Using the Velvet *de novo* Assembler for Short-Read Sequencing Technologies

Daniel R. Zerbino[1,2]

[1]Center for Biomolecular Science and Engineering, Santa Cruz, California
[2]EMBL-EBI, Wellcome Trust Genome Campus, Cambridge, United Kingdom

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. *Curr. Protoc. Bioinform.* 31:11.5.1-11.5.12. © 2010 by John Wiley & Sons, Inc.

Keywords: Genome assembly • Next-Generation Sequencing • de Bruijn Graphs

INTRODUCTION

The Velvet *de novo* assembler (Zerbino and Birney, 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 and Birney, 2008). It then tries to resolve repeats, based on the available information, whether long reads or paired-end reads (Zerbino et al., 2010). It finishes by outputting an assembly of the reads, along with various statistics (see Guidelines for 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).

ASSEMBLING A SET OF READS WITH Velvet

BASIC PROTOCOL

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. The `velveth` program 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. The `velvetg` program 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.

Current Protocols in Bioinformatics 11.5.1-11.5.12, September 2010
Published online September 2010 in Wiley Online Library (wileyonlinelibrary.com).
DOI: 10.1002/0471250953.bi1105s31

Necessary Resources

Hardware

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

Software

`velvetg` and `velveth` (see Support Protocol 1 for installation)
If analyzing colorspace reads, see Support Protocol 3 for specific comments
A Linux/Unix/MacOS X command shell is required

Files

Sequence files in FASTA (*APPENDIX 1B*), FASTQ (Cock et al., 2010), SAM, or BAM format (Li et al., 2009). The chosen format has no impact on results.

1. Prepare paired-end files by converting two FASTA (FASTQ) files into one using the `shuffleReads_fasta.pl` (`shuffleReads_fastq.pl`) script which can be found in the Velvet package:

   ```
   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.

   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.

   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 consist 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`.

   a. For a SAM file, the following is sufficient:
   ```
   sort reads.sam > reads.sorted.sam
   ```
   b. For a BAM file, use SAMTOOLS (Li et al., 2009):
   ```
   samtools sort -n reads.bam reads.sorted
   ```

2. Categorize the reads. Determine whether each file contains long or short reads, paired or unpaired reads, and finally which short paired files can be grouped together.

   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., Roche/454 or capillary Sanger reads) 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 into 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. Choose a hash length (*k*-mer length or word length).

   The hash length is probably the single most important parameter of a Velvet run. See Critical Parameters 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. Run `velveth`. See Support Protocol 1 for installation of `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] ...]
   ```

   Simply stated, you must provide, in the following 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 step 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. Run `velvetg`. See Support Protocol 1 for installation of `velvetg`. For a basic heuristic run of `velvetg`, execute the following command:

