ALLPATHS v2 Manual

Computational Research and Development Group

Genome Sequencing and Analysis Program

Broad Institute of MIT and Harvard

Cambridge, MA

Manual Revision: (3/2/10 3:52 PM)

Table of Contents

ALLPATHS v2 Manual 1

Conventions 5

Introduction 5

Capabilities and limitations 5

Requirements 6

Availability 6

Installation 7

Environment 7

ALLPATHS pipeline overview 7

RunAllPaths module 7

ALLPATHS pipeline directory structure 8

REFERENCE (organism) directory 8

DATA (project) directory 8

RUN (assembly pre-processing) directory 9

ASSEMBLIES directory 9

SUBDIR (assembly) directory 9

Required ALLPATHS arguments 9

Preparing read data 9

SOURCE\_DIR directory 10

Supported library constructions 10

Read orientation 10

Reads and quality scores 10

quala files 11

expected\_lib\_stats file 11

ploidy file 12

Example – Staphylococcus aureus 12

Unpacking the example data 12

Examining the example data 12

Importing data into the pipeline 13

Import read data 13

Import reference 14

Example – Staphylococcus aureus 14

Importing example data: reads only 14

Importing example data: reads and a reference genome 15

Running ALLPATHS – in brief 15

Example – Staphylococcus aureus 16

Pipeline errors 17

Examining the assembly 17

Assembly as a graph 17

Graph features 17

Repeats 17

Homopolymers 18

SNPs and base errors 18

Basic assembly statistics 18

Viewing the assembly graph 18

Edge base sequences 18

Scaffolds 19

ALLPATHS Reference 20

ALLPATHS compilation options 20

ALLPATHS pipeline – in detail 20

Key Features 20

Directory structure – ALLPATHS\_BASE 20

Targets 20

Pseudo targets 21

Target files 21

Evaluation mode 21

Kmer size, K 22

Error correction kmer size, ec\_ks 22

Parallelization 22

Module parallelization 23

LocalizeReads parallelization 23

Logging 23

References 24

# 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:

RunAllPaths PRE=/assemblies DATA=data\_dir RUN=run\_dir SUBDIR=attempt1

Becomes

RunAllPaths

PRE=/assemblies

DATA=data\_dir

RUN=run\_dir

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 such as those produced by the new generation of sequencers. The other 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. These reads typically range in size from 30 to 50 bases, although recent improvements have pushed the upper limit to 100 bases. The version described here has been optimized for, but not necessarily limited to, reads of length 30 to 50 bases. A version is currently in development that will better use 100+ base reads.

ALLPATHS requires paired end reads. It can use unpaired reads to improve error correction, but without pairs it will not be able produce an assembly.

ALLPATHS cannot 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 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.

ALLPATHS requires a minimum of 2 paired end libraries – one short and one long. The short library average separation size should be in the range 200 to 500 bases long. 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. In the case of jumping libraries, the reads must also not cross the junction point between the two ends of the insert.

The current version of ALLPATHS can only assemble small haploid genomes. It has been successfully tested on genomes up to 40 Mb in size.

# Requirements

To compile and run ALLPATHS you will need a UNIX/Linux system with at least 8 GB of RAM. More memory is needed for genomes larger than 4 M bases. You will also need the following software:

The g++ compiler, version 4.3.0 or higher. We use version 4.3.0.

<http://gcc.gnu.org/>

The graph command dot from the graphviz package. We use version 2.12.

<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 to download at:

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

The current version is in the file allpaths-2.2.tar.gz. We refer to the directory into which allpaths-2.2.tar.gz is downloaded as the root directory.

In addition to the ALLPATHS source code, you will find sample data for *Escherichia coli* and *Staphylococcus aureus*.

