ALLPATHS v3 Manual

Computational Research and Development Group

Genome Sequencing and Analysis Program

Broad Institute of MIT and Harvard

Cambridge, MA

Manual Revision: (12/3/09 11:56 AM)

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:

```
RunAllPaths3G PRE=/assemblies DATA=datadir RUN=rundir SUBDIR=attempt1
```

becomes

```
RunAllPaths3G
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 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 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 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 40x 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 several Illumina lanes, though Illumina technology is constantly improving in throughput.

ALLPATHS requires a minimum of 2 paired-end libraries – one short and one long. The short library average separation size must be 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 4000 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.

Requirements

To compile and run ALLPATHS you will need a Linux/UNIX system with at least 32 GB of RAM. We suggest a minimum of 128 Gb, and 512 Gb for mammalian sized genomes. You will also need the following software:

The g++ compiler, version 4.3.3 or higher. We use version 4.3.3.
http://gcc.gnu.org/

The C++ Boost library. We use version 1.38.
http://www.boost.org/

The graph command `dot` from the `graphviz` package. We use version 2.16.1.
http://www.graphviz.org/

The traceback utility `addr2line` (from the `binutils` package), provided by the Free Software Foundation. http://www.gnu.org/

Availability

The ALLPATHS source code is available for download at:

http://www.broadinstitute.org/science/programs/genome-biology/crd

The current version is ALLPATHS v3, which is zipped into the file `allpaths-3.1.tar.gz`.

Getting Help

If you encounter difficulties that cannot be resolved using this manual you can contact the ALLPATHS development team via:

CRDHelp@broad.mit.edu

Installation

After you have downloaded the file `allpaths-3.1.tgz`, unpack it using `gunzip` and `tar xvf`. Then you can simply compile the source code with `configure` and `make`. All of the source code should be in its own directory called `AllPaths`; we will refer to this as the `AllPaths` directory. For example, starting from the root directory (the location of the downloaded file):

```
% gunzip allpaths-3.1.tgz          // unzip file allpaths-3.1.tgz into allpaths-3.1.tar

% tar xvf allpaths-3.1.tar         // expand the tarball; create subdir AllPaths

% cd AllPaths                      // switch into the source directory

% autoconf                         // create autoconfig script

% ./configure                      // run autoconfig script

% make -j8                         // build ALLPATHS (use -j<n> to parallelize compilation)

% make install_scripts             // install perl scripts used by ALLPATHS
```

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, e.g., `configure --with-boost=/path/to/boost/`. 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 `AllPaths`. 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 `RunAllPaths3G` 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 `RunAllPaths3G`.

RunAllPaths3G module

`RunAllPaths3G` 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 `RunAllPaths3G` 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 `RunAllPaths3G` 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 `RunAllPaths3G` 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 `RunAllPaths3G` 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 (non-cheating) 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.

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

RunAllPaths3G 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.

RunAllPaths3G 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 (LocalizeReads3G and beyond). It may also contain intermediate files used in evaluation that are dependent on the assembly parameters chosen.

RunAllPaths3G 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.

RunAllPaths3G 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.

`K` – the kmer size used for assembly - described later.

Preparing read data

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 and how to access the example data sets that we provide.

`SOURCE_DIR` directory

All source data should be placed in a directory, known as `SOURCE_DIR`, which is independent of the ALLPATHS directory structure. You will supply the location of `SOURCE_DIR` to `RunAllPaths3G`, and it will import the data from there into the `DATA` directory (described previously).

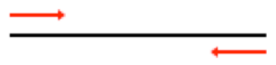
Supported library constructions

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 library. Note that fragment reads should be longer (~100bp, to ensure the overlap) but jumping reads do not need to be.

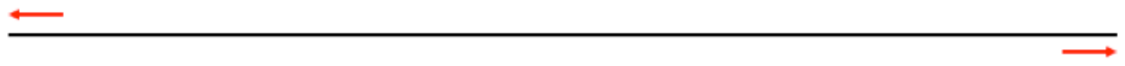
ALLPATHS does not currently support data from other library construction methods, including unpaired reads.

Read orientation

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



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



Reads and quality scores

The reads should be in fasta format and the associated quality scores should be in quala format. You may have more than one pair of read and quality score files. These files must meet the following conditions:

- Each fasta file must have an associated quala file, i.e., for the file `foo.fasta` there must be a corresponding `foo.quala` with exactly the same number and lengths of reads.
- Each pair of fasta and quala files should contain reads from a single library. However, reads from the same library may be split over multiple fasta and quala files – there is no need to combine them.
- For paired reads, the files should appear in pairs labeled A and B corresponding to the read pairings. That is, you should have two files named `foo.A.fasta` and `foo.B.fasta` (along with their .quala files) in which the first read in `foo.A.fasta` pairs with the first read in `foo.B.fasta`, the second read in `foo.A.fasta` pairs with the second read in `foo.B.fasta`, and so forth.

quala files

The quala (also called qual) sequence format is a fasta-like format that stores numerical quality score values for each base in a corresponding fasta file.

Example quala file (first 3 reads):

