ALLPATHS-LG Manual

Computational Research and Development Group

Genome Sequencing and Analysis Program

Broad Institute of MIT and Harvard

Cambridge, MA

Manual Revision: (27-Jan-13 2:47:00 PM)

Table of Contents

Conventions

The following conventions are used in this manual.

Commands, filenames, directories and arguments are typeset in `Courier`.

Command-line arguments are normally split one per line for clarity, listed below the actual command. For example:

```
RunAllPathsLG PRE=/assemblies DATA=datadir RUN=rundir SUBDIR=attempt1
```

becomes

```
RunAllPathsLG
PRE=/assemblies
DATA=datadir
RUN=rundir
SUBDIR=attempt1
```

User-supplied values are indicated by `<description>`. In the example below, the user should provide a value for the target name.

```
TARGETS=<target name>
```

For example:

```
TARGETS=import
```

Introduction

ALLPATHS-LG is a whole-genome shotgun assembler that can generate high-quality genome assemblies using short reads (~100bp) such as those produced by the new generation of sequencers. The significant difference between ALLPATHS and traditional assemblers such as Arachne is that ALLPATHS assemblies are not necessarily linear, but instead are presented in the form of a graph. This graph representation retains ambiguities, such as those arising from polymorphism, uncorrected read errors, and unresolved repeats, thereby providing information that has been absent from previous genome assemblies.

Capabilities and limitations

ALLPATHS-LG is a short-read assembler. It has been designed to use reads produced by new sequencing technology machines such as the Illumina Genome Analyzer. The version described here has been optimized for, but not necessarily limited to, reads of length 100 bases.

ALLPATHS is not designed to assemble Sanger or 454 FLX reads, or a mix of these with short reads.

ALLPATHS-LG requires high sequence coverage of the genome in order to compensate for the shortness of the reads. The precise coverage required depends on the length and quality of the paired reads, but typically is of the order 100x or above. This is raw read coverage, before any error correction or filtering. For small bacterial-sized genomes, this translates to a fraction of an Illumina lane – the minimum the machine is capable of without multiplexing. For larger genomes this translates into roughly one Illumina HiSeq flowcell.

ALLPATHS-LG requires a minimum of 2 paired-end libraries – one short and one long. The short library average separation size must be slightly less than twice the read size, such that the reads from a pair will likely overlap – for example, for 100 base reads the insert size should be 180 bases. The distribution of sizes should be as small as possible, with a standard deviation of less than 20%. The long library insert size should be approximately 3000 bases long and can have a larger size distribution. Additional optional longer insert libraries can be used to help disambiguate larger repeat structures and may be generated at lower coverage.

The libraries must be 'pure', that is, they must consist of reads that do not contain any non-genomic portions from stuffers or similar constructions. Reads from jumping libraries may be chimeric, that is, they may cross the junction point between the two ends of the insert that occurs in libraries produced using the Illumina sheared library protocol.

Staying up to date with our blog

The best source of current news and information on ALLPATHS-LG is our blog:

http://www.broadinstitute.org/software/allpaths-lg/blog/

Here you will find announcements, an FAQ, links to the latest code, manual and test data, build requirements and instructions, and information on how to get help from the developers of ALLPATHS-LG

We recommend that our blog page should be your starting point whenever you have problems, questions or are just looking for the latest version.

Requirements

To compile and run ALLPATHS-LG you will need a Linux/UNIX system with at least 16 GB of RAM. We suggest a minimum of 32 Gb for small genomes, and 512 Gb for mammalian sized genomes.

For the up to date list of requirements please see our General Build help here:

https://www.broadinstitute.org/science/programs/genome-biology/computational-rd/general-instructions-building-our-software

You will need:

The g++ compiler from GCC, version 4.7.0 or higher.
http://gcc.gnu.org/

The GMP library compiled with the C++ interface. Your GCC installation may already include GMP.
http://gmplib.org /

The Picard set of Java-based command-line utilities for SAM file manipulation available at
http://picard.sourceforge.net/

The graph command `dot` from the `graphviz` package.
http://www.graphviz.org/

Availability

The ALLPATHS source code is available for download via our blog at:

http://www.broadinstitute.org/software/allpaths-lg/blog/

We do not issue official releases. Instead, please download the latest version from our nightly builds. Only builds that pass our internal tests are made available in this way - we do not release broken builds.

Getting Help

Please consult the FAQ available on our blog at:

http://www.broadinstitute.org/software/allpaths-lg/blog/

Installation

For the up to date build instructions please see our General Build Instructions here:

https://www.broadinstitute.org/science/programs/genome-biology/computational-rd/general-instructions-building-our-software

After you have downloaded the latest build, unpack it using `tar`. Then you can simply compile the source code with `configure` and `make`. All of the source code will be in its own directory called `allpathslg-<revision>`; we will refer to this as the `AllPaths` directory. For example, starting from the root directory (the location of the downloaded file):

```
% tar xzf allpathslg-<revision>.tar     // expand tarball

% cd allpathslg-<revision>.tar          // move into the source directory

% ./configure --prefix=/path/to/install/directory       // run configure

% make                                  // build ALLPATHS-LG
```

```
% make install                                    // install ALLPATHS-LG
```

Troubleshooting

Of the above steps, the one most likely to fail is `configure`, which checks for the existence of various commands and libraries in your environment. You may need to change your PATH or your `LD_LIBRARY_PATH`. You may also need to run `configure` with flags. For a listing of all such available flags, run `configure --help`.

