goss(1) Gossamer User Manual

Bryan Beresford-Smith, Andrew Bromage,
Thomas Conway, Jeremy Wazny, Justin Zobel

September 26, 2011

NAME

goss - a tool for *de novo* assembly of high throughput sequencing data.

Version 1.1.0

SYNOPSIS

goss build-graph -k 27 -i in.fastq -O graph

goss merge-graphs -G graph1 -G graph2 -O graph-merged

goss trim-graph -G graph -O graph-trimmed

goss prune-tips -G graph -O graph-pruned

goss print-contigs —min-coverage 10 —min-length 100 -G graph-pruned > paths.fa

goss pop-bubbles -G graph-pruned -O graph-popped

goss build-entry-edge-set -G graph-popped

goss build-supergraph -C 10 -G graph-popped

goss thread-pairs -G graph-popped —expected-coverage 70 —insert-expected-length 300

goss thread-reads -G graph-popped —expected-coverage 70

DESCRIPTION

Gossamer is an application for doing *de novo* assembly of high throughput sequencing data. It is a memory efficient assembler based on the *de Bruijn* graph.

The advantage of Gossamer is that large data sets can be assembled on computers with small amounts of memory. The fundamental parameter to *de Bruijn* graph based methods is k, the size of substrings used in the construction of the graph. These substrings are referred to as k-mers and correspond to nodes in the graph. Edges in the graph correspond to (k+1)-mers which are called rho-mers. It should

be noted that larger values of k will require larger memory sizes because the size of the rho-mer space is larger. Another point to note is that by using a small cluster of small memory computers the time can be easily and substantially reduced by building subsections of the graph in parallel.

Input files are base-space reads in FASTA or FASTQ format or in a format with one read per line and in either plain text or compressed format (i.e. gzip).

Using Gossamer

Gossamer runs in several phases and each phase will produce a new graph or additional information for an existing graph. Graphs are represented as a collection of files with a common prefix which may include a filesystem path. For example, the files composing a graph object with prefix "graph" are:

```
graph-counts-hist.txt
graph-counts.ord0
graph-counts.ord1
graph-counts.ord1p.classes
graph-counts.ord1p.class-sum
graph-counts.ord1p.header
graph-counts.ord1p.offsets
graph-counts.ord1p.offset-sum
graph-counts.ord2
graph-counts.ord2p.classes
graph-counts.ord2p.class-sum
graph-counts.ord2p.header
graph-counts.ord2p.offsets
graph-counts.ord2p.offset-sum
graph-edges-d0
graph-edges-d1
graph-edges.header
graph-edges.high-bits
graph-edges.low-bits.lwr
graph-edges.low-bits.upr
graph.header
```

It is currently up to the user to organize the names of graphs in a meaningful way. See the example of the common phases of *goss* given below.

Any graph can be processed to find the contigs in the graph and to output those contigs to a FASTA file (or stdout). Various statistics for each contig are included in the contig description line, including the identifier of the contig, its length and

information about the rho-mer coverage. The rho-mer coverage is the number of times the rho-mer occurs in the set of reads and for each printed contig the minimum, maximum, mean and standard deviation of the coverage in the contig is given.

One of the files representing the graph is a text file called *<prefix>-counts-hist.txt*, each line of which consists of a pair (x, y) where y is the number of different rho-mers which occur exactly x times. This information is useful for determining the minimum multiplicity to use for trimming the graph (see below).

As an example of using gossamer, suppose the sequenced reads are in four files, three of which are FASTQ files (one of which is compressed) and one of which is a FASTA file: in1.fastq, in2.fastq.gz, in3.fastq and in4.fasta.

The common phases of gossamer are:

- Build the complete *de Bruijn* graph for some k-mer size using all of the read data
  - If the data consists of several files e.g. from different lanes of sequencing then it is recommended that *goss build-graph* be run on each file separately and the resulting graphs merged using *goss merge-graphs*.

