Implementing Scalable Bioinformatic Workflows in Snakemake

Nathan S. Watson-Haigh

Sydney, Australia

12 December 2019

Licensing

This work is licensed under a Creative Commons Attribution 3.0 Unported License and the below text is a summary of the main terms of the full Legal Code (the full licence) available at http://creativecommons.org/licenses/by/3.0/legalcode.

You are free:

to copy, distribute, display, and perform the work

to make derivative works

to make commercial use of the work

Under the following conditions:

Attribution - You must give the original author credit.

With the understanding that:

Waiver - Any of the above conditions can be waived if you get permission from the copyright holder.

Public Domain - Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

Other Rights - In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice - For any reuse or distribution, you must make clear to others the licence terms of this work.

Contents

Licensing **3**

Contents **4**

Workshop Information **5**
- The Trainers . . . 6
- Providing Feedback . . . 7
- Document Structure . . . 7
- Provided Compute Infrastructure . . . 8
  - Connecting to the Cluster . . . 8
  - Monitoring Slurm Jobs . . . 9

Introduction to Snakemake **11**
- Key Learning Outcomes . . . 12
- Resources Required . . . 12
  - Tools Used . . . 12
- Useful Links . . . 12
- Snakemake is Like Making Sunday Dinner . . . 13
- Setting Up Your Environment . . . 13
  - Installing Snakemake . . . 13
- Your First Snakemake Workflow . . . 15
- Generalising Rules with Wildcards . . . 16
- Submitting Jobs to Slurm . . . 17
- Reimplementing a Workflow in Snakemake . . . 18
  - Getting the Code . . . 18
  - Getting the Data . . . 20
  - Implementation of BWA Indexing . . . 20
  - Implementing FastQC . . . 23
  - Implementing Trimmomatic . . . 27
  - Implementing BWA-MEM . . . 28
  - Adding New Samples . . . 32
  - Large Workflows . . . 35
- Troubleshooting . . . 37
  - Getting Going After a Disconnect . . . 37

Space for Personal Notes or Feedback **39**

Workshop Information

The Trainers

Dr. Nathan S. Watson-Haigh
Senior Bioinformatician
Bioinformatics Hub, University of Adelaide
nathan.watson-haigh@adelaide.edu.au

BY

Providing Feedback

While we endeavour to deliver a workshop with quality content and documentation in a venue conducive to an exciting, well run hands-on workshop with a bunch of knowledgeable and likable trainers, we know there are things we could do better.

Whilst we want to know what didn't quite hit the mark for you, what would be most helpful and least depressing, would be for you to provide ways to improve the workshop. i.e. constructive feedback. After all, if we knew something wasn't going to work, we wouldn't have done it or put it into the workshop in the first place!

Clearly, we also want to know what we did well! This gives us that "feel good" factor which will see us through those long days and nights in the lead up to such hands-on workshops!

With that in mind, we'll provide a some high tech mechanism through which you can provide anonymous feedback during the workshop:

1. Some empty ruled pages at the back of this handout. Use them for your own personal notes or for writing specific comments/feedback about the workshop as it progresses.

Document Structure

We have provided you with an electronic copy of the workshop's hands-on tutorial documents. We have done this for two reasons: 1) you will have something to take away with you at the end of the workshop, and 2) you can save time (mis)typing commands on the command line by using copy-and-paste.

We advise you to use Acrobat Reader to view the PDF. This is because it properly supports some features we have implemented to ensure that copy-and-paste of commands works as expected. This includes the appropriate copy-and-paste of special characters like tilde and hyphens as well as skipping line numbers for easy copy-and-past of whole code blocks.

> While you could fly through the hands-on sessions doing copy-and-paste, you will learn more if you use the time saved from not having to type all those commands, to understand what each command is doing!

The commands to enter at a terminal look something like this:

```
snakemake --profile profiles/slurm --use-singularity --printshellcmds all
```

The following icons are used in the margin, throughout the documentation to help you navigate around the document more easily:

- ℹ Important

- 📄 For reference
- 👣 Follow these steps
- Q Questions to answer
- ❗ Warning - STOP and read
- ✸ Bonus exercise for fast learners
- ✸ Advanced exercise for super-fast learners

Provided Compute Infrastructure

For the purposes of this training, we are providing you with access to an auto-scaling virtual cluster running on top of AWS. This has been provisioned through Ronin (`https://ronin.cloud`). The specification of this cluster is as follows:

Head Node
`m5ad.4xlarge` (16 vCPU, 64G RAM and 2x300 SSD)

Compute Nodes
`t2.medium` (2 vCPU, 4G RAM and EBS), min nodes: 20 max nodes: 40, spot price enabled

Shared Storage
`1000G`

Connecting to the Cluster

👣 First up, lets connect to the head node of the HPC cluster using `ssh`.

See your local facilitator for connection details. You will have one user account per person.

Upon connecting, feel free to use `screen` or `tmux` if you are familiar with either of those tools.