# Installation

Once you finish downloading allpaths-2.2.tar.gz , the next step is to unpack the archive with tar xzf. With the archive unpacked, you can proceed to configure the package. When configuring AllPaths2, we strongly recommend you select a separate directory structure as the installation home for AllPaths2, such as $HOME/local/AllPaths2. We will refer to this as the AllPaths directory. Here is an example of the steps required to install AllPaths2.

|  |  |
| --- | --- |
| % tar xzf allpaths-2.2.tar.gz | Unpack the package. |
| % cd allpaths-2.2 | CD into the package. |
| % ./configure --prefix=$HOME/local/allpaths2.2 | Configure the package. The --prefix is optional, but recommended. |
| % make | Build the package. |
| % make install | Install the package. |

## Environment

After compilation and installation, the executable binary files will be in the subdirectory bin of AllPaths. You may 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. You may need to djust 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 RunAllPaths controls the entire pipeline. Although it is possible to run the individual modules manually, users are expected to rely on RunAllPaths.

## RunAllPaths module

RunAllPaths 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 RunAllPaths 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 RunAllPaths 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 RunAllPaths has been called on a particular set of source data.

## 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 RunAllPaths 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 (or organism) directory is used to separate assembly projects by organism and possibly also by isolate. You should create a REFERENCE directory for each new organism, and all assembly projects of that organism will be contained in that directory.

Given a reference genome, the pipeline can perform evaluations at various stages of the assembly process and of the assembly itself. If a reference genome is available, it should be placed in the 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.

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

RunAllPaths argument: REFERENCE\_NAME

### DATA (project) directory

The DATA directory contains the original read data (but in the ALLPATHS internal format) used in a particular assembly attempt. 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.

RunAllPaths argument: DATA\_SUBDIR

### RUN (assembly pre-processing) directory

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

RunAllPaths argument: RUN

### ASSEMBLIES directory

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

### SUBDIR (assembly) directory

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

RunAllPaths 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

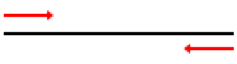
All source data should be placed in a directory that is known as the SOURCE\_DIR. You will import the read data from the SOURCE\_DIR into the DATA directory of the ALLPATHS pipeline directory (described previously). Once the data has been imported, ALLPATHS ignores SOURCE\_DIR.

## Supported library constructions

ALLPATHS has been tested on three types of data: unpaired reads, paired reads obtained by directly reading the ends of small fragments, and 26 base paired reads obtained from EcoP15I jumping libraries. Data from other library construction methods are not supported at this time.

## 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. E.g., 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

## expected\_lib\_stats file

This file contains the metadata that describes the read pair libraries used and links this library information with the fasta and quala files. The first line denotes the six columns of information in the expected\_lib\_stats file and must be entered exactly as follows:

FILE LIBRARY\_NAME PAIRED JUMPING SEP DEV

Each subsequent line in expected\_lib\_stats 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. Paired files should be listed after non-paired files, if any. (T/F)

For non-paired read files, the remaining three fields have no effect:

**JUMPING** – is this a jumping library? (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 library.

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

Boolean values for PAIRED and JUMPING are specified using T or F.

For example, for a paired read jumping library called 201FK with separation 3500 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 expected\_lib\_stats file describing two paired libraries:

FILE LIBRARY\_NAME PAIRED JUMPING SEP DEV

reads\_orig.13229.1.A.fasta 300AW T F 130 20

reads\_orig.13229.1.B.fasta 300AW T F 130 20

reads\_orig.201FK.5.A.fasta 201FK T T 3600 540

reads\_orig.201FK.5.B.fasta 201FK T T 3600 540

## 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 (currently unsupported).

## Example – *Staphylococcus aureus*

We provide two example data sets: *Escherichia coli* and *Staphylococcus aureus*. These are available from our web site:

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

### Unpacking the example data

This section and following sections on importing data and running the pipeline refer to the *S. aureus* read data example. Download sample-2.0.staph.tar.gz into the root directory (the directory into which allpaths-2.2.tar.gz was downloaded) and then unzip it as follows:

% gunzip sample-2.0.staph.tar.gz

% tar xvf sample-2.0.staph.tar

This will create a subdirectory called Staph.sample\_reads, containing the input read data and metadata, and another subdirectory called Staph.reference, containing the reference genome for *S. aureus*. Staph.sample\_reads is the SOURCE\_DIR for the *Staphylococcus aureus* data set.