Environment

After compilation, the executable binary files will be in the subdirectory `bin` of the `allpathslg-<revision>` directory. You may want to add this directory to your PATH so that you can call the ALLPATHS binaries from anywhere. Also modify your PATH to include the directories containing `addr2line` and your chosen version of g++. You may need to change your `LD_LIBRARY_PATH` as well.

ALLPATHS pipeline overview

ALLPATHS consists of a series of modules. Each module performs a step of the assembly process. Different modules may be run, and in varying order, depending on the assembly parameters. A single module called `RunAllPathsLG` controls the entire pipeline, deciding which modules to run and how to run them. Although it is possible to run the individual modules manually, you should be able to accomplish everything you need through `RunAllPathsLG`.

RunAllPathsLG module

`RunAllPathsLG` uses the Unix `make` utility to control the assembly pipeline. It does not call each module itself, but instead creates a special `makefile` that does. Within `RunAllPathsLG` each module is defined in terms of its source and target files, and the command line used to call it. A module is only run if its target files don't exist, or are out of date compared to its source files, or if the command used to call the module has changed. In this way `RunAllPathsLG` can be run again and again, with different parameters, and only those modules that need to be called will be. This is efficient and ensures that all intermediate files are always correct, regardless of how many times `RunAllPathsLG` has been called on a particular set of source data and how many times a module fails or aborts partway through.

ALLPATHS pipeline directory structure

The assembly pipeline uses the following directory structure to store its inputs, intermediates, and outputs. The pipeline automatically creates the directories (if they don't already exist) and populates them. The names shown here are commonly used to refer to the directories, although command-line arguments determine the actual directory names.

```
REFERENCE/DATA/RUN/ASSEMBLIES/SUBDIR
```

The meaning of each directory is given below. The data separation described is the ideal and occasionally this is broken for convenience. Some files are duplicated between directories, but only in the downward direction. All files within this directory structure are under the control of the pipeline.

The location of the pipeline directory structure is specified with the RunAllPathsLG command-line argument PRE.

Typically in the directory PRE there will be a number of REFERENCE directories, one for each organism being assembled by ALLPATHS.

REFERENCE (organism) directory

The REFERENCE directory is so called because there should be one for each reference genome you use. It is used to separate assembly projects by organism and possibly also by isolate (if, for example, you want to use two different *E.coli* references) and is typically named after the organism. All assembly projects for a given organism/isolate will be contained in that REFERENCE directory. All intermediate files generated for use in evaluation that are independent of the particular assembly attempt will be stored here and shared by all assemblies.

You do not need to supply a reference genome – ALLPATHS is, after all, a *de novo* assembler. But even in *de novo* assemblies, the pipeline can perform useful evaluations at various stages of the assembly process, so you should provide a reference genome if you have one (see "Import reference" below for info on how to set up this file.) If you do not have a reference genome, simply create a single REFERENCE directory for the organism you wish to assemble.

The REFERENCE directory may contain many DATA directories, each representing a particular set of read data to assemble.

RunAllPathsLG argument: REFERENCE_NAME

DATA (project) directory

The DATA directory contains the original read data used in a particular assembly attempt. (This data is stored in internal ALLPATHS formats: fastb, qualb, pairs.) It also contains intermediate files derived from the original data that are independent of the particular assembly attempt – typically files used in evaluation.

Each DATA directory may contain many RUN directories, each representing a particular attempt to assemble the original data using a different set of parameters.

RunAllPathsLG argument: DATA_SUBDIR

RUN (assembly pre-processing) directory

The RUN directory contains all the non-localized assembly files, that is, those intermediate files generated from the original read data in preparation for the final assembly stage (LocalizeReadsLG and beyond). It may also contain intermediate files used in evaluation that are dependent on the assembly parameters chosen.

RunAllPathsLG argument: RUN

ASSEMBLIES directory

The ASSEMBLIES directory contains the actual assembly (or assemblies). There is no argument for naming this directory. It is actually named ASSEMBLIES.

SUBDIR (assembly) directory

The SUBDIR directory is where the localized assembly is generated, along with some assembly intermediate and evaluation files.

RunAllPathsLG argument: SUBDIR

Required ALLPATHS arguments

The following command-line arguments must be supplied:

PRE – the root directory in which the ALLPATHS pipeline directory will be created.

REFERENCE_NAME – the REFERENCE (organism) directory name - described previously.

DATA_SUBDIR – the DATA (project) directory name - described previously.

RUN – the RUN (assembly pre-processing) directory name - described previously.

SUBDIR – the SUBDIR (assembly) directory name - described previously.

Preparing data for ALLPATHS

Before running ALLPATHS, you must prepare your data for import into the ALLPATHS pipeline. This task will require you to gather the read data in the appropriate formats, and then add metadata to describe them. If you are using a reference genome for evaluation, you will need that as well. This section describes the required data formats.