Monitoring Slurm Jobs

You can monitor all jobs in the slurm squeue, or just your own job(s) using the slurm command `squeue`:

```
# All jobs in the queue
squeue

# Just your own jobs
squeue --user ${USER}
```

For convienience we have provided you with the `sq` function which produces nicer output than the default `squeue`:

```
# All jobs in the queue
sq

# Just your own jobs
sq --user ${USER}
```

Introduction to Snakemake

Primary Author(s):
Nathan S. Watson-Haigh nathan.watson-haigh@adelaide.edu.au

Contributor(s):

Key Learning Outcomes

After completing this module the trainee should be able to:

- Install Snakemake in a conda environment
- Execute a Snakemake workflow
- Use the provided "profile" to execute jobs on a compute cluster
- Write simple Snakemake rules capable of generating some output(s) by executing some code which operates on some input(s)

Resources Required

For the purpose of this training you need access to:

- A compute cluster with the `module` command available to you for loading software
- Singularity (`https://sylabs.io/singularity/`) - available as a module on the above cluster
- Conda(`https://www.anaconda.com/distribution/`) - available as a module on the above cluster

Tools Used

Snakemake
`https://snakemake.readthedocs.io`

Graphviz
`https://www.graphviz.org`

Useful Links

Slurm Documentation
`https://slurm.schedmd.com/documentation.html`

Snakemake is Like Making Sunday Dinner

The way Snakemake approaches running workflows is a bit like the way you prepare for dinner/tea. First, you think about what you want for dinner/tea, say a Sunday roast. To create the Sunday roast, you need meat and vegetables, all of which need preparing (e.g. peeling and seasoning). If you haven't got an ingredient, you go out to the shop and buy it.

With Snakemake, you decide what output files you want to create, say some BAM files. To create the BAM files, you need FASTQ files and a reference genome, all of which need preparing (e.g. quality/adapter trimming and indexing). If you haven't got those files, you need to create them.

Setting Up Your Environment

For the purpose of the workshop we will be working on the head node of an HPC cluster running slurm (`https://slurm.schedmd.com/documentation.html`). This is the most likely infrastructure that fellow bioinformaticians already find themselves using on a regular basis. We also assume that the cluster provides the `module` command for you to load software and the modules `Anaconda3` and `Singularity` are available to use.

The execution of the Snakemake workflow will actually take place on the cluster head node with jobs being submitted to Slurm for queing and processing. From the head node, Snakemake will monitor the submitted jobs for their completion status and submit new jobs as dependent jobs complete sucessfully.

Installing Snakemake

The recommended installation route for Snakemake is through a conda environemnt (`https://snakemake.readthedocs.io/en/stable/getting_started/installation.html`). As such, you need Anaconda3, usually available to you on your cluster via the module system.

```
# We use a specific version for reproducibility reasons
# Find the latest version: https://anaconda.org/search?q=snakemake
SNAKEMAKE_VERSION="5.8.1"

# Load miniconda
module load \
  miniconda3-4.6.14-gcc-5.4.0-kkzv7zk

#####
# One-time commands
#####
# Integrate conda into bash
```

```
conda init bash
. ${HOME}/.bashrc

# Change the default location into which conda saves packages
# and environments
conda config --prepend pkgs_dirs /shared/${USER}/.conda/pkgs
conda config --prepend envs_dirs /shared/${USER}/.conda/envs

# Change the default channels used for finding software and
# resolving dependencies
conda config --add channels defaults
conda config --add channels bioconda
conda config --add channels conda-forge
#####
```

Do NOT run the following command! This is provided for future reference so you know how to Install Snakemake on another system. We have already run this for you and doing so again will cause delays.

```
## Install snakemake using conda
## This might take 5-10mins
#SNAKEMAKE_VERSION="5.8.1"
#
#conda create \
# --name snakemake \
# snakemake=${SNAKEMAKE_VERSION}
```

Snakemake installation is now complete.

All you need to do is to activate the environment which will make `snakemake` available to you on the command line:

```
# Activate the conda environment
conda activate snakemake
```

Integrate Snakemake autocompletion into bash:

```
complete -o bashdefault -C snakemake-bash-completion snakemake
```

Test if Snakemake is actually working:

```
snakemake --version
```

While waiting for others to catch up, why not have a look into how you would go about updating `Snakemake` within this conda environment if there is a new version available.

Your First Snakemake Workflow

To get started with Snakemake, all you need to do is create a `Snakefile` (note the capitalisation) containing a rule which specifies how to create an output file.

Setup a working directory for this task:

```
mkdir --parents /shared/${USER}/snakemake/hello
cd /shared/${USER}/snakemake/hello
```

Create a file called `Snakefile` and add the following content:

Like Python, indentation in a `Snakefile` matters! For consistency, I'd recommend tab indentations.

```
rule hello_world:
  output:
    "Hello/World.txt",
  shell:
    """
    echo "Hello, World!" > {output}
    """
```

You can now run this workflow in one of 3 ways:

```
# Request Snakemake to generate the specific output file
# "Hello/World.txt"
snakemake Hello/World.txt

# Request Snakemake to execute the specific rule "hello_world"
snakemake hello_world

# Request Snakemake to execute the first rule in the Snakefile
snakemake
```

What happens if you run one of the above commands two or more times? Why?

