Snakemake Overview

Why Snakemake is awesome
and everyone should use it forever

Snakemake

- a dialect of Python for writing computational pipelines, and
- a tool to interpret the pipelines written in this dialect

Rule-based language modelled after UNIX Make.

Example:

```
rule sort_bam:
   input: "not_sorted.bam"
   output: "sorted.bam"
   shell: 'samtools sort not_sorted.bam -o sorted.bam'
```

Launching Snakemake

1. create a Conda environment with `snakemake-minimal` package
2. within that environment, call
   $ `snakemake`
3. Profit!

That's assuming that the main script resides in a file named `Snakefile` in the current directory (just like `make` and `Makefile`!).

Accepts various command-line parameters to customize and tune the workflow.

Rules

`rule` is the main keyword of Snakemake, followed by a unique identifier. A number of fields can then be provided for a `rule`:

- `input, output`
- `log, params, threads, shadow, ...`
- `shell / run / script / ...`

Rules example

```
rule sort_bam:
    input: "not_sorted.bam"
    output: "sorted.bam"
    shell: 'samtools sort not_sorted.bam -o sorted.bam'
```

This rule:

- receives a file `not_sorted.bam` in the working directory
- calls the specified shell command
- expects to find a file `sorted.bam` once that shell command finishes

Shell specifics

By default, `snakemake` runs shell commands in **unofficial bash strict mode!**

This means that every shell command is implicitly prefaced with

```
set -euo pipefail;
```

Translated into human language:

- fail immediately if any command has non-zero exit status;
- fail on uninitialized variables;
- if any part of pipe fails, the pipe fails.

Chaining rules

Just like `make`, `snakemake` is designed for chaining rules.

```
rule sam_to_unsorted_bam:
   input: "sample.sam"
   output: "not_sorted.bam"
   shell: 'samtools view -bS sample.sam -o not_sorted.bam'
rule sort_bam:
   input: "not_sorted.bam"
   output: "sorted.bam"
   shell: 'samtools sort not_sorted.bam -o sorted.bam'
```

If `sort_bam` doesn't find its input in working directory, it will automatically try to invoke `sam_to_unsorted_bam` to produce it.

Rule DAG

When launched, `snakemake` organizes rules into DAG.

They are then invoked in a topological order if necessary.

A rule won't be invoked if its output exists and is newer than input.

The user can provide the rules that they want invoked (just like `make`!).

```
$ snakemake sort_bam
```

If not provided, `snakemake` targets the first rule of the script.

DRY and format language

```
rule sort_bam:
   input: "not_sorted.bam"
   output: "sorted.bam"
   shell: 'samtools sort not_sorted.bam -o sorted.bam'
```

can be replaced by

```
rule sort_bam:
   input: "not_sorted.bam"
   output: "sorted.bam"
   shell: 'samtools sort {input} -o {output}'
```

Snakemake applies `format` to the shell string before launching.

DRY and wildcards

What if we need to sort more than one file?

```
rule sort_bam:
    input: "{sample}_not_sorted.bam"
    output: "{sample}_sorted.bam"
    shell: 'samtools sort {input} -o {output}'
```

An implicit wildcard value `sample` is added here. If we then request sorted files:

```
rule all:
    input: "A_sorted.bam", "B_sorted.bam"
    output: "result.tsv"
    shell: 'do-smth {input}'
```

`snakemake` will infer that it can generate those files by applying `sort_bam` with `sample=A` and `sample=B`.

Wildcards and mysterious exceptions

Can you see a problem with this rule set?

```
rule sort_bam:
    input: "{sample}_not_sorted.bam"
    output: "{sample}_sorted.bam"
    shell: 'samtools sort {input} -o {output}'
rule all:
    input: "A_sorted.bam", "B_sorted.bam"
    output: "result.tsv"
    shell: 'do-smth {input}'
```

Wildcards and mysterious exceptions

Can you see a problem with this rule set?

```
rule sort_bam:
    input: "{sample}_not_sorted.bam"
    output: "{sample}_sorted.bam"
    shell: 'samtools sort {input} -o {output}'
rule all:
    input: "A_sorted.bam", "B_sorted.bam"
    output: "result.tsv"
    shell: 'do-smth {input}'
```

`snakemake` needs `A_not_sorted.bam`, but that file can be produced by `sort_bam` with `sample=A_not`! An infinite recursion ensues.

Fortunately, `snakemake` can detect it and generate a special `PeriodicWildcardException`.

Wildcards and mysterious exceptions

How to fix this?