Supported library constructions

Any input dataset should include at least one *fragment library* and one *jumping library*. A fragment library is a library with a short insert separation, less than twice the read length, so that the reads may overlap (e.g., 100bp Illumina reads taken from 180bp inserts.) A jumping library has a longer separation, typically in the 3kbp-10kbp range, and may include sheared or EcoP15I libraries or other jumping-library construction; ALLPATHS can handle read chimerism in jumping libraries. Note that fragment reads should be long enough to ensure the overlap.

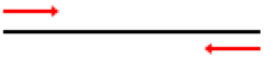
Additionally, ALLPATHS also supports *long jumping libraries*. A jumping library is considered to be long if the insert size is larger than 20 kbp. These libraries are optional and used only to improve scaffolding in mammalian-sized genomes. Typically, long jump coverage of less than 1x is sufficient to significantly improve scaffolding.

ALLPATHS also accepts *long unpaired reads* (e.g., PacBio reads at 50x coverage), which are optional and are used only to patch gaps in the later stages of the assembly process. Currently this is only tested for small, bacterial-sized genomes.

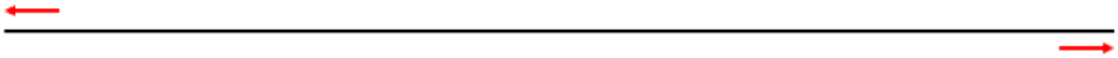
Any other type of library construction is not supported by ALLPATHS at this point.

Read orientation

Fragment library reads are expected to be oriented towards each other (inward):



Jumping library reads are expected to be oriented away from each other (outward), as a result of the typical jumping library construction methods:



Long jumping library reads are expected to be oriented towards each other (inward), as a result of the typical jumping library construction methods:

ALLPATHS input files

The DATA directory must initially hold files containing the sequenced reads, their quality scores and information concerning their pairing. In addition a `ploidy` file must also be present. These files may already exist if you are continuing or restarting an existing assembly, or may be assembled together using tools provided with the ALLPATHS distribution.

Base, quality score, and pairing information files

The read libraries mentioned in the previous section must each be provided as a file containing the bases, a file holding the quality scores, and a file with the pairing and library info. The specific file names are:

```
<REF>/<DATA>/frag_reads_orig.fastb
<REF>/<DATA>/frag_reads_orig.qualb
<REF>/<DATA>/frag_reads_orig.pairs

<REF>/<DATA>/jump_reads_orig.fastb
<REF>/<DATA>/jump_reads_orig.qualb
<REF>/<DATA>/jump_reads_orig.pairs
```

The following long jump files are optional:

```
<REF>/<DATA>/long_jump_reads_orig.fastb
<REF>/<DATA>/long_jump_reads_orig.qualb
<REF>/<DATA>/long_jump_reads_orig.pairs
```

As is the following long unpaired reads file:

`<REF>/<DATA>/long_reads_orig.fastb`

These files can be automatically generated from a set of BAM, fastq, or fasta files as described below.

The `ploidy` file

The file `ploidy` is a single-line file containing a number. As the name suggests, this number indicates the ploidy of the genome with 1 for haploid genomes and 2 for diploid genomes. Polyploid genomes are not currently supported. The specific file name is:

`<REF>/<DATA>/ploidy`

Preparing ALLPATHS input files

The Perl script `PrepareAllPathsInputs.pl` can be used to automatically convert a set of BAM, fasta, fastq, or fastb files to ALLPATHS input files. It will also optionally create the necessary `ploidy` file. This is the easiest way to prepare data for ALLPATHS given a set of files from the Illumina platform. The user must provide as input two comma-separated-values (.csv) files:

```
in_groups.csv
in_libs.csv
```

These describe, respectively, the locations and library information of the various files to be converted.

Accepted data file formats

Each data file must contain paired reads from a single library, but a library may be split over many files. Typically a data file will represent a single lane of an Illumina flowcell.

As mentioned above, currently accepted formats are .bam, .fastq, and .fasta. The quality scores for .fasta files are expected in corresponding .quala files.

For .fastq files you MUST check how the quality scores are encoded. By default it is assumed that the quality scores are encoded using ASCII 33 to 126. If the quality scores are encoded using ASCII 64 to 126 you MUST specify the option `PHRED_64=1` when running the conversion script (this is described below). If you have fastq files with a mix of the two phred encodings refer to the end of the manual for instructions.

`in_groups.csv` file

Each line in `in_groups.csv` provides, for each data file, the following information:

`group_name`: a UNIQUE nickname for this specific data set.

`library_name`: the library to which the data set belongs.

`file_name`: the absolute path to the data file. Wildcards '*' and '?' are accepted (but not in the extension) when specifying multiple files as in the case of two paired or multiple unpaired fastq or fasta files. Supported extensions are: '.bam', '.fasta', '.fa', '.fastq', '.fq', '.fastq.gz', and '.fq.gz', all case-

insensitive. For '.fasta' and '.fa' it is expected that corresponding '.quala' and '.qa' files exist, respectively.

Example `in_groups.csv`:

```
group_name, library_name,                       file_name
       302, Illumina_011,      /seq/Illumina/011/302.bam
       303, Illumina_012,  /seq/Illumina/012/303.?.fasta
       100,   PacBio_007, /seq/PacBio/007/100.*.fastq.gz
```

`in_libs.csv` file

Each line in `in_libs.csv` describes a library. The specific fields are:

`library_name`: matches the same field in `in_groups.csv`.