  goss build-graph -k 27 -i in1.fastq -i in2.fastq.gz -O graph1

  goss build-graph -k 27 -i in3.fastq -I in4.fasta -O graph2

  goss merge-graphs -G graph1 -G graph2 -O graph12-merged

- Trim all rho-mers with low frequency from the complete graph.
  - Rho-mers with low frequency arise overwhelmingly from sequencing errors, at least if there is good genome coverage. By default, Gossamer will automatically choose an appropriate cut-off frequency which eliminates as many of these incorrect rho-mers while keeping as it can while keeping as many correct rho-mers as possible.
  - Under some circumstances, for example if the coverage is low, the user may need to supply an appropriate cut-off frequency using the -C parameter. To pick an appropriate cut-off frequency, it may be necessary to inspect the histogram of rho-mer counts in the graph which has been built from all of the reads. For this example, the file containing this histogram would be called graph12-merged-counts-hist.txt.

  goss trim-graph -G graph12-merged -O graph12-merged-trimmed

- Prune the tips of the trimmed graph using *goss* prune-tips, to clean up additional edges resulting from sequencing errors.

  goss prune-tips -G graph12-merged-trimmed -O graph12-merged-trimmed-pruned1

- Repeat the pruning since one step of pruning can create further tips which can be removed. This can be done until sufficiently small numbers of new tips are discovered, or even until no more tips are discovered.

  goss prune-tips -G graph12-merged-trimmed-pruned1 -O graph12-merged-trimmed-pruned2

- Output the contigs to a FASTA file.

  – At this stage of the assembly, all of the read data has been incorporated into a *de Bruijn* graph and that graph has been substantially cleaned of features arising from read errors. It is now possible to print all of the unbranched paths in the graph even though further steps in the assembly process will most likely substantially improve the overall lengths of the contigs. In fact, at any point in the graph building, e.g. after pruning, these contigs can be printed.

  goss print-contigs —min-coverage 10 —min-length 100 -G graph12-merged-trimmed-pruned2 -o paths.fa

- Pop the "bubbles" in the graph.

  – A "bubble" is a graph structure with two similar, alternative paths between a start and an end for which the paths are not too long. These paths may arise, for example, because of the heterozygous nature of a diploid genome, sequencing errors, or by variants such as SNPs or cloning artifacts. It can be beneficial during the assembly process to remove one of the two alternative paths. The pop-bubbles command detects such "bubbles" and removes them from the graph.

  goss pop-bubbles -G graph12-merged-trimmed-pruned2 -O graph12-merged-trimmed-pruned2-popped

- Build the graph edge entry sets.

  – After the initial filtering steps described above, pairs of reads can be used to extend the contigs by resolving repeat structures. A new graph which we call a supergraph is constructed and manipulated during this process. The supergraph structure, which is not itself a *de Bruijn* graph, is stored beside the original *de Bruijn* graph, which remains unmodified.

  There are three stages to this process: 1. Build the entry edge sets. 2. Build a supergraph from the established linear segments. 3. Resolve paths corresponding to pairs of reads.

  As a first stage, a set of entry edge sets is constructed for the graph. These identify the beginnings of (unbranching) linear segments in the *de Bruijn* graph.

goss build-entry-edge-sets -G graph12-merged-trimmed-pruned2-popped

- Build the initial supergraph.
  - Once the entry edge sets have been constructed the information can be used to build the *de Bruijn* graph's supergraph. Note that no explicit name is provided for the new graph: in later commands, the supergraph is identified by reference to the *de Bruijn* graph from which it was generated.

  goss build-supergraph -G graph12-merged-trimmed-pruned2-popped

- Use read pairs to resolve repeat structures.
  - The initial supergraph represents all of the unbranching segments of the *de Bruijn* graph. Paired read information can be used to establish which of the many possible paths through the graph correspond to true sequences of the original genome. Gossamer requires read pairs to be supplied as a pair of files; the nth reads of the first and the second input files make up the nth read pair. Both paired-end and mate-pair formats are supported. Additionally, the estimated genome coverage of the original *de Bruijn* graph, and the expected pair insert size must be provided. The result of "threading" pairs is a supergraph which is updated in-place. The pair threading step may be run multiple times, with different inputs and metadata, resulting in a more refined supergraph after each step. Note that, at any stage the supergraph can be restored to its initial state by re-running the build-supergraph command.

  goss thread-pairs -G graph12-merged-trimmed-pruned2 -i in1.fastq -i in2.fastq —insert-expected-size 200 —expected-coverage 70

- Use individual reads to resolve remaining small-scale repeat structures
  - Just as read pairs are used to join distant contigs, individual reads may be used to connect contigs which are spanned by individual reads. The read threading step operates on the same supergraph, updating it in place, in the same way as pair threading. It also requires an expected coverage estimate.