### Examining the example data

This data set consists of *Staphylococcus aureus* reads generated using Illumina technology. There are four lanes of constant-length reads, paired with one another to create paired-production reads. Two of the lanes contain 35-bp reads in a ~200-bp fragment library, while the other two contain 26-bp reads in a ~4000-bp jumping library. The total raw physical coverage is 205x.

The Staph.sample\_reads directory should contain:

**reads\_orig.\*.fasta** - These are the raw, unfiltered reads in fasta format. In this dataset there are four fasta files, corresponding to the four Illumina flowcell lanes. There are two libraries, a short fragment library and a longer jumping library. The first read of each pair in each library is found in the files labeled A; the second read of each pair is in the corresponding file labeled B.

**reads\_orig.\*.quala** - The quala files contain the read quality scores – there is a quala file for each fasta file.

**expected\_lib\_stats** - This file defines the paired-read libraries. It contains one line for each of the fasta files as described above.

**ploidy** -This one-line file contains the ploidy of the input genome. *S. aureus* is a haploid genome, so the ploidy is 1.

The Staph.reference directory should contain:

**genome.fastb** - This is the reference genome, in fastb format.

# 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 RunAllPaths 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:

RunAllPaths

PRE=<user pre>

DATA\_SUBDIR=MyTestData

RUN=MyRun

REFERENCE\_NAME=Staph

TARGETS=import

SOURCE\_DIR=/reads/staphdata

K=20

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

<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 RunAllPaths.

## 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\_FASTB=<reference genome>.fastb

The reference genome should be in the fastb format. If you have a genome that is not in this format you can convert it using the 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 imported into the REFERENCE directory you can omit the REFERENCE\_FASTB argument when running RunAllPaths.

Instead of using the REFERENCE\_FASTB argument, you may simply create the REFERENCE directory and place the reference genome in it. The reference genome file must be in the fastb format and should be named:

<reference genome>.fastb

## Example – *Staphylococcus aureus*

This example explains how to import the example *S. aureus* data into the ALLPATHS pipeline directory structure in preparation for assembly without starting the assembly process. You must first have downloaded and unpacked the example data as described previously (see [Unpacking the example data](#_Unpacking_the_example)).

As mentioned before, the location of the pipeline directory is defined by the command line argument PRE. The directory defined by PRE should already exist.

### Importing example data: reads only

The RunAllPaths command should be executed in the directory that contains the Staph.sample\_reads directory; otherwise, specify an absolute path for SOURCE\_DIR.

RunAllPaths

PRE=<user pre>

DATA\_SUBDIR=MyTestData

RUN=MyRun

REFERENCE\_NAME=Staph

TARGETS=import

SOURCE\_DIR=Staph.sample\_reads

K=20

This command creates a new pipeline directory structure in the following location:

<PRE>/Staph/MyTestData/MyRun

### Importing example data: reads and a reference genome

To also import the reference genome (for evaluation purposes), use the following slightly modified RunAllPaths command. Again, it should be executed in the directory that contains both the Staph.sample\_reads and the Staph.reference directories; otherwise, specify absolute paths for both SOURCE\_DIR and REFERENCE\_FASTB.

RunAllPaths

PRE=<user pre>

DATA\_SUBDIR=MyTestData

RUN=MyRun

REFERENCE\_NAME=Staph

TARGETS=import

SOURCE\_DIR=Staph.sample\_reads

REFERENCE\_FASTB=Staph.reference/genome.fastb

K=20

# 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 RunAllPaths arguments commonly used to run the ALLPATHS pipeline. Complete descriptions of all arguments are provided in the [ALLPATHS Reference](#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 use:

EVALUATION=REFERENCE

**kmer size, K** - The kmer size, K, is restricted by the smallest read size in the read data to assemble. The value of K must be smaller than this size, and only certain values are supported. Suggested values of K for reads of 30 bases and larger are, 20 and 24.