- rename the file patterns to avoid recursion
- narrow down the wildcard value by specifying a regex:

  input: **"{sample, [a-zA-Z]*}_not_sorted.bam"**

  Now `sample` won't match `A_not`.
- specify rule precedence (e.g. always try `sam_to_unsorted_bam` first):

  ruleorder: sam_to_unsorted_bam > sort_bam

Named entries and params

`input`, `output` and other fields can have named entries.

```
rule align_paired_sam:
   input:
      first="{sample}_1.fastq",
      second="{sample}_2.fastq"
   output: "bams/{sample}.sam"
   shell: 'bowtie -St -m 1 -v 3 --trim5 5 --best --strata mm9 -1 {input.first} -2 {input.second} {output}'
```

You can also provide shell with parameters calculated in Python:

```
rule call_peaks_sicer:
   [...]
   params:
      effective_genome_fraction=effective_genome_fraction("mm9", "mm9.chrom.sizes")
    shell: 'SICER-rb.sh bams/pileup {input.bed} sicer mm9 1 200 150 {params.effective_genome_fraction} 600 100'
```

Configuration

`snakemake` pipeline can be configured in two ways:

- providing a YAML file either in `Snakefile` or as a command-line parameter;
- providing a set of key-value pairs via command-line:
  `--config key1=value1 key2=value2`

Either way, the config is then accessible via built-in `config` object.

```
rule download_chrom_sizes:
  output: "{config[genome]}.chrom.sizes"
  shell: 'wget -nc -O {output} '
         'http://hgdownload.cse.ucsc.edu/goldenPath/{config[genome]}/bigZips/{config[genome]}.chrom.sizes'
```

Concurrency

`snakemake` runs concurrently out of the box.

- set maximum concurrency with `--threads <num>` command-line option;
- request threads for your rules with threads field;
- not all required threads might be granted by scheduler.

```
rule align_paired_sam:
  input:
     first="{sample}_1.fastq",
     second="{sample}_2.fastq"
  output: "bams/{sample}.sam"
  threads: 4
  shell: 'bowtie -p {threads} -St -m 1 -v 3 --trim5 5 --best --strata mm9 -1 {input.first} -2 {input.second} {output}'
```

Cluster, cloud etc.

`snakemake` should be able to deal with cluster and cloud computation.

Just provide an appropriate job scheduler (e.g. qsub) via command line.

You can also create a custom wrapper that provides additional qsub parameters based on rule properties.

Temporary and protected output

Wrap output in `temp` method if it's not needed when pipeline finishes.

```
rule align_paired_sam:
    input:
        first="{sample}_1.fastq",
        second="{sample}_2.fastq"
    output: temp("bams/{sample}.sam")
    threads: 4
    shell: 'bowtie -p {threads} -St -m 1 -v 3 --trim5 5 --best --strata mm9 -1 {input.first} -2 {input.second} {output}'
```

It will be deleted once all rules that require it as input have finished running.

Conversely, a file wrapped in `protected` is write-protected after the rule completes.

Shadow rules

Use `shadow` field to run a rule inside a temporary directory with symlinks.

The shadow directory will be removed once the rule completes (naturally, the output will be copied to the main folder first).

Shadow levels:

- `minimal`: just the inputs are symlinked;
- `shallow`: top-level files and directories are symlinked;
- `full`: the entire working directory structure is replicated.

Logs

Provide a `log` field to let snakemake know that you're keeping a log file.

Log files can be detected as `input` for other rules.

```
rule align_paired_sam:
    input:
        first="{sample}_1.fastq",
        second="{sample}_2.fastq"
    output: temp("bams/{sample}.sam")
    threads: 4
    log: "{sample}.log"
    shell: '(bowtie -p {threads} -St -m 1 -v 3 --trim5 5 --best --strata mm9 -1 {input.first} -2 {input.second} {output}) '
           '&> {log}'
```

Why not just make them `output`?

Logs

Provide a `log` field to let snakemake know that you're keeping a log file.

Log files can be detected as `input` for other rules (e.g. multiQC).

```
rule align_paired_sam:
    input:
        first="{sample}_1.fastq",
        second="{sample}_2.fastq"
    output: temp("bams/{sample}.sam")
    threads: 4
    log: "{sample}.log"
    shell: '(bowtie -p {threads} -St -m 1 -v 3 --trim5 5 --best --strata mm9 -1 {input.first} -2 {input.second} {output}) '
           '&> {log}'
```

Why not just make them `output`?

`output` is deleted when the rule completes with an error, while `log` is not!

Various features and tricks

- Provide a function as `input` to generate complex input requirements based on wildcards.
- Provide a `conda` field with an environment file to run the rule in the specified Conda environment (looking at you, MACS2).
- Wrap `input`/`output` in `directory` method if it's a directory:
  - explicit permission to delete on rule failure or relaunch,
  - correct timestamp handling.

```
rule download_fa:
    output: directory("fa")
    shell: 'rsync -avz --partial --exclude="*.txt" '
           'rsync://hgdownload.cse.ucsc.edu/goldenPath/{config[genome]}/chromosomes/ {output} && '
           'gunzip -f {output}/*.fa.gz'
```

Wrappers

Use a wide array of wrappers from Snakemake Wrapper Repository to avoid writing the same shell commands again and again.

```
rule sort_bam:
    input: "{sample}_not_sorted.bam"
    output: "{sample}_sorted.bam"
    wrapper: '0.2.0/bio/samtools/sort'
```

A wrapper comes with a version tag, increasing reproducibility.

Other rule invocation options

Beside `shell` and `wrapper`, one can use:

- `run` -- arbitrary Python code
- `script` -- Python or R script file. The rule environment is provided as a `snakemake` object available in the script's scope.
- `cwl` -- tool descriptions in something called Common Workflow Language.

Modules

Modules can be included in the main pipeline with `include` keyword.

Sub-workflows can be defined and used as input via `subworkflow` keyword.