`project_name`: a string naming the project.

`organism_name`: the organism.

`type`: `fragment`, `jumping`, `EcoP15`, etc. This field is only informative.

`paired`: `0`: Unpaired reads; `1`: paired reads.

`frag_size`: average number of bases in the fragments (only defined for FRAGMENT libraries).

`frag_stddev`: estimated standard deviation of the fragments sizes (only defined for FRAGMENT libraries).

`insert_size`: average number of bases in the inserts (only defined for JUMPING libraries; if larger than 20 kb, the library is considered to be a LONG JUMPING library).

`insert_stddev`: estimated standard deviation of the inserts sizes (only defined for JUMPING libraries).

`read_orientation`: `inward` or `outward`. Outward oriented reads will be reversed.

`genomic_start`: index of the FIRST genomic base in the reads. If non-zero, all the bases before `genomic_start` will be trimmed out.

`genomic_end`: index of the LAST genomic base in the reads. If non-zero, all the bases after `genomic_end` will be trimmed out.

Here is an example `in_libs.csv` (NOTE: all the fields should be on a single line; that makes the lines too long to show here, hence the '...'):

```
library_name, project_name, organism_name,     type, paired, ...
Illumina_011,      Awesome,        E.coli, fragment,      1, ...
Illumina_012,      Awesome,        E.coli,  jumping,      1, ...
  PacBio_007,      Awesome,        E.coli,     long,      0, ...

     ... frag_size, frag_stddev, insert_size, insert_stddev, ...
     ...       180,          10,           ,              , ...
     ...         ,           ,        3000,           500, ...
     ...         ,           ,           ,              , ...

                ... read_orientation, genomic_start, genomic_end
                ...           inward,             ,
                ...          outward,             ,
                ...                 ,             ,
```

Running conversion script

Simplest example of a `PrepareAllPathsInputs.pl` run:

```
PrepareAllPathsInputs.pl
  DATA_DIR=<full_path to REFERENCE DIR>/mydata
  PICARD_TOOLS_DIR=/opt/picard/bin
```

where `DATA_DIR` is the location of the ALLPATHS DATA directory where the converted reads will be placed, and `PICARD_TOOLS_DIR` is the path to the Picard tools needed for data conversion, if your data is in BAM format. There are other options that can be specified:

`IN_GROUPS_CSV` – use a file other than `./in_groups.csv`.

`IN_LIBS_CSV` – use a file other than `./in_libs.csv`.

`INCLUDE_NON_PF_READS` – `1`:(default) include non-PF reads. `0`: include only PF reads.

`PHRED_64` – (for 'fastq' files only) `0`:(default) – provided fastq's have quality scores encoded with ASCII 33 to 126. `1`: ASCII 64 to 126.

`PLOIDY` – generate the `ploidy` file. Valid values are 1 or 2.

`HOSTS` – list of hosts to use in parallel by forking (NOTE: forking to remote hosts requires password-less ssh access, e.g. using ssh-agent/ssh-add). Example: `'2,3.host2,4.host3'` which translates to:

- 2 processes forked on the localhost;
- 3 processes forked on host2;
- 4 processes forked on host3.

The following options allow the user to select, randomly, a fraction of the total number of reads:

FRAG_FRAC – fraction of fragment reads to include, e.g. 30% or 0.3.

JUMP_FRAC – fraction of jumping reads to include, e.g. 20% or 0.2.

LONG_JUMP_FRAC – fraction of long jumping reads to include, e.g. 90% or 0.9.

GENOME_SIZE – estimated genome size for the purpose of coverage estimation.

FRAG_COVERAGE – fragment library desired coverage, e.g. 45. Requires GENOME_SIZE.

JUMP_COVERAGE – jumping library desired coverage, e.g. 45. Requires GENOME_SIZE.

LONG_JUMP_COVERAGE – jumping library desired coverage, e.g. 1. Requires GENOME_SIZE (typically, only very low coverage is required for long jumps).

Note, however, that there are some restrictions on the above options. If you specify FRAG_FRAC or JUMP_FRAC, you cannot also specify FRAG_COVERAGE or JUMP_COVERAGE. If you specify FRAG_COVERAGE or JUMP_COVERAGE you must specify GENOME_SIZE, since both values are necessary for the calculation of the read fraction to include.

After a successful run of PrepareAllPathsInputs.pl the necessary ALLPATHS input files should be in place and ready for an assembly run to start.

Import reference

If you plan to perform evaluations, you can import a reference genome into the pipeline directory at the same time as the read data. The reference genome to import is specified using the argument:

```
REFERENCE_DIR=<directory containing reference>
```

The reference genome must be supplied as two files: genome.fasta and genome.fastb. The fastb file is a binary version of the fasta file. You can convert from fasta to fastb using the ALLPATHS module Fasta2Fastb.

This argument is ignored if a reference genome already exists in the REFERENCE directory. It will not cause an existing reference genome in the pipeline directory to be overwritten.

Once the reference has been imported into the REFERENCE directory, you can omit the REFERENCE_DIR argument when running RunAllPathsLG.