**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 forms of parallelization that can be used to decrease runtime, given enough memory. To turn parallelization off, use:

MAXPAR=1  
PARALLEL\_BATCHES=1

You may wish to turn parallelization on by increasing these values up to the number of processors available, but only if you have sufficient memory. The memory requirements depend on the genome size and complexity and are difficult to determine in advance. See the [ALLPATHS Reference](#Reference) for more details.

## Example – *Staphylococcus aureus*

This example assembles the *S. aureus* data and runs a subset of the evaluation modules. It assumes that you have download and unpacked the example data (see [Unpacking the example data](#_Unpacking_the_example)) and imported the example *S. aureus* data into the ALLPATHS pipeline (see [Importing example data: reads and a reference genome](#_Importing_example_data:)). The command line argument PRE should be changed to reflect the actual value used in the import process.

RunAllPaths

PRE=<user pre>

DATA\_SUBDIR=MyTestData

RUN=MyRun

SUBDIR=MyAssembly

REFERENCE\_NAME=Staph

TARGETS=standard

K=20

EVALUATION=REFERENCE

MAXPAR=1

PARALLEL\_BATCHES=1

In response to this command, the pipeline runs all the core assembly modules plus a subset of the evaluation modules. The assembly output can be found in:

<PRE>/Staph/MyTestData/MyRun/ASSEMBLIES/MyAssembly

## Pipeline errors

The pipeline will stop on encountering an error. There are two types of error that can occur:

**rule consistency check error** - Before any modules are called, RunAllPaths 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 RunAllPaths command. The pipeline restarts at the point it previously failed.

# Examining the 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. Ignoring polymorphism the genome is linear, and ideally the assembly would be too. 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.

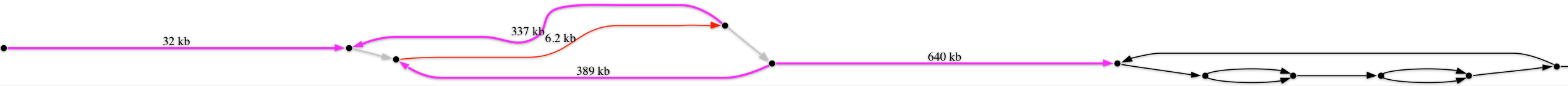
An edge can be considered to be equivalent to a contig.

A component can be considered to be equivalent to a scaffold.

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.