Generalising Rules with Wildcards

The original `hello_world` rule wasn't very flexible. We couldn't say "Hello, World!" in Spanish, Polish or French. However, we can generalise the rule using "wildcards" (https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#wildcards):

```
rule hello_world:
  output:
    "{cheer}/{world}.txt",
  shell:
    """
    echo "{wildcards.cheer}, {wildcards.world}!" > {output}
    """
```

Now we can use whatever language we want:

```
# In English - nothing should be done since the file
# already exists
snakemake Hello/World.txt

# In Polish
snakemake Czesc/Swiat.txt

# In French and Spanish at the same time
snakemake Monde/Monde.txt Ciao/Mondo.txt
```

Take a look at the files created and their contents:

```
tree ./
cat */*.txt
```

Submitting Jobs to Slurm

By default, Snakemake executes jobs on the same computer on which it is running. For Snakemake to be able to submit jobs to a cluster resource management/queuing system, such as Slurm, we can use a "profile". A profile is really just a way to group scripts required for Snakemake to "talk" to a cluster backend for things such as job submission and monitoring.

To avoid having to delve into implementing our own "profile" for use with our Slurm cluster, we have created a Slurm profile ready for you to use. So lets grab it:

```
# Ensure a working directory exists and move into it
mkdir --parents /shared/${USER}/snakemake/tutorial
cd /shared/${USER}/snakemake/tutorial

# Clone the Snakemake template repository from GitHub
git clone https://github.com/UofABioinformaticsHub/snakemake_template ./

# Checkout the "hello" branch
git checkout hello
```

The `Snakefile` in this branch of the repository is the same "Hello, World!" example you created above, with wildcards. Lets see how we use the provided "profile" to get Snakemake to submit jobs to Slurm:

```
snakemake \
  --profile profiles/slurm \
  Hello/World.txt Czesc/Swiat.txt Monde/Monde.txt Ciao/Mondo.txt

# See what files we have
tree
```

If the `STDOUT` and `STDERR` of the command(s) in a rule are not explicitly sent to a file, then they will end up in Slurm's log file for a particular job which is normally something like `slurm-<job_id>.out`. This isn't that helpful for debugging purposes, so the provided profile changes this to `logs/<rule_name>/<wildcards>.out` e.g. `logs/hello_world/cheer=Ciao,world=Mondo.out`. See:

```
tree logs/
```

We've finished with this simple "Hello, World!" example, so cleanup after yourself:

```
snakemake \
  --delete-all-output \
  Hello/World.txt Czesc/Swiat.txt Monde/Monde.txt Ciao/Mondo.txt

# See what files we have
tree
```

There are examples of community developed profiles, available for other compute back-ends, see: https://github.com/Snakemake-Profiles/doc.

Reimplementing a Workflow in Snakemake

A bioinformatic "pipeline" is commonly a single, monolithic bash script which performs all the tasks which need to be performed. For example, someone might have written a script for performing the following tasks:

- Run FastQC across all the raw read files
- Adapter, quality, and read length filtering using Trimmomatic
- Index the reference FASTA file
- Perform a `bwa-mem` read alignment

We will walk you through the steps of reimplementing the first few steps of the above script into a Snakemake workflow. Along the way, we will introduce the core concepts of Snakemake and then ask you to reimplement the `bwa-mem` step yourselves. For those working quickly, you will have the opportunity to reimplement the `multiqc` step. This will provide you with a foundation for you to be able to convert your own workflows into Snakemake rules and begin reaping the rewards of being able to run your analyses in Snakemake.

Getting the Code

We have the monolithic script available for you on the `walkthrough` branch, switch to it and have a look:

```
# Checkout the walkthrough branch
git checkout walkthrough

# Have a look at the contents of the
# monolithic analysis script
less analysis.sh
```

While the author of such a script should be commended for their efforts in documenting their analysis using a script, it has several significant limitations:

Not parallelised
Loops over input files, executing independant commands in sequential order

Resources over-specified
The compute resources needed by the script are dictated by the command(s) with the largest requirement(s)

Not idempotent
Significant programming logic is needed to wrap around commands to detect failures and only execute parts of the analysis which failed in earlier attempts

How might you modify the above script to:

- Add new samples
- Rerun the script if you find one of the files generated is corrupt
- Include readgroup information at the `bwa-mem` step (`-b` argumanet)

How would you avoid rerunning commands which take a long time and already completed sucessfully on a previous run e.g. the reference index, `bwa-mem` etc?

CC BY

Getting the Data

We've provided you with some real whole genome shotgun (WGS) sequencing data from wheat, together with a small chunk of the wheat genome. The data set is small enough so each step in the analysis will take less than a couple of minutes to run. We have a copy of this data available locally, so you just need to move it to the right location.

```
# Get a copy of the data
mv \
  /shared/${USER}/walkthrough_data/{raw_reads,references,misc} \
  ./

# Have a look at the files we provided
tree raw_reads references misc
```

Implementation of BWA Indexing

We have provided an out-of-the-box `Snakefile` capable of indexing the provided reference sequence, together with comments. Lets have a bit of a play before we get around to actually running the workflow.

```
less Snakefile

# These commands have the same effect
snakemake --dry-run all
snakemake --dry-run
```

Why doesn't this work:

```
snakemake --dry-run bwa_index
```

Do we have to specify all 5 of the BWA index files in the `all` pseudo-rule?