  goss thread-reads -G graph12-merged-trimmed-pruned2 -i in1.fastq -i in2.fastq —expected-coverage 70

- Print the supergraph contigs.
  - Once the supergraph has been built, including after any pair or read threading step, the contigs can be printed in FASTA format, along with information about each contig.

  goss print-contigs —min-length 100 -G graph12-merged-trimmed-pruned2-popped -o supergraphContigs.fa

OPTIONS COMMON TO ALL COMMANDS

The following options can be used with all of the *goss* commands and are therefore not listed separately for each command.

-h, –help Show a help message.

-l *FILE*, –log-file *FILE* Place to write progress messages. Messages are only written if the -v flag is used. If omitted, messages are written to stderr.

-T *INT*, –num-threads *INT* The maximum number of *worker* threads to use. The actual number of threads used during the algorithms depends on each implementation. *goss* may use a small number of additional threads for performing non cpu-bound operations, such as file I/O.

–tmp-dir *DIRECTORY* A directory to use for temporary files. This flag may be repeated in order to nominate multiple temporary directories.

-v, –verbose Show progress messages.

-V, –version Show the software version.

-D *arg*, –debug *arg* Enable particular debugging output.

COMMANDS AND OPTIONS

goss build-graph

goss build-graph [-B *INT*] [-S *INT*] -k *INT* {-I *FASTA-filename* | -i *FASTQ-filename* | —line-in *filename*}+ -O *PREFIX*

Build the *de Bruijn* graph from the reads contained in the given FASTA and FASTQ files and output the resulting graph object as a set of files with the given *PREFIX*. Both FASTA and FASTQ input files are supported, with options *-I* and *-i* respectively.
In addition, files with one read per line are also supported. The input files can be compressed with the compression method implied by the file suffix as follows:

- *.gz* Compressed using gzip.

For large projects (such as sequencing the human genome with high coverage) several separate *goss build-graph* commands may be done on different files in parallel (e.g. on files from different lanes of an Illumina GAII), followed by a sequence of

merging steps using *goss merge-graphs*. Even in the absence of a cluster, building parts and merging is usually faster than building a large graph with all the input files in one go. Future releases may do this automatically.

OPTIONS

-S *INT*, –log-hash-slots *INT* S is the log (base 2) of the number of slots in the hash table used during build-graph (default is 24). The memory used by the hash table is H=2^(S+4) Gigabytes. For example, S=26 implies H=1 Gigabyte, S=30 implies H=16 Gigabytes, S=35 implies H=512 Gigabytes.

Build-graph will actually use slightly more RAM than H so, for example, on a machine with a total of 32 Gigabytes of RAM the maximum value of S which can be used is S=30.

Specifying small S allows build-graph to be run on machines with a small amount of memory. However, for large data sets larger values for *S* will improve performance.
The optimal value for the buffer-size is related to a number of factors, including the final graph size.

-B *INT*, –buffer-size *INT* Maximum buffer-size for in-memory buffers is *INT* Gigabytes (defults to 2). This is a convenient way of setting the maximum amount of RAM which will be used by build-graph and
allows build-graph to be run on machines with a small amount of memory.
For large data sets, however, larger values for *B* will improve performance. For example, for a data set with a large number (e.g. 70 million) of reads of length 100, a typical value of B would be 24 (i.e. use 24 Gb buffers). The actual optimal value for the buffer-size is related to the final graph size. If this B parameter is used then a value of S (see previous parameter) is calculated. A typical value for B would be the amount of machine RAM, in Gigabytes. The calculated value of S (see below) would be floor(log2(RAM/16)).

-I *FILE*, –fasta-in *FILE* Input file in FASTA format.

-i *FILE*, –fastq-in *FILE* Input file in FASTQ format.

—**line-in *FILE*** Input file with one read per line and no other annotation.

-O *PREFIX*, –graph-out *PREFIX* Use *PREFIX* as the prefix name of the output graph object. The *PREFIX* must be a valid file name prefix.

-k *INT*, –kmer-size *INT* The k-mer size to use for building the graph: in version 0.3.0 this *must be an integer strictly less than 63.*

goss help

goss help

Prints a summary of all of the gossamer commands.

goss lint-graph

goss lint-graph {-G | —graph-in} *PREFIX*

Report any inconsistencies in the graph. This includes a check of the symmetry of paths and their reverse complements in the graph.