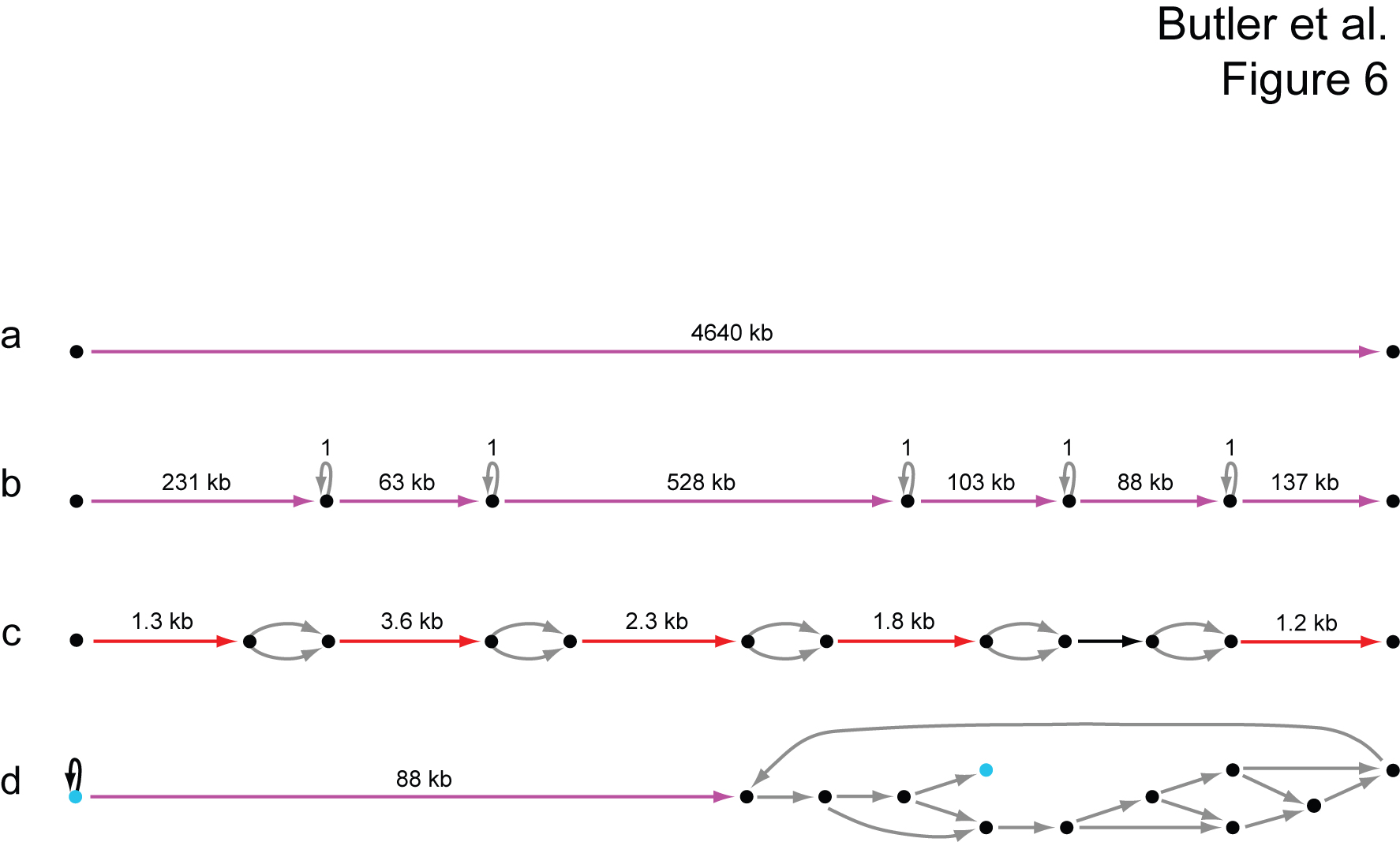
### Repeats

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.