Instead of using the REFERENCE_DIR argument, you may simply create the REFERENCE directory and place the reference genome files in it. The reference genome files must be named:

```
genome.fasta          and
genome.fastb
```

Running ALLPATHS – in brief

Once the read data has been imported you may run the ALLPATHS pipeline as often as desired, each time with different assembly parameters. Each time you run the ALLPATHS pipeline it will determine which modules need to run (or re-run) depending on the parameters you have chosen. Unless you want to overwrite your previous assembly, specify a new RUN directory each time.

This section briefly describes the `RunAllPathsLG` arguments commonly used to run the ALLPATHS pipeline. Complete descriptions of all arguments are provided in the ALLPATHS Reference.

evaluation mode - Given a reference genome, the pipeline can perform evaluations at various stages of the assembly process and of the assembly itself. To turn evaluation on, set `EVALUATION=STANDARD`.

targets –The value of the `TARGETS` parameter determines the operations performed by the pipeline:

`TARGETS=full_eval`	Runs a version of the pipeline that includes additional evaluation modules.
`TARGETS=standard`	Runs a streamlined version of the pipeline that skips many of the evaluation modules.

parallelization - The pipeline has two levels of parallelization. It can run two or more modules concurrently if their dependencies are independent. Many individual modules are also capable of being parallelized via multithreading. By default, only multithreaded parallelization is on. See the ALLPATHS Reference for more details.

Example

The `TARGETS` argument of `RunAllPathsLG` determines whether the ALLPATHS pipeline runs to completion or imports the data and stops. To run an assembly using previously imported data use:

```
TARGETS=standard
```

For example, for data imported using `PrepareAllPathsInputs.pl` with `DATA_SUBDIR=<user pre>/staph/mydata` use:

```
RunAllPathsLG
  PRE=<user pre>
  DATA_SUBDIR=mydata
  RUN=myrun
  REFERENCE_NAME=staph
  TARGETS=standard
```

This will create (if it doesn't already exist) the following pipeline directory structure:

```
<user pre>/staph/mydata/myrun
```

Where `staph` is the REFERENCE directory, `mydata` is the DATA directory containing the imported data, and `myrun` is the RUN directory.

Pipeline errors

The pipeline will stop when it encounters an error. There are two types of error that can occur:

> **rule consistency check error** - Before any modules are called, `RunAllPathsLG` checks to see if it knows how to make all the output files for the given assembly parameters. If not, the pipeline halts immediately before any modules are run, reporting the files that it does not know how to make. Check and correct your arguments and try again.
>
> **runtime consistency check error** - After each module in the pipeline has completed, the pipeline checks to see if correct output files were created. If any files are missing, the pipeline halts, reporting the missing files and the module that failed to produce them. This most often occurs when a module crashes. Check the log for an error message from the module in question.

Once the error has been identified and corrected, re-run the `RunAllPathsLG` command. The pipeline restarts at the point it previously failed.

The ALLPATHS assembly

Assembly as a graph

Unlike a conventional genome assembly, an ALLPATHS assembly is a graph. Edges in this graph represent base sequences, and each path through the graph represents a possible solution to the assembly problem. An ideal assembly would be a single edge, with occasional blips corresponding to SNPs in a diploid genome. However, uncorrected sequencing errors, unresolved repeat structures, and assembly algorithm inadequacies result in ambiguity. By representing the assembly as a graph we can capture this ambiguity rather than arbitrarily choosing a solution and therefore losing information.

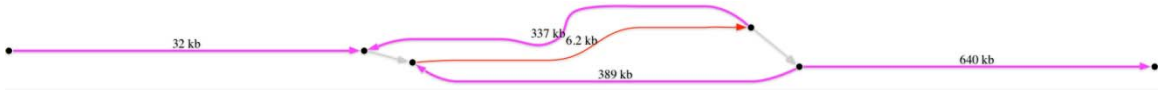
Graph features

A graph assembly consists of *components* and *edges*. A component is a collection of connected edges. An assembly may consist of a number of components, scaffolded together as in a linear assembly.

In the following examples the edge lengths are not to scale. Purple represents long edges; red, medium sized edges; black, short edges; and grey, very short edges.

Repeats

The graph below contains a 6.2 kb repeat that occurs 3 times in the genome. The repeat is longer than the largest insert size available and so could not be resolved. However we do know the two possible orderings of edges and can represent this in a graph.



Homopolymers

With short reads, long homopolymer runs can be difficult to resolve. Rather than assuming a value for the homopolymer length, they are represented as a loop of length 1 base.

SNPs and base errors

When the reads offer two seemingly equally possible alternatives for a base, we represent this as a small bubble. This situation can arise from SNPs, in which case the bubble is "correct", but it may also be due to particularly hard-to-correct base substitution errors in the raw reads. In a conventional assembly, bases of low quality would represent these ambiguities.

Flattening and Scaffolding

The graph assembly mentioned in the previous section provides the most complete description of the genome. However, this graph incompatible with existing annotation and analysis tools and is also troublesome to scaffold. To overcome these problems we attempt to flatten the graph, retaining as much ambiguity as possible, then scaffold to create a (nearly) conventional assembly consisting of scaffolded linear contigs.

The flattening and scaffolding processes are given elsewhere, but the end result is a pair of files – a conventional `fasta` file plus a related `efasta` file. These files are described in detail in the next section.