Why do we have to specify all 5 index files as `output` of the `bwa_index` rule?

The effect `--dry-run` is to simply show you what "would" be run, without actually running it. It's useful to ensure you're going to get what you though, especially as your worflows get larger and more interconnected.

Another useful feature is to generate a directed acyclic graph (DAG) of the jobs which comprise the workflow and how they are linked together. Although for this workflow is not yet that impressive, but we'll have a look at how we generate the DAG:

```
snakemake \
  --dag \
| dot -Tpdf \
> dag1.pdf
```

If you want to view the `dag1.pdf` file yourself, you will need to copy the file to your local computer using `scp`, Filezilla (`https://filezilla-project.org`), WinSCP (`https://winscp.net`) or similar. The DAG generated should look like that shown in Figure 1.

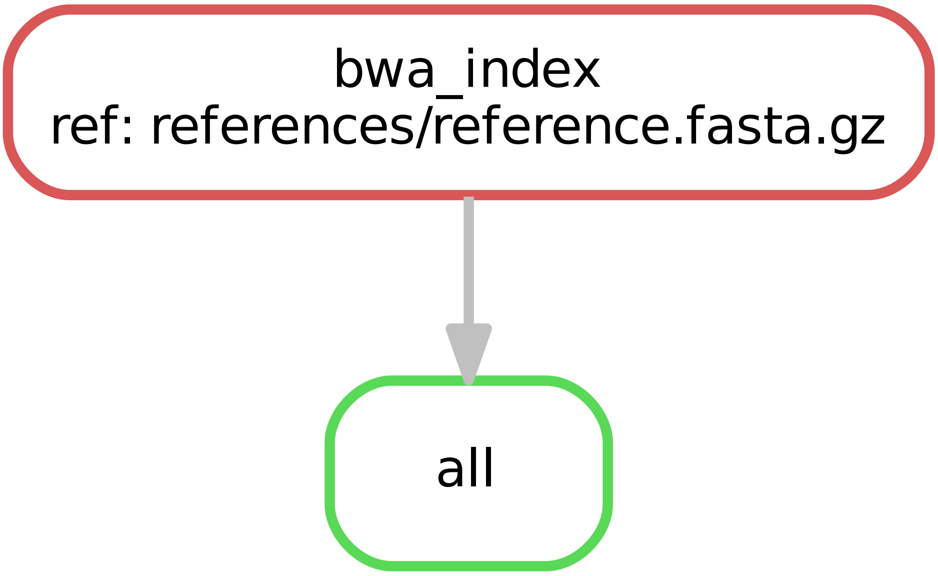
bwa_index
ref: references/reference.fasta.gz
all

Figure 1: DAG of jobs showing `bwa_index` job dependant on the `all` pseudo-rule.

Implementing FastQC

Lets add a rule for performing FastQC on our input files. Looking at the `analysis.sh` file we see the following command is executed for each `SAMPLE` while interating over the `SAMPLES` list:

```
fastqc --threads 1 \
  raw_reads/${SAMPLE}_R1.fastq.gz \
  raw_reads/${SAMPLE}_R2.fastq.gz
```

This command can be converted into a Snakemake `rule` by adding the following rule to the `Snakefile`:

```
rule fastqc:
  input:
    r1 = "raw_reads/{SAMPLE}_R1.fastq.gz",
    r2 = "raw_reads/{SAMPLE}_R2.fastq.gz",
  output:
    zip = [
      "raw_reads/{SAMPLE}_R1_fastqc.zip",
      "raw_reads/{SAMPLE}_R2_fastqc.zip",
    ],
    html = [
      "raw_reads/{SAMPLE}_R1_fastqc.html",
      "raw_reads/{SAMPLE}_R2_fastqc.html",
    ],
  shell:
    """
    fastqc --threads 1 {input.r1} {input.r2}
    """
```

Improving FastQC Parallelisation

There are a few improvements we can make to the `fastqc` rule:

- We don't need to process both the R1 and R2 read files with the same FastQC job. We can operate on one read file at a time. By doing this, Snakemake will be able to execute the FastQC job for each file in parallel.
- We want a convienient way of generating FastQC outputs for ALL samples without typing them all at the command line.