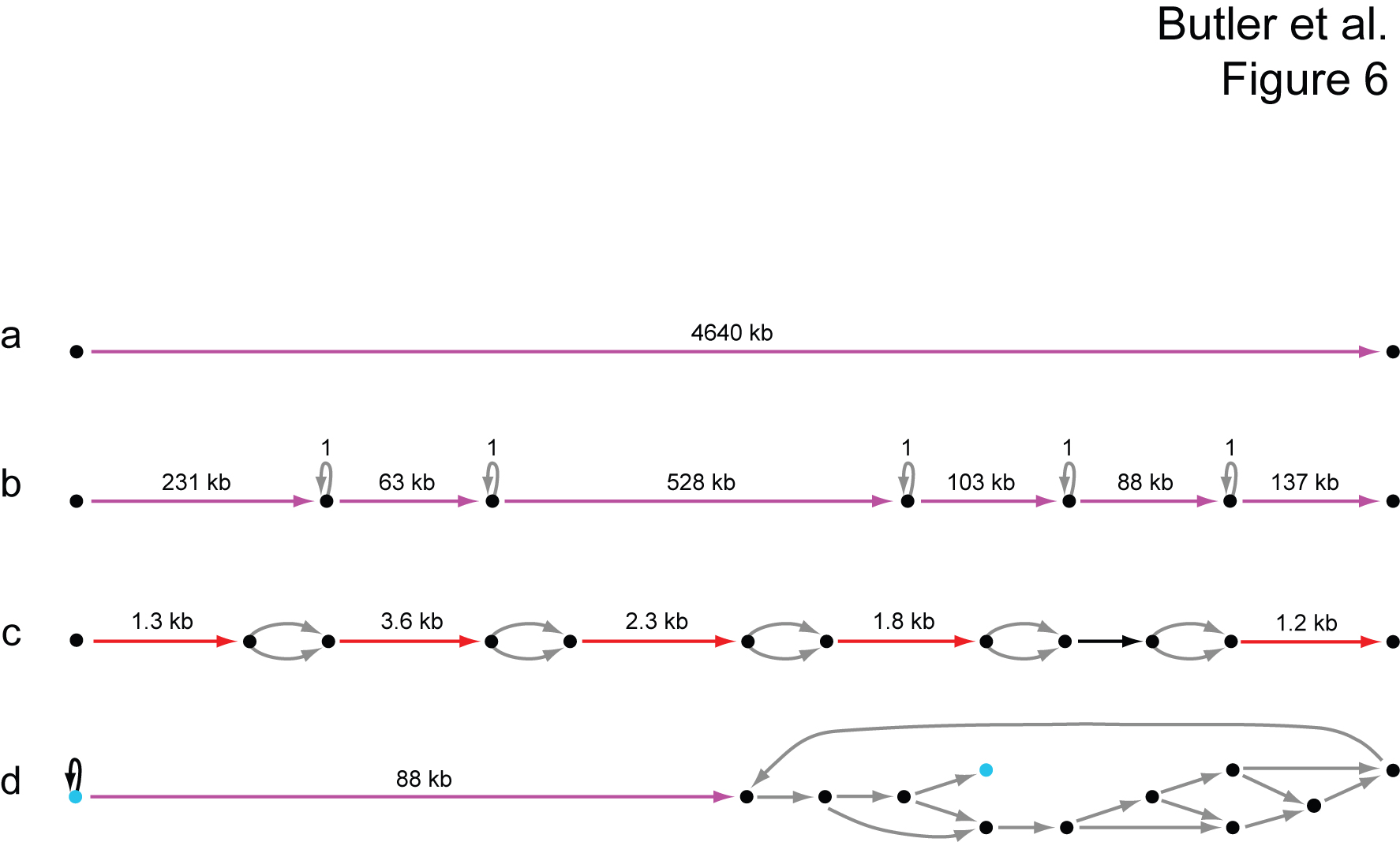
### Homopolymers

With short reads, long homopolymer runs can be difficult or impossible to resolve. Rather than assuming a value for the homopolymer length, they are represented as a loop of length 1 base. Where possible, however, a homopolymer run will be unrolled.



### SNPs and base errors

When the reads offer two seemingly equally possible alternatives for a base, we represent this as a small single base bubble. This situation can arise from SNPs or 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 files report.full and report.brief in the SUBDIR directory contain an evaluation of the final assembly. The key graph assembly statistics are given at the end of each file. 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.

## 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

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](#_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 contigs 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 in the SUBDIR directory and is called:

standard\_scaffold.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](#_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 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 RunAllPaths 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.

**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 RunAllPaths 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 and of the assembly itself.

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 perturbation of the assembly should be neutral, at least on average. Such ‘unsafe’ evaluations allow much more detailed information to be gathered about the assembly process and are extremely useful during development.

The evaluation mode used is controlled by:

EVALUATION=<evaluation mode>

There are three evaluation modes:

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

**REFERENCE** - safe evaluation against a reference genome.

**CHEAT** – 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. Typical values of K for reads of 30 bases and larger are 20 and 24.

The kmer size is passed to RunAllPaths using:

K=<kmer size>

## Error correction kmer size, ec\_ks

The read error correction process is also dependant on kmer size and the same restrictions apply. The kmer size used for error correction need not be, and often isn’t, the one used for assembly.

The error correction kmer size is passed to RunAllPaths using:

ec\_ks=<error correction kmer size>

For the default RunAllPaths parameters the error correction kmer size must be K+1. For example, if K=20 then ensure that ec\_ks=21.

## 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.

### 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 RunAllPaths 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.

### LocalizeReads parallelization

LocalizeReads is a key module in the assembly process. It can be run in parallel on multiple processors (and even on multiple machines - which is not described here). This form of parallelization is independent of the module parallelization described above. To parallelize LocalizeReads use:

MAX\_BATCHES=<n>

Again, for maximum performance and assuming ample memory, set this value to the number of processors available.

## 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.DumpFasta.out

The .cmd file contains the command used 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 DumpFasta.

References

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