```
>sequence_0
40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
40 40 40 40 40 40 40 40 40 40 40 40

>sequence_1
40 40 40 40 40 40 40 40 40 40 40 40 40 40 37 40 40 40 40 40 40 40 28
40 40 40 40 40 40 40 40 40 40 40 25

>sequence_2
40 40 40 40 40 40 13 10 40 40 40 12 40 13 10 28 24 28 37 40 32 21 13
24 24 40 5 13 12 29 14 3 2 2 11
```

`frag_reads_lib_stats` and `jump_reads_lib_stats` files

These files contain the metadata that describes the read pair libraries used and links this library information with the fasta and quala files. The `frag_reads_lib_stats` file describes the fragment libraries, whilst the `jump_reads_lib_stats` file describes the jumping libraries. In both files the first line denotes the six columns of information in the file and must be entered exactly as follows:

```
FILE LIBRARY_NAME PAIRED JUMPING SEP DEV
```

Each subsequent line describes a fasta file in `SOURCE_DIR`. Each line contains the following information, separated by spaces:

FILE – the fasta filename. Every fasta file in `SOURCE_DIR` should be listed here.

LIBRARY_NAME – a unique name for this read pair library. Paired reads are grouped by library for the purposes of evaluating library statistics. This value is ignored for unpaired reads.

PAIRED – is this a paired library? Paired fasta files should listed one after the other. ALLPATHS V3 does not currently handle unpaired reads. (`T/F`)

JUMPING – is this a jumping library? For `frag_reads_lib_stats` this values should always be false and for `jump_reads_lib_stats` it should always be true. (`T/F`)

SEP – for a paired read library, the expected separation between the two reads, *not including the read lengths themselves*. The value should be an estimate of the mean of the distribution of separations in the library. It should be the same for all fasta files in a single library, but may vary between libraries.

DEV – for a paired read library, the standard deviation of the pair separation above.

For example, for a paired read jumping library called `201FK` with separation 3600 bases and standard deviation 540 bases, with associated fasta files `reads_orig.201FK.5.A.fasta` and `reads_orig.201FK.5.B.fasta`, the entries would be:

```
reads_orig.201FK.5.A.fasta 201FK T T 3600 540
reads_orig.201FK.5.B.fasta 201FK T T 3600 540
```

Example `jump_reads_lib_stats` file describing the jumping library:

```
FILE LIBRARY_NAME PAIRED JUMPING SEP DEV
reads_orig.201FK.5.A.fasta 201FK T T 3600 540
reads_orig.201FK.5.B.fasta 201FK T T 3600 540
```

Similarly, for a paired read fragment library called `13229` with separation 180 bases and standard deviation 20 bases, with associated fasta files `reads_orig.13229.1.A.fasta` and `reads_orig.13229.1.B.fasta`, the entries would be:

```
reads_orig.13229.1.A.fasta 13229 T F 180 20
reads_orig.13229.1.B.fasta 13229 T F 180 20
```

Example `frag_reads_lib_stats` file describing the fragment library:

```
FILE LIBRARY_NAME PAIRED JUMPING SEP DEV
reads_orig.13229.1.A.fasta 13229 T F 180 20
reads_orig.13229.1.B.fasta 13229 T F 180 20
```

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.

Importing data into the pipeline

The SOURCE_DIR is only required the first time you run ALLPATHS for a particular set of read data. After the initial import this directory can be removed or moved. There is no further need to reference it. The newly created pipeline directory structure now contains all that is required to run assembly experiments on the read data – the original data has been imported into the DATA directory.

Import read data

The TARGETS argument of RunAllPaths3G determines whether the ALLPATHS pipeline runs to completion or imports the data and stops. To import the read data into the pipeline directory structure and then stop, use the following option:

```
TARGETS=import
```

For example, to import data from a directory called /reads/staphdata, use:

```
RunAllPaths3G
PRE=<user pre>
DATA_SUBDIR=MyTestData
RUN=MyRun
REFERENCE_NAME=Staph
TARGETS=import
SOURCE_DIR=/reads/staphdata
K=96
```

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

```
<user pre>/Staph/MyTestData/MyRun
```

Where Staph is the REFERENCE directory, MyTestData is the DATA directory containing the imported data, and MyRun is the RUN directory that for the moment is empty.

Note that once the data has been imported into the DATA directory in this manner, the pipeline will ignore any attempts to overwrite it – for example, by specifying a different SOURCE_DIR. To replace the data you have imported you must delete the DATA directory.

The pipeline now runs independently of SOURCE_DIR. From this point onwards you can omit the SOURCE_DIR argument when running RunAllPaths3G.

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 `RunAllPaths3G`.

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 `RunAllPaths3G` 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=REFERENCE`.

kmer size, `K` - The kmer size, `K`, is restricted by the smallest fragment read size in the read data to assemble. The value of `K` must be smaller than this size, and only certain values are supported. For 100-bp fragment reads, the suggested value is `K=96`.

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

`TARGETS=import`	Imports the read data and stops.
`TARGETS=all`	Runs the entire pipeline to completion, including all 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. Several individual modules are also capable of being parallelized via multithreading. By default, these forms of parallelization are on, which will speed up your run but may cause problems on some machines. To turn parallelization off, set `PARALLELIZE=False`. See the ALLPATHS Reference for more details.

Example

The `TARGETS` argument of `RunAllPaths3G` 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 with DATA_SUBDIR=MyTestData use:

```
RunAllPaths3G
PRE=<user pre>
DATA_SUBDIR=MyTestData
RUN=MyRun
REFERENCE_NAME=Staph
TARGETS=standard
K=96
```

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

```
<user pre>/Staph/MyTestData/MyRun
```

Where `Staph` is the `REFERENCE` directory, `MyTestData` 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, `RunAllPaths3G` 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 `RunAllPaths3G` command. The pipeline restarts at the point it previously failed.

The ALLPATHS graph-based 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.

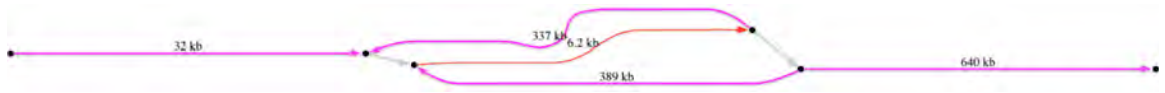
Graph features

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.

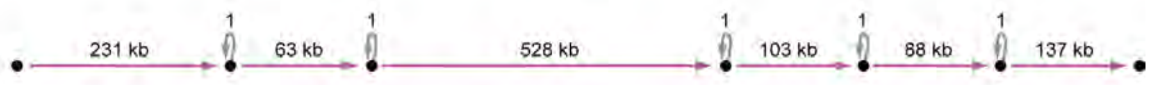
Repeats

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.



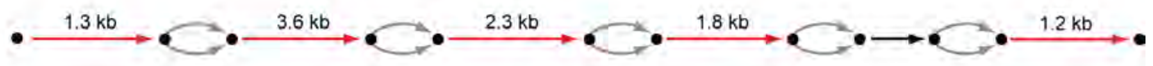
Homopolymers

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.



Basic assembly statistics

The file `SUBDIR/EvalHyper3G/EvalHyper3G.summary.out` contains an evaluation of the final assembly. The number of components, edges and vertices are reported, along with the component and edge N50 sizes. If a reference is available and `EVALUATION=REFERENCE`, then the assembly is evaluated against it, and the result is written to `SUBDIR/EvalHyper3G/EvalHyper3G.align_to_ref.out`.

Viewing the assembly graph

The assembly graph can be viewed as a PostScript image. The file `hyper.dot` in the `SUBDIR` directory contains a description of the graph in the `dot` format. To turn this into a postscript file, use:

```
dot -Tps hyper.dot -o hyper.ps
```

(This requires that the `graphviz` library be installed.) View the resulting image in your favorite postscript viewer, for example `gv`:

```
gv hyper.ps
```

The edges are color-coded as described above (see Graph features).

Edge base sequences

You can decompose the assembly into edges and ignore the additional graph information. The pipeline does this automatically. In `SUBDIR` you will find all the edges in fasta format in the file:

```
hyper.fasta
```

Each edge is represented by a contig in the fasta file.

In addition, unipaths that are not represented in the final assembly graph are identified and then extended unambiguously, where possible. Typically these represent small regions that have relatively high copy number. These extra, unconnected unipaths can be found in the `SUBDIR` directory in the file:

```
hyper.extra.fastb
```

Scaffolds

The assembly graph may be divided into connected components, between which there are no edges. Using paired reads we may form scaffolds, which are linked sequences of one or more such components, separated by gaps. As part of the output of ALLPATHS we convert these graph scaffolds into traditional, linear scaffolds, which are presented via a fasta file with Ns for gaps. This standard output makes the data compatible with existing analytical tools. In these linear scaffolds, ambiguities (unresolved regions of the assembly graph) are replaced by gaps, entailing some loss of information. The scaffold file can be found at `SUBDIR/linear_scaffolds.fasta`.

ALLPATHS Reference

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 `RunAllPaths3G` using:

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

There are 4 possible pseudo targets:

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

`import` – create the pipeline directory structure and import the read data from `SOURCE_DIR`, then stop.

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

`all` – create all known target files, including all evaluation and experimental files (even those that are not needed to create the assembly).

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 `RunAllPaths3G` 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.

`REFERENCE` – perform only safe evaluation against a reference genome.

`CHEAT` – perform detailed evaluation that potentially modifies the assembly.

The default mode is `NONE`.

Kmer size, `K`

The kmer is the building block of the ALLPATHS assembly. The choice of kmer size impacts on many aspects of the assembly process. For a detailed explanation of kmers and how they are used see the original ALLPATHS paper [Butler *et al.* 2008].

The kmer size chosen is restricted by the smallest read size in the read data to assemble. The value of `K` must be smaller than this size and small enough that each read will provide a number of kmers – not just one or two. For 100-bp fragment reads, the suggested value is `K=96`.

The kmer size is passed to `RunAllPaths3G` using:

```
K=<kmer size>
```

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 `RunAllPaths3G` 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>
```

For maximum performance, assuming ample memory, set this value to the number of processors available. In practice, there is very little additional benefit to setting `MAXPAR` above 4, as there are few parts of the pipeline where so many modules can run independently.

Parallelization of individual modules

Several of ALLPATHS's most resource-intensive modules have been engineered to run with parallel threading. This form of parallelization is independent of the module parallelization described above. Each module's level of parallelization can be controlled by a separate argument to `RunAllPaths3G`, as follows:

Module name	Parameter for parallelizing	Default value of parameter
`FastbToKmerParcels`	`KP_THREADS`	8
`MarkDuplicatePairs`	`MDP_THREADS`	8
`FindErrors`	`FE_THREADS`	8
`FillFragments`	`FF_THREADS`	16
`CommonPather`	`CP_THREADS`	16
`CloseUnipathGaps`	`CUG_THREADS`	8
`ErrorCorrectJump`	`ECJ_THREADS`	16
`LocalizeReads3G`	`LR_THREADS`	16

For maximum performance, set these values 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; typically there is little or no speedup beyond 16-way parallelization.

If you set `PARALLEL=False`, these values will all be set to 1, as will `MAXPAR`.

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 `hyper.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

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.