mer length of 21 and no cutoff.

With the R instruction:

```
(R) > data = read.table("stats.txt", header=TRUE)
(R) > hist(data$short1_cov, xlim=range(0,50), breaks=1000000)
```

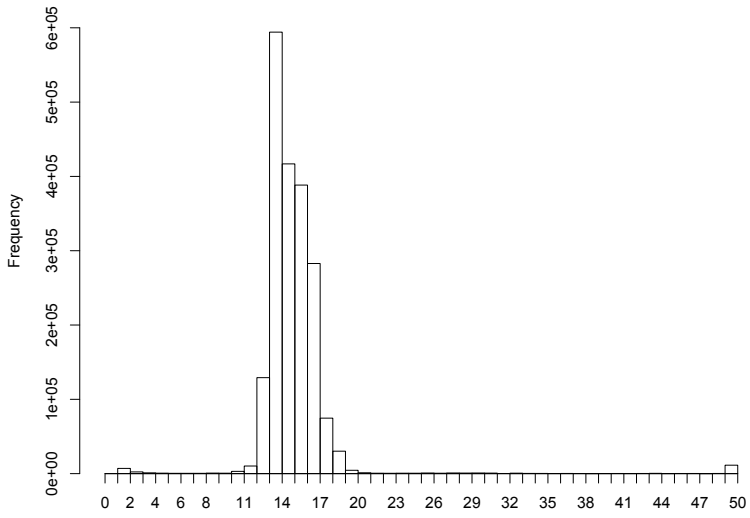
...you can obtain:



However, if you weight the results with the node lengths (you need to install the *plotrix* package for R to do this):

```
(R) > library(plotrix)
(R) > weighted.hist(data$short1_cov, data$lgth, breaks=0:50)
```

. . . you obtain:



The comparison of these two plots should convince you that below 7 or 8x you find mainly short, low coverage nodes, which are likely to be errors. Set the exact cutoff at your discretion.

However beware that there is such a thing as an over-aggressive cutoff, which could create mis-assemblies, and destroy lots of useful data.

If you have read-pair information, or long reads, it may be profitable to set a low coverage cutoff and to use the supplementary information resolve the more ambiguous cases.

5.3 Determining the expected coverage

From the previous weighted histogram it must be pretty clear that the expected coverage of contigs is near 14x.

5.4 Visualising contigs and assemblies

This section will be quite vague, as there are a number of solutions currently available, and presumably new ones under development. The following indications are just hints, as I have not done any exhaustive shopping nor benchmarking.

Most assembly viewers require an assembly format, which come in a variety of shapes and colours: ACE, AMOS, CELERA, TIGR, etc. Velvet only ouputs AMOS .afg files, but these can easily be converted with open-source software (amos.sourceforge.net).

5.5 What's long and what's short?

Velvet was pretty much designed with micro-reads (e.g. Illumina) as short and short to long reads (e.g. 454 and capillary) as long. Reference sequences can also be thrown in as long.

That being said, there is no necessary distinction between the types of reads. The only constraint is that a short read be shorter than 32kb. The real difference is the amount of data Velvet keeps on each read. Short reads are presumably too short to resolve many repeats, so only a minimal amount of information is kept. On the contrary, long reads are tracked in detail through the graph.

This means that whatever you call your reads, you should be able to obtain the same initial assembly. The differences will appear as you are trying to resolve repeats, as long reads can be followed through the graph. On the other hand, long reads cost more memory. It is therefore perfectly fine to store Sanger reads as "short" if necessary.

6 For more information

Publication: For more information on the theory behind Velvet, you can turn to:

> D.R. Zerbino and E. Birney. 2008. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, **18**: 821-829

Please use the above reference when citing Velvet.

Webpage: For general information and FAQ, you can first take a look at www.ebi.ac.uk/~zerbino/velvet.

Mailing list: For questions/requests/etc. you can subscribe to the users' mailing list: velvet-users@ebi.ac.uk.

To do so, see listserver.ebi.ac.uk/mailman/listinfo/velvet-users .

Contact emails: For specific questions/requests you can contact us at the following addresses:

- Daniel Zerbino <zerbino@ebi.ac.uk>
- Ewan Birney: <birney@ebi.ac.uk>

Reporting bugs: We are very grateful to all the people who send us bugs. However, to speed up the process and avoid useless delays, please:

1. ensure that you have the very last version of Velvet, to the last digit, as displayed on the website.

2. attach to your e-mail the Log file from within the Velvet directory.

3. if the program crashed and created a core dump file could you please:

   (a) destroy the core.* file

   (b) recompile Velvet with the instruction "make debug"

   (c) re-run Velvet and let it crash (therefore creating a new core file)

   (d) launch the GNU debugger with the instructions:

   ```
   > gdb ./velvetg core.*
   ```

   (e) within gdb, request a backtrace:

   ```
   (gdb) bt
   ```

   (f) send the listing with the entire gdb session.