Change the `fastqc` rule to the following:

```
rule fastqc:
  input:
    "raw_reads/{prefix}.fastq.gz",
  output:
    zip = "raw_reads/{prefix}_fastqc.zip",
    html = "raw_reads/{prefix}_fastqc.html",
  shell:
    """
    fastqc --threads {threads} {input}
    """
```

Now run the same Snakemake dry-run command as before:

```
snakemake --dry-run raw_reads/ACBarrie_R1_fastqc.html \
   raw_reads/ACBarrie_R2_fastqc.html
```

Rule-Specific Resource Specification

The profile provided in this repository specifies default values for Slurm resources. However, it is possible to provide rule-specific overrides so that jobs can make use of more time, memory, cores etc. The ‘cluster-configs/default.yaml‘ file is where these settings can be modified.

```
less cluster-configs/default.yaml
```

Pseudo-Rules

We can use “pseudo-rules” to define a list of target filenames for creation when we use the rule name as a “target”. Pseudo-rules consist of just an `input` directive:

```
rule all:
  input:
    "raw_reads/ACBarrie_R1_fastqc.html",
    "raw_reads/ACBarrie_R2_fastqc.html",
```

By convention, the first pseudo-rule in the `Snakefile` is called `all` and specifies all the output filenames of the workflow. This now means we can execute a workflow in any of the following ways:

```
# Not specifying a target will result in Snakemake executing the
# first rule in the Snakefile ("all" in this case)
snakemake --dry-run

# Explicityly request the "all" rule
snakemake --dry-run all
```

When workflows get larger and the lists of filenames get bigger, specifying long lists of filenames in pseudo-rules can start to feel cumbersome. Since Snakemake syntax is an extension of Python, we can start to use some Python data structures and functions to help.

Add the following Python list of sample names (with most commented out for now) at the top of the file:

```
SAMPLES = [
  "ACBarrie",
  "Alsen",
# "Baxter",
# "Chara",
# "Drysdale",
# "Excalibur",
# "Gladius",
# "H45",
# "Kukri",
# "Pastor",
# "RAC875",
# "Volcanii",
# "Westonia",
# "Wyalkatchem",
# "Xiaoyan",
# "Yitpi",
]
```

Add FastQC output files for all samples in the `SAMPLES` list, as well as both read files, as new targets to the existing `all` rule. We'll make use of the `expand()` function to simplify things somewhat. The resulting `all` pseudo-rule should look like this:

```
rule all:
  input:
    expand("references/reference.fasta.gz.{ext}",
      ext=['amb', 'ann', 'bwt', 'pac', 'sa']
    ),
    expand("raw_reads/{SAMPLE}_{read}_fastqc.html",
      SAMPLE=SAMPLES,
      read=['R1', 'R2']
    ),
```

Lets take a look at what jobs would be run if we run the whole workflow. Remember, the following commands are equivilent:

```
# Explicitly run the "all" pseudo-rule
snakemake --dry-run all

# Run the first rule in the Snakefile. This should be the
# "all" rule by convention
snakemake --dry-run
```

Lets look at the DAG for the workflow:

```
snakemake \
  --dag \
| dot -Tpdf \
> dag2.pdf
```

If you want to view the `dag2.pdf` file yourself, you will need to copy the file to your local computer using `scp`, Filezilla (`https://filezilla-project.org`), WinSCP (`https://winscp.net`) or similar. The DAG generated should look like that shown in Figure 2.

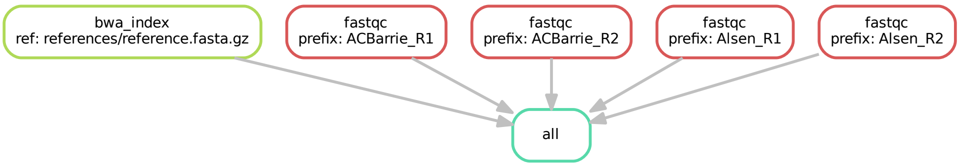
bwa_index
ref: references/reference.fasta.gz
fastqc
prefix: ACBarrie_R1
fastqc
prefix: ACBarrie_R2
fastqc
prefix: Alsen_R1
fastqc
prefix: Alsen_R2
all

Figure 2: DAG of jobs showing `bwa_index` and several `fastqc` job dependant on the `all` pseudo-rule.

Executing the Workflow on Slurm

Up until now, we've just been playing around with `--dry-run`, so lets move on and start executing the workflow on the Slurm cluster! Remember, we need to use the Slurm profile we've provided you with so Snakemake knows how to communicate with Slurm.

In addition, we're also going to execute the jobs within a singularity container which has the tools we need already installed inside it.

```
# Make sure Singularity is available
module load \
  singularity-3.2.1-gcc-5.4.0-tn5ndnb

# Execute the workflow
snakemake \
  --profile profiles/slurm \
  --use-singularity
```

Depending on how quickly everyone else is in executing their workflows, you might get to see your jobs in the Slurm queue by executing this in another window:

```
sq
```

Implementing Trimmomatic

We've gone through implementing the FastQC command as a Snakemake rule and demonstrated the core concepts of Snakemake along the way. We'll go through implementing one more command as a Snakemake rule before you go off and try one on your own!