Assembly Results

The results of the assembly pipeline are given in the following two files in the assembly `sub_dir` directory:

```
final.assembly.fasta

final.assembly.efasta
```

Both these files contain the final flattened and scaffolded assembly. The `efasta`, "enhanced" `fasta`, file is a new format used by ALLPATHS and is based on the standard `fasta` file format.

fasta format

The `fasta` file contains contigs that have been scaffolded together, separated by `n`, where the number of `n`s represents the best estimate of the gap size between contigs. A single `n` represents an unresolved negative gap. Within each contig, an ambiguous base is represented by an N. For example, an A/T SNP would become: ATGTC**N**TGTCG.

efasta format

The `efasta` file contains contigs that have been scaffolded together, separated by N, where the number of Ns represents the best estimate of the gap size between contigs. A single N represents an unresolved negative gap. Within each contig, ambiguity is represented by an expression within a pair of braces, {}. For example, an A/T SNP would become: ATGTC**{A,T}**TGTCG. An unresolved homopolymer run of T, where the evidence suggested there should be 6, 7 or 8 Ts, would become: GTCACTTTTTT**{,T,TT}**GCTGT. In this enhanced `fasta` format simple ambiguities that would otherwise be lost are now retained. The `efasta` format can be easily flattened by picking the first option for each ambiguity, resulting in the assembly given in the associated `fasta` file. Within the braces the options are ordered in terms of decreasing likelihood.

ALLPATHS Reference

ALLPATHS Cache for power users

The Perl script `PrepareAllPathsInputs.pl` that imports data to ALLPATHS is in fact a wrapper around a few tools. It first creates a temporary cache of fastb/qualb files in `<DATA>/read_cache/` for each data file described in `in_groups.csv`. Then, it automatically merges all these cached files into each of the input files expected by ALLPATHS.

Alternatively, a non-temporary cache can be created separately at a different location and it can work as a repository of data for many different projects. The advantage of having a cache is that it separates the time-consuming step of converting data files (especially BAM files) to the fastb and qualb format from the merging of the fastb and qualb files into ALLPATHS input files. This is useful, for example, when a user wants to run different assemblies based on different subsets of the original data.

Creating the ALLPATHS Cache

The ALLPATHS cache stores all the information regarding the libraries and groups in two files in the cache directory: `libraries.csv` and `groups.csv`. To build the cache the following commands need to be run:

```
CacheLibs.pl
  CACHE_DIR=<CACHE_DIR>
  IN_LIBS_CSV=in_libs.csv
  ACTION=Add

CacheGroups.pl
  CACHE_DIR=<CACHE_DIR>
  PICARD_TOOLS_DIR=/opt/picard/bin
  IN_GROUPS_CSV=in_groups.csv
  TMP_DIR=/large-tmp
  HOSTS='2,3.host2,4.host3'
  ACTION=Add
```

The `CacheLibs.pl` command simply adds the library information in `in_libs.csv` to the cache `libraries.csv`. The `CacheGroups.pl` command converts all the data files described in `in_groups.csv` to fastb and qualb files in the cache, and adds the corresponding entries to the cache `groups.csv`. The common options are:

`CACHE_DIR` – the full path to the cache directory. Can be omitted if the environment variable `ALLPATHS_CACHE_DIR` is defined.

`ACTION` – `Add`, `List`, or `Remove` entries to, in, and from the cache.

`CacheLibs.pl` options:

`IN_LIBS_CSV` – alternative file to the default `./in_libs.csv`.

`GroupLibs.pl` options:

`PICARD_TOOLS_DIR` – the full path to the Picard tools needed for data conversion. Can be omitted if the environment variable `ALLPATHS_PICARD_TOOLS_DIR` is defined.

`IN_GROUPS_CSV` – alternative file to the default `./in_groups.csv`.

`TMP_DIR` – the full path of a local temporary directory; must be large if your data is large.

`INCLUDE_NON_PF_READS` – `1`:(default) include non-PF reads. `0`: include only PF reads.

`HOSTS` – list of hosts to use in parallel by forking. Each fork converts a single data file (NOTE: forking to remote hosts requires password-less ssh access, e.g. using ssh-agent/ssh-add). Example: `'2,3.host2,4.host3'` which translates to:

- 2 processes forked on the localhost;
- 3 processes forked on host2;
- 4 processes forked on host3.

Finally, the contents of the cache can easily be listed by running:

```
CacheGroups.pl
  CACHE_DIR=<CACHE_DIR>
  ACTION=List
```

Importing fastq files of mixed PHRED encoding

When you have some fastq files encoded PHRED+64 and others PHRED+33 you will have to import them to the cache separately.

First import the libraries with the CacheLibs.pl command as described in the previous section.

Then, create two files (`in_groups_33.csv` and `in_groups_64.csv`) with the corresponding fastq groups.

Run the two following commands to import the groups to the cache:

```
CacheGroups.pl
  CACHE_DIR=<CACHE_DIR>
  IN_GROUPS_CSV=in_groups_33.csv
  PHRED_64=False
  …<plus other relevant options>
```

```
CacheGroups.pl
  CACHE_DIR=<CACHE_DIR>
  IN_GROUPS_CSV=in_groups_64.csv
  PHRED_64=True
  …<plus other relevant options>
```

Now that you have successfully and correctly imported your groups to the cache, follow the instructions in the next section to generate the ALLPATHS input files.