OPTIONS

-G *PREFIX*, –graph-in *PREFIX* The name of the graph object. This is the string used as the prefix for the names of the files making up a graph object.

goss merge-graphs

goss merge-graphs {-G *PREFIX*}+ -O *PREFIX*

Create a new graph by merging one or more other graphs.

Merging uses a small, constant amount of memory per graph; very likey less than 1 GB of memory in total.

OPTIONS

-G *PREFIX*, –graph-in *PREFIX* The name of the graph object. This is the string used as the prefix for the names of the files making up a graph object. When merging several graphs this parameter is repeated for each graph, as demonstrated in the example above.

-O *PREFIX*, –graph-out *PREFIX* Use *PREFIX* as the prefix name of the output graph object. The *PREFIX* must be a valid file name prefix.

goss print-contigs

goss print-contigs -G *PREFIX* [—min-coverage *INT*] [—min-length *INT*] [-o *FILE*] [—no-sequence]

Print all of the non-branching paths in a given graph. By default, contigs will be printed from the supergraph if it is present, otherwise the *de Bruijn* graph will be used. The paths are printed in FASTA format with the composition of the

descriptor line for each contig depending on whether the underlying graph is a *de Bruijn* graph or a supergraph. In the case of a *de Bruijn* graph, the descriptor line contains the following information (separated by colons ':'):

- Path Number
- Path Length
- Minimum Path Coverage
- Maximum Path Coverage
- Average Path Coverage
- Standard Deviation of Path Coverage

The coverage of a rho-mer is defined to be the number of times it occurs in all reads. The minimum path coverage is the minimum coverage of all of the rho-mers in the path.

For contigs generated from a supergraph, the descriptor line is made up of the following fields (separated by commas ','):

- Path Number
- Path Length
- List of Segment Lengths (separated by colons ':')
- Minimum Path Coverage
- Maximum Path Coverage
- Average Path Coverage
- Standard Deviation of Path Coverage

OPTIONS

-G *PREFIX*, –graph-in *PREFIX* The name of the graph object. This is the string used as the prefix for the names of the files making up a graph object.

—**min-coverage *INT*** Only print those paths which have a minimum rho-mer coverage >= C. This flag is only used when printing contigs from a *de Bruijn* graph. Defaults to 0.

–**min-length *INT*** Only print those paths which have a length >= the specified minimum length. Defaults to 0.

-o *FILE*, –output-file *FILE* The name of the FASTA output file for the printed paths. Use '-' to output to standard output. The *FILE* must be a valid file name. Defaults to standard output.

–**no-sequence** Suppresses the printing of the actual contig sequences. Instead, only the information present in the descriptor lines will be printed in a tab separated format. The default is to print sequences.

–**print-linear-segments** Only print linear segments, ignoring the supergraph if it is present. The default is to use the supergraph if available.

goss prune-tips

Create a new graph by pruning the tips in the input graph. Pruning tips is the process of removing short dead-ends in the graph. It is probable that such graph features arise from errors in the reads.

goss prune-tips -G *PREFIX* -O *PREFIX*

OPTIONS

-G *PREFIX*, –graph-in *PREFIX* The name of the input graph. This is the string used as the prefix for the names of the files making up a graph object.

-O *PREFIX*, –graph-out *PREFIX* The name of the output graph. Use *PREFIX* as the prefix name of the output graph object. The *PREFIX* must be a valid file name prefix.

goss trim-graph

goss trim-graph [-C *INT*] -G *PREFIX* -O *PREFIX*

Create a new graph by trimming the input graph. Trimming is the process of removing all edges in the graph which occur less than times. This is used to remove those edges which most likely arose from errors.

OPTIONS

-G *PREFIX*, –graph-in *PREFIX* The name of the input graph. This is the string used as the prefix for the names of the files making up a graph object.

-O *PREFIX*, –graph-out *PREFIX* The name of the output graph. Use *PREFIX* as the prefix name of the output graph object. The *PREFIX* must be a valid file name prefix.