If you compare the `trimmomatic` command in `analysis.sh` to the rule provided below, you will see that we have simply pulled out all references to input or output files into the `input` or `output` directives. Where we had used the bash variable `${SAMPLE}` in the filenames we are now using Snakemake "wildcards". It is almost the same syntax - just notice the absence of the $ but the curly braces are retained. The biggest changes seen are in the `shell` directive, where we now have to refer to the input and output files via `{input.r1}`, `{output.r1_unpaired}` etc.

```
rule trimmomatic:
  input:
    r1 = "raw_reads/{SAMPLE}_R1.fastq.gz",
    r2 = "raw_reads/{SAMPLE}_R2.fastq.gz",
    adapters = "misc/trimmomatic_adapters/TruSeq3-PE.fa"
  output:
    r1 = "qc_reads/{SAMPLE}_R1.fastq.gz",
    r2 = "qc_reads/{SAMPLE}_R2.fastq.gz",
    r1_unpaired = "qc_reads/{SAMPLE}_R1.unpaired.fastq.gz",
    r2_unpaired = "qc_reads/{SAMPLE}_R2.unpaired.fastq.gz",
  shell:
    """
    trimmomatic PE \
      -threads {threads} \
      {input.r1} {input.r2} \
      {output.r1} {output.r1_unpaired} \
      {output.r2} {output.r2_unpaired} \
      ILLUMINACLIP:{input.adapters}:2:30:10:3:true \
      LEADING:2 \
      TRAILING:2 \
      SLIDINGWINDOW:4:15 \
      MINLEN:36
    """
```

Next, we need to add the trimmomatic output files to our `all` pseudo-rule to make it convenient to create them. Your `all` pseudo-rule should look like this:

```
rule all:
  input:
    expand("references/reference.fasta.gz.{ext}",
      ext=['amb', 'ann', 'bwt', 'pac', 'sa']
    ),
    expand("raw_reads/{SAMPLE}_{read}_fastqc.html",
      SAMPLE=SAMPLES,
      read=['R1', 'R2']
    ),
    expand("qc_reads/{SAMPLE}_{read}.fastq.gz",
      SAMPLE=SAMPLES,
      read=['R1', 'R2']
    ),
```

We won’t run the workflow in a piecemeal fashion, we’ll save all our jobs for running a bit later. So for now, lets look at the dry-run again:

```
# Run the first rule in the Snakefile. This should be the
# "all" rule by convention
snakemake --dry-run
```

Implementing BWA-MEM

Now is your opportunity to put into practice what you have learnt from the above walk-thoughs of implementing FastQC and Trimmomatic commands. Your task is to implement the `bwa mem` command into a Snakemake rule.

Here are some questions to get you thinking as you try to implement this rule:

What input read files are required for the command/rule?

Does the command/rule need the FASTA reference file or the index files as input?

The BWA-MEM command uses a “prefix” to the FASTA index files, not the index filenames themselves. How will you specify this in the `shell` directive?

If you hard-coded it, what would the rule look like?

Hard-coding the path is simple, but not ideal. What if you changed the name of the reference file or wanted to use the rule with a different project? You would have to modify the paths in multiple places, once in the `input` directive and once in the `shell` directive.

Move the hard-coded path out of the `shell` directive into the `params` directive (see: https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#non-file-parameters-for-rules).

Now you are making use of the `params` directive, this opens up the possibility of using some Python to do some string manipulations on the paths defined in the `input` directive. In particular, we can use a Python lambda function in the `params` directive (see: `https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#non-file-parameters-for-rules`).

How might you change a hard-coded path in the `params` directive to use a lambda function which manipulates the index file path(s) set in the `input` directive to define the prefix? Hint: take the path of one of the index files and remove the last few characters corresponding to the last file extension. You will probably need to do some reading of the Snakemake and/or Python documentation.

Now add the BAM files corresponding to all the samples in the `SAMPLES` list, that you want the BWA-MEM rule to produce, to the `all` pseudo-rule.

Don't worry if you didn't complete the above implementation of the BWA-MEM command, we have a git repository branch with the rules we developed. Get it, execute the workflow and generate the DAG:

```
# Checkout the branch with our implemetation of these rules
git checkout final

# Execute the workflow
snakemake \
  --profile profiles/slurm \
  --use-singularity

# Generate a DAG
snakemake \
  --dag \
| dot -Tpdf \
> dag3.pdf
```

If you want to view the `dag3.pdf` file yourself, you will need to copy the file to your local computer using `scp`, Filezilla (`https://filezilla-project.org`), WinSCP (`https://winscp.net`) or similar. The DAG generated should look like that shown in Figure 3.

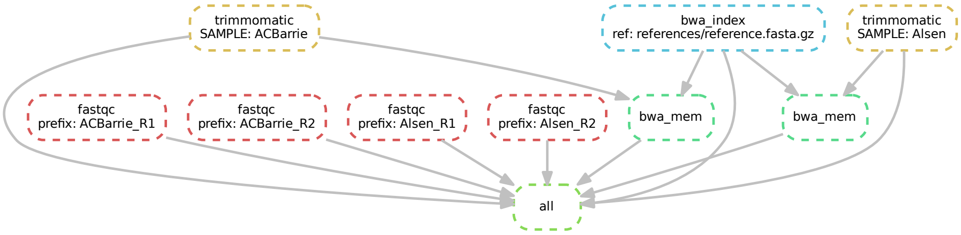
trimmomatic
SAMPLE: ACBarrie
bwa_index
ref: references/reference.fasta.gz
trimmomatic
SAMPLE: Alsen
fastqc
prefix: ACBarrie_R1
fastqc
prefix: ACBarrie_R2
fastqc
prefix: Alsen_R1
fastqc
prefix: Alsen_R2
bwa_mem
bwa_mem
all

Figure 3: DAG of jobs showing the dependencies which exist in our final implementation of the `analysis.sh` workflow.