Using the ALLPATHS Cache

Once the cache is created it can be used to generate the ALLPATHS input files:

```
<DATA>/frag_reads_orig.fastb
<DATA>/frag_reads_orig.qualb
<DATA>/frag_reads_orig.pairs

<DATA>/jump_reads_orig.fastb
<DATA>/jump_reads_orig.qualb
<DATA>/jump_reads_orig.pairs

<DATA>/long_jump_reads_orig.fastb
<DATA>/long_jump_reads_orig.qualb
<DATA>/long_jump_reads_orig.pairs

<DATA>/long_reads_orig.fastb
```

The command to generate the ALLPATHS input files is:

```
  CacheToAllPathsInputs.pl
    CACHE_DIR=<CACHE_DIR>
    GROUPS="{12345AAXX.{1,2,3},67890ABXX.{6,7}}"
    DATA_DIR=<DATA_DIR>
    FRAG_FRAC=50%
    JUMP_FRAC=34%
```

The options are:

CACHE_DIR – the path to the cache directory. Can be omitted if the environment variable ALLPATHS_CACHE_DIR is defined.

DATA_DIR – the full path to the ALLPATHS DATA directory where the input files will be placed.

GROUPS – a list of the groups to include as inputs.

IN_GROUPS_CSV – file including the groups description. Optional alternative to GROUPS.

FRAG_FRAC – fraction of fragment reads to include, e.g. 30% or 0.3.

JUMP_FRAC – fraction of jumping reads to include, e.g. 20% or 0.2.

LONG_JUMP_FRAC – fraction of long jumping reads to include, e.g. 90% or 0.9.

FRACTIONS – (use with GROUPS only) list of fractions, one per group, e.g. "{0.5,30%,100%}".

GENOME_SIZE – estimated genome size for the purpose of coverage estimation.

FRAG_COVERAGE – (requires GENOME_SIZE) fragment library desired coverage, e.g. 45.

JUMP_COVERAGE – (requires GENOME_SIZE) jumping library desired coverage, e.g. 45.

LONG_JUMP_COVERAGE – (requires GENOME_SIZE) jumping library desired coverage, e.g. 1. (typically, only very low coverage is required for long jumps).

COVERAGES – (use with GROUPS only, requires GENOME_SIZE) list of coverages, one per group, e.g. "{45,50,2}".

LONG_READ_MIN_LEN – (default 500) this sets the threshold for what qualifies as a long unpaired read (e.g. PacBio reads).

As in PrepareAllPathsInputs.pl, there are some restrictions on the above options. If you specify FRAG_FRAC, JUMP_FRAC, or FRACTIONS, you cannot also specify FRAG_COVERAGE or JUMP_COVERAGE, or COVERAGES. If you specify FRAG_COVERAGE, JUMP_COVERAGE, or COVERAGES you must specify GENOME_SIZE, since both values are necessary for the calculation of the read fraction to include. If you specify one of FRACTIONS and COVERAGES lists you must specify a GROUPS list and supply one fraction or coverage entry for each group.

After a successful run of CacheToAllPathsInputs.pl the necessary ALLPATHS input files should be in place and ready for an assembly run to start.

ALLPATHS compilation options

The following command-line options may be appended to make when building ALLPATHS:

-j<n> Split the compilation into n parallel processes. If you set n equal to the number of CPUs on your machine, it will speed up compilation approximately n-fold. See Installation for an example.

ALLPATHS pipeline – in detail

Key Features

The ALLPATHS pipeline incorporates the following key features:

- Runs only those modules that are required for a particular set of parameters.
- Ensures intermediate files are always consistent.
- If the parameters for a module change, reruns only the changed module and modules that depend on its output.
- In the event of a problem, restarts at the point the problem occurred.
- Supports easy parallelization by allowing modules that don't depend on each other's output to run concurrently.
- Can easily be run up to any point.
- Can initially exclude modules that are not required for the assembly process (evaluation modules for example), then easily run them once the assembly is complete.
- Determines if it has all the necessary input files and knows how to build all the requested output files before starting any modules. Stops immediately if there is a problem.

Directory structure – ALLPATHS_BASE

In addition to using the command-line argument PRE to specify the location of the pipeline directory, you may optionally also use ALLPATHS_BASE. The pipeline directory location is either:

PRE

or

PRE/ALLPATHS_BASE

Targets

The pipeline determines which output files it needs to generate by means of a list of targets. If a particular target file is requested, then the modules required to create both it, and any intermediate files it depends on, will be run in the correct order. Only these modules will be run. Further, if any required intermediate files already exist and are up to date with respect to the files that they in turn depend on, then the call to the module required to build them is skipped. This holds true for the final target file or files – if they already exist and are up to date then nothing will be done.

You can specify the target files to build in two ways. The simplest is to use one of the predefined pseudo targets that represent a set of useful target files – much like pseudo targets in `Make`. The second is to specify a list of individual files that the pipeline knows how to make. Both methods may be used at the same time.

If you ask for a target file that the pipeline doesn't know how to make you will get an error message.