-C *INT*, –cutoff *INT* Edges with coverage at or below this value are removed.

goss pop-bubbles

goss pop-bubbles [—max-edit-distance *INT*] [—max-=error-rate *FLOAT*] [—max-sequence-length *INT*] -G *PREFIX* -O *PREFIX*

A "bubble" is a graph structure with two alternative paths between a start and an end vertex for which the paths are similar but not too long. These paths may arise, for example, because of the heterozygous nature of a diploid genome, sequencing errors, or by variants such as SNPs or cloning artifacts. It can be beneficial during the assembly process to remove one of the two alternative paths. pop-bubbles detects "bubbles" and removes them from the graph.

OPTIONS

-G *PREFIX*, –graph-in *PREFIX* The name of the input graph. This is the string used as the prefix for the names of the files making up a graph object.

-O *PREFIX*, –graph-out *PREFIX* The name of the output graph. Use *PREFIX* as the prefix name of the output graph object. The *PREFIX* must be a valid file name prefix.

–max-edit-distance *INT* The maximum edit distance to qualify as a bubble.

–max-error-rate *FLOAT* The maximum error rate to qualify as a bubble.

–max-sequence-length *INT* the maximum length of a sequence to consider.

goss build-entry-edge-set

goss build-entry-edge-sets -G *PREFIX*

Build the graph entry edge sets. The entries are represented by files with prefix *PREFIX*-entries.

OPTIONS

-G *PREFIX*, –graph-in *PREFIX* The name of the input graph. This is the string used as the prefix for the names of the files making up a graph object.

goss build-supergraph

goss build-supergraph -G *PREFIX*

Build the supergraph. The graph with the specified prefix is augmented with further files representing the supergraph.

OPTIONS

-G *PREFIX*, –graph-in *PREFIX* The name of the input graph. This is the string used as the prefix for the names of the files making up a graph object.

goss thread-pairs

goss thread-pairs -G *PREFIX* {-I *FASTA-filename* | -i *FASTQ-filename* | —line-in *filename*}+ —expected-coverage *INT* —insert-expected-size *INT* [—insert-size-std-dev *FLOAT*] [—insert-size-tolerance *FLOAT*] [—min-link-count *INT*] [—search-radius *INT*] [—edge-cache-rate *INT*] [—paired-ends] [—mate-pairs]

This command locates pairs of reads on the supergraph and uses them to join sequences of branching segments into new, longer contigs. The first step is to identify pairs of contigs related by read pairs. *goss* finds and stores the supergraph location of both reads in each pair. We call each pair of reads which connects a pair of contigs, as well the position of those reads within the contigs, a link. Using the position information within a link, the distance between linked contigs is estimated. Pairs can be supplied in either paired-end or mate-pairs format.

Once all read pairs have been anchored to the graph, any contig pairs which do not have at least a minimal number of links are removed from consideration. For the remaining pairs, *goss* will attempt to find paths in the supergraph which join both sides of the pair. Only paths with lengths near the insert size are considered. The range of valid path lengths is specified by providing: the expected insert size; the size of a standard deviation, as a percentage of the insert size; and a tolerance value, which indicates the number of standard deviations in range. For example, specifying an insert size of 200 bases, a standard deviation of 10% and a tolerance of 2.0 standard deviations, means that only paths of length in the range [160,240] bases are considered. By default, *goss* will attempt to find all paths, between linked contigs, within the distance bounds. When a unique qualifying path is found, it is used to join the two sides of the corresponding contig pair, yielding a new, longer contig in their place. To improve performance, the area of the graph which is searched can be limited to within some distance of the target contig.

The result of running thread-pairs is a new supergraph, which is written over the top of the original. This command can be repeated for multiple sets of pairs, each of which will modify the same supergraph in-place.

OPTIONS

-G *PREFIX*, –graph-in *PREFIX* The name of the input graph. This is the string used as the prefix for the names of the files making up a graph object.

-I *FILE*, –fasta-in *FILE* Input file in FASTA format.

-i *FILE*, –fastq-in *FILE* Input file in FASTQ format.

—**line-in *FILE*** Input file with one read per line and no other annotation.

—**expected-coverate *INT*** The expected genome coverage of the reads used to build the underlying *de Bruijn* graph. These may or may not be the same as the read pairs used in this stage. This flag is mandatory.

—**insert-expected-size *INT*** The insert size for the given pairs. i.e. the distance, in bases, between extreme ends of the pairs when mapped to the genome. This value must be supplied.