Adding New Samples

Our `SAMPLES` list contains a lot of samples which are currently commented out. Lets uncomment them and have a look at some other features of Snakemake:

```
# Manually uncomment the samples or use this sed command
sed -i 's/^# \{2\}"/ "/' Snakefile
```

With so many more samples, the DAG becomes next to useless with dot's default layout algorithm:

```
# Generate a DAG
snakemake \
  --dag \
| dot -Tpdf \
> dag4.pdf
```

If you want to view the `dag4.pdf` file yourself, you will need to copy the file to your local computer using `scp`, Filezilla (`https://filezilla-project.org`), WinSCP (`https://winscp.net`) or similar. The DAG generated should look like that shown in Figure 4.

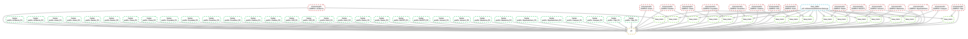

Figure 4: DAG of jobs for the whole workflow consisting of 16 samples.

Instead, the "rulegraph" might provide a better view of the workflow. Unlike the DAG, that shows the individual jobs and their dependencies, the rulegraph shows only the rules and and their dependencies so provides a simplified view of the workflow:

```
# Generate a rulegraph
snakemake \
  --rulegraph \
| dot -Tpdf \
> rulegraph.pdf
```

If you want to view the `rulegraph.pdf` file yourself, you will need to copy the file to your local computer using `scp`, Filezilla (`https://filezilla-project.org`), WinSCP (`https://winscp.net`) or similar. The rulegraph generated should look like that shown in Figure 5.

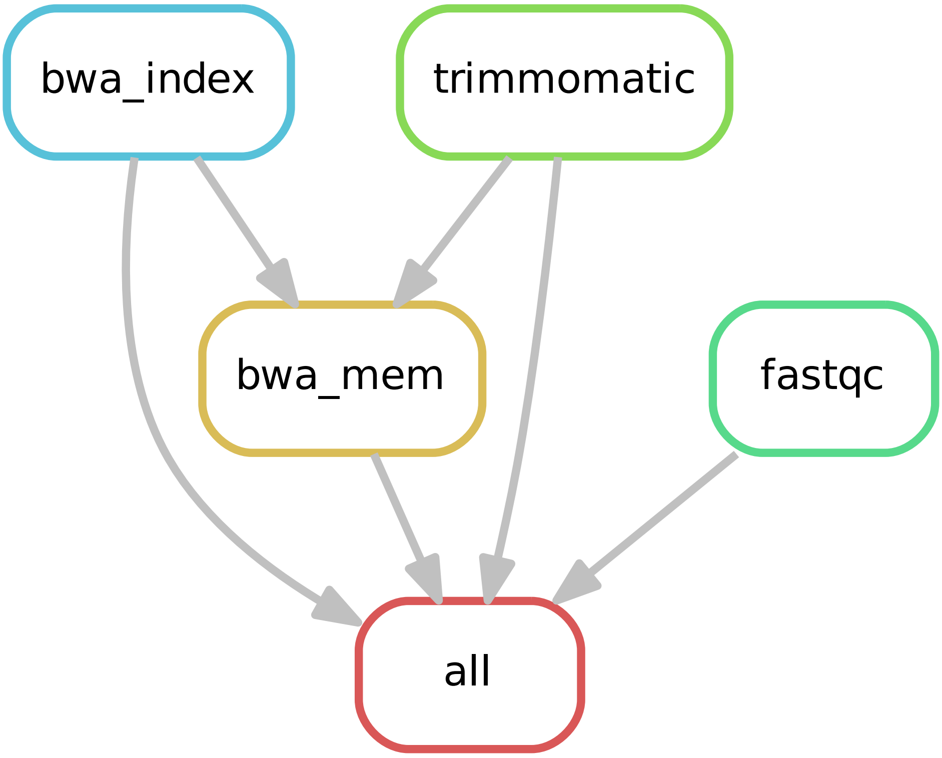
bwa_index
trimmomatic
bwa_mem
fastqc
all

Figure 5: Rulegraph for the whole workflow.

Execute the rest of the workflow:

```
# Execute the workflow
snakemake \
  --profile profiles/slurm \
  --use-singularity
```

In a different terminal, have a look at your jobs in the queue:

```
sq
```

How many `fastqc` and total number of jobs are run as part of the whole workflow? Hint: try using `--forceall` in combination with `--dry-run`.

Using the Snakemake help, which command line argument can be used to get Snakemake to print the shell commands associated with each job during a `dry-run`?

Using the Snakemake help, which command line argument can be used to delete all the outputs associated with a given "target"?