   ```
   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. Optimize the hash length. Repeat steps 4 to 5 using different hash lengths. Some simple tips are provided in the Critical Parameters section.

7. Optimize other parameters. After running step 4 once with the optimal hash length, you can repeat step 5 with different options to determine the optimal parameters for `velvetg`.

   In the following order, optimize: the coverage cutoff, the expected coverage, 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 (these data are displayed on the screen as `velvetg` *exits). See Critical Parameters 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 irrelevant.*

As an example, if you want to set the coverage cutoff at 10× and the expected coverage at 30×, 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 1

INSTALLING Velvet

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

Necessary Resources

Hardware

A system with as much physical memory as possible (12 GB or more) is recommended. Velvet can in theory function in a 32-bit environment, but such systems have memory limitations which might ultimately be a constraint for assembly. A 64-bit 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 64-bit Linux environment with *gcc*. It has been tested on other systems such as MacOS, Sparc/Solaris, and Windows (with Cygwin installed)

1. Download the latest version of Velvet at *http://www.ebi.ac.uk/~zerbino/velvet/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 that you intend to use.

   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 (see Basic Protocol and Critical Parameters), you can then recompile 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 (see 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 your `PATH` environment variable to point to the Velvet directory.

USING VelvetOptimiser

SUPPORT PROTOCOL 2

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.

Necessary Resources

Hardware

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

Software

Velvet software, installed (Support Protocol 1)
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*

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` (see Basic Protocol, step 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 50 Mbp 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.

PROCESSING COLORSPACE DATA

SUPPORT PROTOCOL 3

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. Colorspace files have very different properties in term of strandedness and error rates. This is why it is recommended to use colorspace-compatible software throughout, and only convert 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 colorspace data.

Materials

Hardware

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

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 colorspace reads with other data, use the *Corona Lite* package (see Internet Resources)

Files

A set of colorspace reads in csfasta format (see Internet Resources). In the examples below, a paired-end dataset is represented by files `reads_1.csfasta` and `reads_2.csfasta`

1. Compile the colorspace version of Velvet.

   Follow the procedure described in Support Protocol 1, except that at step 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 colorspace reads with other datasets, you should convert these other reads to colorspace. For this, the Corona Lite package contains the `encodeFasta.py` script which converts FASTA files. To convert file `reads.fa`, execute the following command:

   ```
   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*, execute the following commands:

```
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

Output 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 Advanced Parameters 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 of 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 contigs though repeated regions, whenever possible, 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 $L + k - 1$ base pairs long.

Similarly, statistics derived from lengths are also subjected to this transformation. If the mean coverage of an assembly is reported as C_k read *k*-mers per contig *k*-mer, this coverage corresponds in fact to roughly $C_k L/(L + k - 1)$ read base pairs 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.

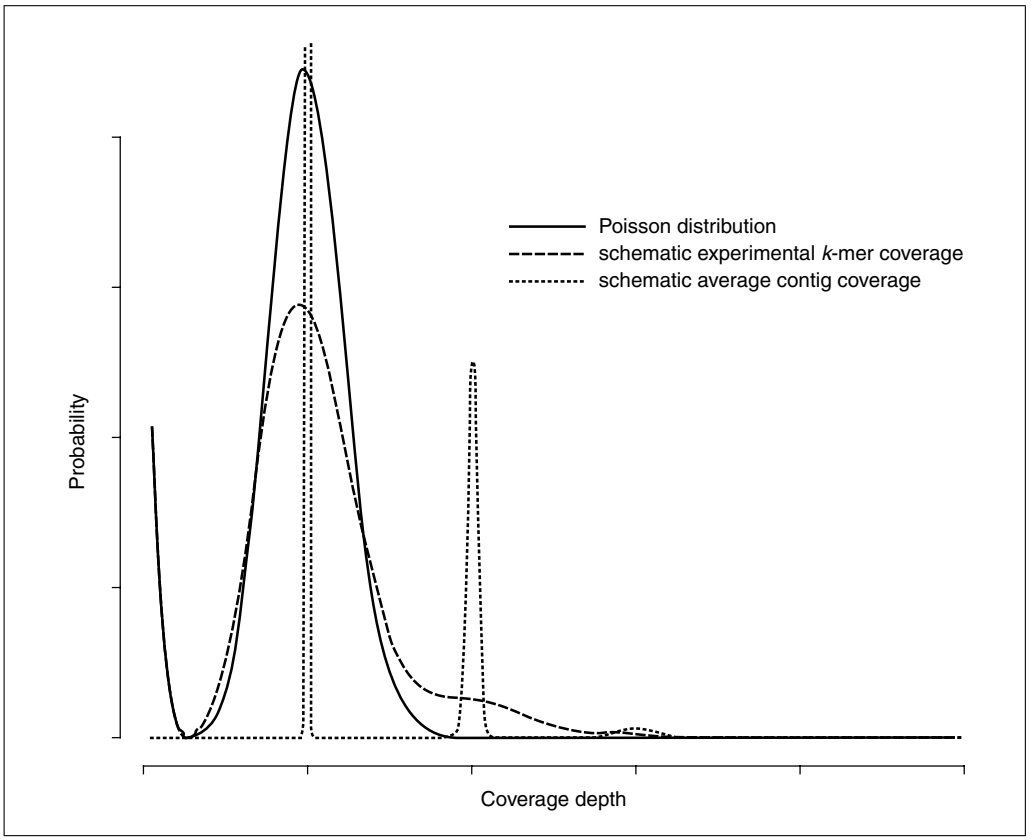

Figure 11.5.1 Schematic representation of the coverage distribution. The solid curve represents the expected *k*-mer coverage distribution in an idealized sequencing project of a repeat-free genome, i.e., a Poisson distribution. The dashed curve schematically represents the *k*-mer coverage distribution 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.

The coverage of individual unique *k*-mers should normally obey a Poisson distribution, as shown on Figure 11.5.1. 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 11.5.1. 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 that 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 is a multiple 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 that decreases proportionally to the inverse of the contigs' length. As the contigs get longer, the width of the peaks should get narrower, as shown in Figure 11.5.1.

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 in Figure 11.5.1, then the run was probably very successful and a simple coverage cutoff would presumably be very efficient in