—**insert-size-std-dev *FLOAT*** The standard deviation for allowed insert sizes, as a percentage. The default is 10%.

—**insert-size-tolerance *FLOAT*** The range of valid insert sizes, in standard deviations. This defaults to 2.0.

—**min-link-coverage *INT*** Disregard pairs of contigs which are linked by fewer than this number of read pairs. The default value is 10.

—**search-radius *INT*** When searching for a path from one contig to another, limit the search to within this number of linear segments from the target. A radius value of 0 means that the search is not constrained to any region of the graph. Restricting path finding to an area of the graph is likely to decrease search times. There may be no noticeable benefit for relatively small graphs, however. The default value is 10.

—**edge-cache-rate *INT*** To efficiently locate read pairs within the supergraph, *goss* builds a data structure for caching the positions of *de Bruijn* graph edge within the supergraph edges that contain them. In order to save memory, this information is stored for only a proportion of the complete set of *de Bruijn* graph edges. For a value of n, position information is stored for one edge out of every 2^n, where each entry takes 16 bytes. High values will reduce memory usage, at the expense of an increased runtime, while lower values will lower the runtime, but require more memory. The default is 4, which implies that the edge cache requires one byte of memory for each edge.

—**paired-ends** The input read pairs are in paired-end format. This is the default.

—**mate-pairs** The read pairs are in mate-pair format.

goss thread-reads

goss thread-reads -G *PREFIX* {-I *FASTA-filename* | -i *FASTQ-filename* | —line-in *filename*}+ —expected-coverage *INT* [—min-link-count *INT*] [—edge-cache-rate *INT*]

The thread-reads command takes each individual read from the given input, and identifies the sequence of superpath segments that read runs through. Any pair of segments which is unambiguously linked by a sufficient number of reads, is then joined into a new, longer segment. Two segments are considered to be linked unambiguously if every read going through the first also runs through the second.

OPTIONS

-G *PREFIX*, –graph-in *PREFIX* The name of the input graph. This is the string used as the prefix for the names of the files making up a graph object.

-I *FILE*, –fasta-in *FILE* Input file in FASTA format.

-i *FILE*, –fastq-in *FILE* Input file in FASTQ format.

—**line-in *FILE*** Input file with one read per line and no other annotation.

—**expected-coverate *INT*** The expected genome coverage of the reads used to build the underlying *de Bruijn* graph. These may or may not be the same as the read pairs used in this stage. This flag is mandatory.

—**min-link-coverage *INT*** Disregard pairs of contigs which are linked by fewer than this number of reads. The default value is 10.

—**edge-cache-rate *INT*** To efficiently locate reads within the supergraph, *goss* builds a data structure for caching the positions of *de Bruijn* graph edge within the supergraph edges that contain them. In order to save memory, this information is stored for only a proportion of the complete set of *de Bruijn* graph edges. For a value of n, position information is stored for one edge out of every 2^n, where each entry takes 16 bytes. High values will reduce memory usage, at the expense of an increased runtime, while lower values will lower the runtime, but require more memory. The default is 4, which implies that the edge cache requires one byte of memory for each edge.

—

LIMITATIONS

Version 1.1.0 is a prototype assembler. Only the steps described above are supported. It has been tested on very large data sets of Illumina reads for human genome assembly.

In summary, this version can be used to:

- Build the *de Bruijn* graph from the specified input reads for k-mer size k < 63. Merging graphs is supported, allowing parallel tree-based construction of graphs from sets of reads.
- Trim low frequency rho-mers from the graph.
- Prune the tips (short dead-ends) in the graph.
- Pop-bubbles in the graph.
- Resolve some repeat structures by threading pairs and individual reads through the graph and constructing a "supergraph".
- Printing contigs from either the *de Bruijn* graph or from the supergraph.

The following limitations apply:

- SOLiD reads in colour space are not supported in this release.
- Bzip-compressed input is not supported in this release due to a bug in an external library.

FUTURE RELEASES

Bzip support will be re-introduced.

A future release will also support colour space reads.

Publication

Please reference the original gossamer paper:

Thomas C Conway, Andrew J Bromage, "Succinct data structures for assembling large genomes", Bioinformatics, 2011 vol. 27 (4) pp. 479–86 **GossamerPaper**