CMCC - Notes on Snakemake

Fabio Viola

Contents

1 Introduction **2**

2 Installation **2**

3 How it works **2**
- 3.1 Defining rules . . . 2
- 3.2 Execution of rules . . . 2
- 3.3 Generalization of the mapping rule . . . 3
- 3.4 Aggregation . . . 3
- 3.5 Scripts . . . 4
- 3.6 Best practices . . . 4

4 Main Commands **4**

5 References **5**

1 Introduction

Snakemake is a **workflow management system**, a tool to create reproducible and scalable data analysis tasks. Workflows are described via a human readable, python-based language. They can be seamlessly scaled to server, cluster, grid and cloud environments, without the need to modify the workflow definition. Finally, Snakemake workflows can entail a description of required software, which will be automatically deployed to any execution environment.
Snakemake works through a file called `Snakefile` containing a set of rules specifying the way input files are manipulated to produce output files.

2 Installation

The recommended installation method is through conda:

```
$ conda install -c conda-forge -c bioconda snakemake
```

3 How it works

3.1 Defining rules

A Snakemake workflow is defined by specifying rules in a **Snakefile**. **Rules** decompose the workflow into small steps. Every rule specifies how to create sets of **output** files from sets of **input** files. Snakemake automatically determines the dependencies between the rules by matching file names. The Snakemake language extends the Python language, adding syntactic structures for rule definition and additional controls.

An example rule is the following:

```
rule bwa_map:
    input:
        "data/genome.fa",
        "data/samples/A.fastq"
    output:
        "mapped_reads/A.bam"
    shell:
        "bwa mem {input} | samtools view -Sb - > {output}"
```

Here, we can identify a **name** for the rule (here `bwa_map`) and a number of **directives**, here `input`, `output` and `shell`. The latter is followed by a Python string containing the **shell command** to execute. In the shell command string, we can refer to elements of the rule via braces notation (similar to the Python format function). Here, we refer to the output file by specifying `{output}` and to the input files by specifying `{input}`. What happens when (as in this case) there are multiple input files? Snakemake will concatenate them separated by a whitespace in the same command. Note that in the documentation, outputs are also referred to as **targets**.

3.2 Execution of rules

Snakemake applies the rules given in the Snakefile in a top-down way. The application of a rule to generate a set of output files is called **job**. For each input file of a job, Snakemake determines rules that can be applied to generate it. This yields a **directed acyclic graph (DAG)** of jobs where the edges represent dependencies.

The execution of this very simple example is invoked through:

```
$ snakemake mapped_reads/A.bam
```

where, as you may notice, the parameter passed no snakemake is the content of the output directive. In this way we are asking snakemake to generate that output file.

If the referred rule does not have wildcards, we could also invoke snakemake using rule names as targets. An example of DAG is reported in Fig. 1.

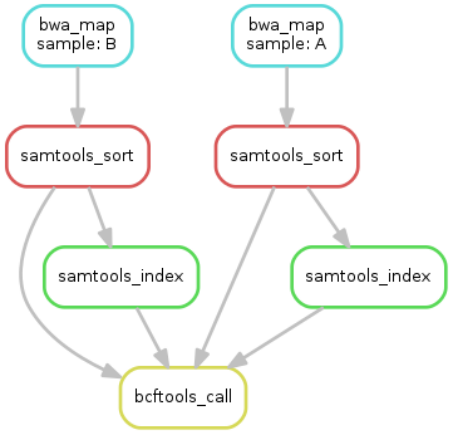

Figure 1: An example DAG

3.3 Generalization of the mapping rule

In the previous example, one of our input directives was referring to the file `"data/samples/A.fastq"`. What if we would like to generalise this rule? We can use a wildcard:

```
rule bwa_map:
    input:
        "data/genome.fa",
        "data/samples/{sample}.fastq"
    output:
        "mapped_reads/{sample}.bam"
    shell:
        "bwa mem {input} | samtools view -Sb - > {output}"
```

Both input and output directives have a `{sample}` wildcard. We can notice that the output directive has the same basename of the second input directive. So, to generate the output file, we just need to write:

```
$ snakemake -np mapped_reads/B.bam
```

In this way snakemake will infer that `B` is a good candidate for the wildcard `{sample}` and then the input file will be `data/samples/B.fastq` as we desired. Bash expansion can also be used to render this functionality very handy.
Note: Snakemake allows to access wildcards in the shell command via the `wildcards` object that has an attribute with the value for each wildcard. For example, to use the value of our wildcard in the shell command, we can't use `{sample}`, but we must use `{wildcards.sample}`.

3.4 Aggregation

Snakemake provides a helper function for collecting input files that helps us to describe the aggregation in this step. For example, with the following line:

```
expand("sorted_reads/{sample}.bam", sample=SAMPLES)
```

we obtain a list of files where the given pattern `sorted_reads/{sample}.bam` was formatted with the values in a given list of samples `SAMPLES`, i.e. `"sorted_reads/A.bam"` and `"sorted_reads/B.bam"`.

We can use multiple wildcards and have their product:

```
expand("sorted_reads/{sample}.{replicate}.bam", sample=SAMPLES, replicate=[0, 1])
```

and then obtain:

- sorted_reads/A.0.bam
- sorted_reads/A.1.bam
- sorted_reads/B.0.bam
- sorted_reads/B.1.bam

3.5 Scripts

The shell directive is used to specify the rules to generate output files from the input ones. We may also want to make other tasks, for example generating statistics. This can be done with proper scripts. Since the script logic should be separated from the workflow logic, snakemake provides the script directive through which we can invoke a proper Python or R script. In that script, all the properties of the rule like input, output, wildcards, etc. are available as attributes of a global snakemake object.
Let's see a simple example. The following is our simple rule:

```
rule plot_quals:
    input:
        "calls/all.vcf"
    output:
        "plots/quals.svg"
    script:
        "scripts/plot-quals.py"
```

The content of the script plot-quals.py is reported in the following listing:

```
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
from pysam import VariantFile

quals = [record.qual for record in VariantFile(snakemake.input[0])]
plt.hist(quals)

plt.savefig(snakemake.output[0])
```

3.6 Best practices

It is best practice to have a rule named all **at the top** of the workflow which has all typically desired target files as input files. This rule is placed on top because snakemake considers the first rule of the workflow as default target. Apart from this, the appearance of rules in the snakefile is arbitrary and does not influence the DAG of jobs.

4 Main Commands

- Ask the generation of a given output file (snakemake will determine how to do it):

```
$ snakemake content_of_output_directive
```

- Generate a visual representation of the DAG of dependencies:

```
$ snakemake --dag | dot -Tsvg > dat.svg
```

- See what rules and commands would be executed:

```
$ snakemake --dryrun --printshellcmds
```

5 References

- Snakemake documentation
- Snakemake - Reproducible and Scalable Bioinformatic Workflows
- The Snakemake Workflow Manager