clearing out any uncorrected errors. 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 versus 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 are *EULER* (Pevzner et al., 2001; Chaisson et al., 2009), *ALLPATHS* (Butler et al., 2008), *ABySS* (Simpson et al., 2009), *SOAPdenovo* (Li et al., 2010), 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 out false-positive alignments.

Critical Parameters

Hash lengths

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* (see 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 the 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 15×. The function that associates a contig N50 to each hash length generally has a very distinct bell shape. Thus, it is quite easy to find the optimum value with a few iterations.

It is very 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 recompile Velvet with a higher `MAXKMERLENGTH` parameter (see Support Protocol 1).

Expected coverage and coverage cutoff

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 to enter it manually on the `velvetg` command line.

The coverage cutoff is a simple but effective way of removing many basic errors that passed the previous filters. It simply removes contigs with an average coverage 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

Lack of computer memory

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 typically requires 15× *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 and the lower the memory consumption. Therefore, a proper setting of the hash length can help fit a dataset into the computer.

Missassemblies

Another common issue is missassembly. 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 values 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` threshold 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 that 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 that 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 an 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 are libraries (or categories) provided to `velveth`, changing the flag's number accordingly: `-ins_length2`, `-ins_length5_sd`, etc.

If you are using preassembled 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 precomputed 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), using the following command:

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

Literature Cited

Butler, J., MacCallum, I., Kleber, M., Shlyakhter, I.A., Belmonte, M.K., Lander, E..S., Nusbaum, C., and Jaffe, D.B. 2008. ALLPATHS: De novo assembly of whole-genome shotgun microreads. *Genome Res.* 18:810-820.

Cock, P.J., Fields, C.J., Goto, N., Heuer, M.L., and Rice, P.M. 2010. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Res.* 38:1767-1771.

Chaisson, M.J., Brinza, D., and Pevzner, P.A. 2009. De novo fragment assembly with short mate-paired reads: Does the read length matter? *Genome Res*. 19:336-346.

Li, H., Handsaker, B., Wysoker, A., Fennel, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., and the 1000 Genome Project Data Processing Subgroup. 2009. The Sequence Alignment/Map format and SAMtools. *Bioinformatics* 25:2078-2079.

Li, R., Zhu, H., and Wang, J. 2010. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res.* 20:265-272.

Kurtz, S., Narechania, A., Stein, J., and Ware D. 2008. A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics* 9:517.

Pevzner, P.A., Tang, H., and Waterman, M.S. 2001. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. U.S.A.* 98: 9748-9753.

Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J., and Birol, I. 2009. ABySS: A parallel assembler for short read sequence data. *Genome Res.* 19:1117-1123.

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

Zerbino, D.R., McEwen, G.K., Margulies, E.H., and Birney, E. 2010. Pebble and rock band: Heuristic resolution of repeats and scaffolding in the velvet short-read *de novo* assembler. *PLoS ONE* 4:e8407.

Key References

Zerbino and Birney, 2008. See above.

This first publication mainly described the implementation of de Bruijn graphs within Velvet and the error-correction algorithm, TourBus.

Zerbino et al., 2010. See above.

This follow-up paper describes how Velvet resolves complex repeats using long reads or paired-end read information.

Internet Resources

http://www.ebi.ac.uk/∼zerbino/velvet

Velvet Web site, where code and information on Velvet can be downloaded.

http://bioinformatics.net.au/software.shtml

VelvetOptimiser by Simon Gladman and Torsten Seeman. This wrapper software scans different parameters of Velvet to produce an optimal assembly, as described in Support Protocol 2.

http://solidsoftwaretools.com/gf/project/denovotools/

Colorspace de novo pipeline by Craig Cummings, Vrunda Sheth, and Dima Brinza. These scripts allow you to do all the appropriate colorspace conversions described in Support Protocol 3 (a registration is required, but the software is free).

http://solidsoftwaretools.com/gf/project/corona/
The Corona Lite package can be found on this server.

http://sourceforge.net/apps/mediawiki/amos/
AMOS suite by the AMOS Consortium. This suite of tools allows the user to manipulate, convert or analyze AFG assembly files.

http://tools.invitrogen.com/content/sfs/manuals/SOLiD_SAGE_SoftwareGuide.pdf
Colorspace documentation by Applied Biosystems. This document describes colorspace, and the csfasta format in particular.