Pseudo targets

This is the best way to control which files the pipeline will create. The pseudo target value is passed to `RunAllPathsLG` using:

```
TARGETS=<pseudo target name>
```

There are 3 possible pseudo targets:

none – no pseudo targets, only make explicitly listed target files (see below).

`standard` – create the assembly and selected evaluation files.

`full_eval` – create the assembly and additional evaluation files.

The default target is `standard`.

Target files

Individual files may be specified as targets instead of, or in addition to, the pseudo targets. Lists of target files in each pipeline subdirectory are passed to `RunAllPathsLG` using:

```
TARGETS_DATA=<target files in the DATA dir>

TARGETS_RUN=<target files in the RUN dir>

TARGETS_SUBDIR=<target files in the SUBDIR dir>
```

Multiple target files may be passed in the following manner:

```
TARGETS_RUN="{target1,target2,target3}"
```

The list of valid target files changes based on the assembly parameters chosen.

Evaluation mode

Given a reference genome, the pipeline can perform evaluations at various stages of the assembly process.

Certain evaluations have the potential to alter the assembly, as they require reference genome data to be incorporated into data structures used by the assembly process. Any such perturbation of the assembly should be neutral but will have a stochastic effect on the result. Such 'unsafe' evaluations allow much more detailed information to be gathered about the assembly process and are extremely useful during development, but can be considered "cheating" from the point of view of *de novo* assembly.

The evaluation mode used is controlled by:

```
EVALUATION=<evaluation mode>
```

There are three evaluation modes:

`NONE` – do not evaluate/no reference is available.

`BASIC` – basic evaluation that does not require a reference.

`STANDARD` – run evaluation modules using a supplied reference.

`FULL` – turn on in-place evaluation in certain assembly modules. Does not perturb assembly.

CHEAT – run in-place evaluations that potentially perturb the assembly (in a neutral fashion), but allow a more detailed analysis.

The default mode is `BASIC`.

Kmer size, K

The user should **not** adjust the kmer size from the default value of K=96.

The relationship between kmer size K and read size is not a direct one in ALLPATHS-LG, unlike in may other assemblers. ALLPATHS-LG actually uses a number of different sizes of K internally, and because of this, it is not intended that users change the K values for an assembly.

Parallelization

Given sufficient memory, it is possible to parallelize the pipeline in order to reduce runtime. Two forms of parallelization are possible and both may be used at the same time.

Cross-module parallelization

Modules in the pipeline that do not depend on each other may be run concurrently. This functionality is provided by `make`, which is used by `RunAllPathsLG` to execute the pipeline. It is equivalent to using the option `-j<n>` when compiling the ALLPATHS source code. No checks are made to ensure that there is enough memory to run multiple ALLPATHS modules at the same time. Set the maximum number of modules that can run concurrently using:

MAXPAR=<n>

The majority of the pipeline now uses parallel threading, so in most cases there is little to be gained in setting this value about 1.

Parallelization of individual modules

Many of ALLPATHS's modules have been engineered to run with parallel threading. This form of parallelization is independent of the module parallelization described above. The level of parallelization can be controlled using the argument to `RunAllPathsLG:`

THREADS=<n>

For maximum performance, set this value to the number of processors available – but be wary of exceeding available memory as the number of threads increases. Due to hardware restraints (such as I/O limiting and heap contention) you will find diminishing returns in runtime improvement as the number of threads increases.

By default the pipeline will attempt to use all available processors.

Logging

In addition to standard out, the output from each ALLPATHS module is captured to file. In each pipeline directory there exists a subdirectory named `makeinfo` that contains various logging files plus

metadata used by the pipeline to control and track progress. Every single file produced by the pipeline will have two log files associated with it. For example, the file h`yper.fasta` will have the following log files in `SUBDIR/makeinfo`:

`hyper.fasta.cmd`

`hyper.fasta.DumpHyper.out`

The `.cmd` file contains the command used to generate `hyper.fasta`. The `.out` file contains the captured output of the module used to create `hyper.fasta`. In this case the module is called `DumpHyper`, as you would see from looking at the file `hyper.fasta.cmd`.

References

Gnerre S, MacCallum I, Przybylski D, Ribeiro F, Burton J, Walker B, Sharpe T, Hall G, Shea T, Sykes S, Berlin A, Aird D, Costello M, Daza R, Williams L, Nicol R, Gnirke A, Nusbaum C, Lander ES, Jaffe DB. High-quality draft assemblies of mammalian genomes from massively parallel sequence data *Proceedings of the National Academy of Sciences* January 2011 vol. 108 no. 4 1513-1518

MacCallum I, Przybylski D, Gnerre S, Burton J, Shlyakhter I, Gnirke A, Malek J, McKernan K, Ranade S, Shea TP, Williams L, Young S, Nusbaum C, Jaffe DB. ALLPATHS 2: small genomes assembled accurately and with high continuity from short paired reads. *Genome Biology* 2009, **10**(10):R103.

Butler J, MacCallum I, Kleber M, Shlyakhter IA, Belmonte MK, Lander ES, Nusbaum C, Jaffe DB. ALLPATHS: De novo assembly of whole-genome shotgun microreads, *Genome Res.* May 2008 **18**:810-820.