Large Workflows

A workflow I have developed is capable of mapping RNA-Seq, WGS, exome capture and Iso-Seq data to the wheat genome. In addition, downstream variant analysis and creation of various tracks for JBrowse (e.g. BigWig) are possible. Much of the workflow involves broking the tasks down so each of the chromosomes is proccessed independantly, and in parallel by the workflow. All up, there are more than 10,000 individual jobs representing 14 rules. DAG's of this size are best opened and layed out by Gephi using the `ForceAtlas 2` algorithm:

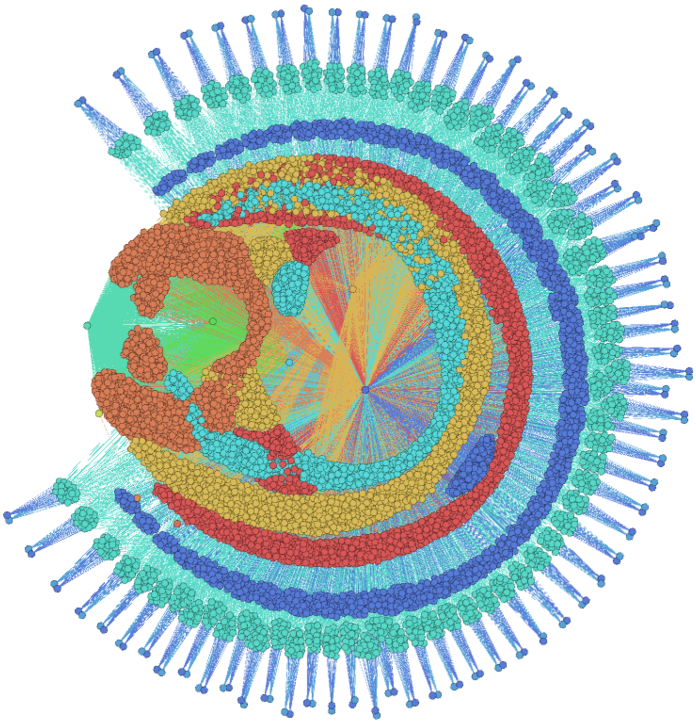

Figure 6: Large DAG, from the DAWN workflow, consisting of 10,587 nodes/jobs and 32,353 edges/dependancies. Layed out using `ForceAtlas 2` algorithm of Gephi

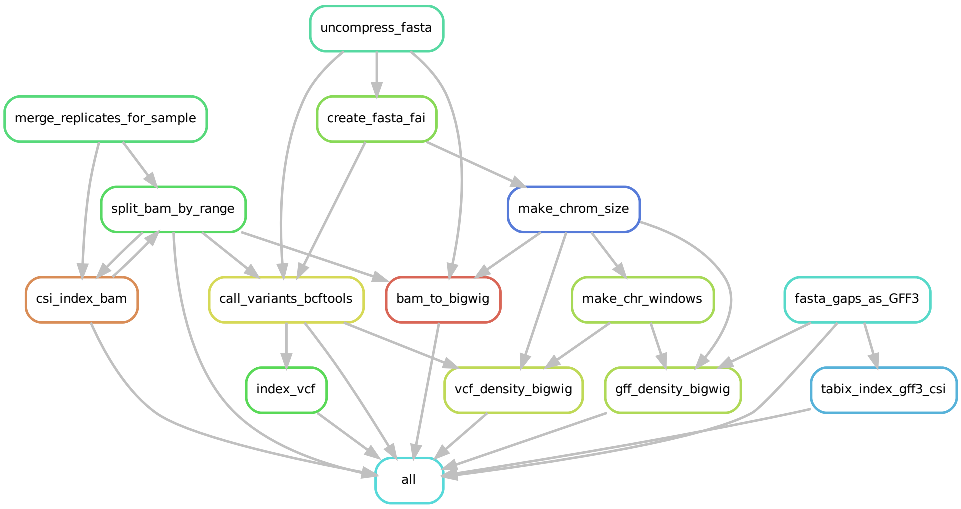
uncompress_fasta
merge_replicates_for_sample
create_fasta_fai
split_bam_by_range
make_chrom_size
csi_index_bam
call_variants_bcftools
bam_to_bigwig
make_chr_windows
fasta_gaps_as_GFF3
index_vcf
vcf_density_bigwig
gff_density_bigwig
tabix_index_gff3_csi
all

Figure 7: Rulegraph, from the DAWN workflow.

CC BY

Troubleshooting

Getting Going After a Disconnect

If you find that your connection to the server has been dropped, you can get yourself going again using this convienient block of commands:

```
# Load the required software modules
module load \
  miniconda3-4.6.14-gcc-5.4.0-kkzv7zk \
  singularity-3.2.1-gcc-5.4.0-tn5ndnb

# Activate the snakemake conda environment and
# integrate shell autocompletion into bash
conda activate snakemake
complete -o bashdefault -C snakemake-bash-completion snakemake

# Move to the correct directory location, most
# likely here:
cd /shared/${USER}/snakemake/tutorial
```

Space for Personal Notes or Feedback