Show Source

# neal.sampler.SimulatedAnnealingSa

**SimulatedAnnealingSampler.sample**(*bqm: BinaryQuadraticModel, beta_range: List[float] | Tuple[float, float] | None = None, num_reads: int | None = None, num_sweeps: int | None = None, num_sweeps_per_beta: int = 1, beta_schedule_type: Literal['linear', 'geometric', 'custom'] = 'geometric', seed: int | None = None, interrupt_function=None, beta_schedule: Sequence[float] | ndarray | None = None, initial_states: Sequence[float | floating | integer] | Mapping[Hashable, float | floating | integer] | Tuple[Sequence[float | floating | integer], Sequence[Hashable]] | Tuple[ndarray, Sequence[Hashable]] | ndarray | Sequence[Sequence[float | floating | integer]] | Tuple[Sequence[Sequence[float | floating | integer]], Sequence[Hashable]] | Sequence[Sequence[float | floating | integer] | Mapping[Hashable, float | floating | integer] | Tuple[Sequence[float | floating | integer], Sequence[Hashable]] | Tuple[ndarray, Sequence[Hashable]] | ndarray] | Iterator[Sequence[float | floating | integer] | Mapping[Hashable, float | floating | integer] | Tuple[Sequence[float | floating | integer], Sequence[Hashable]] | Tuple[ndarray, Sequence[Hashable]] | ndarray] | None = None, initial_states_generator: Literal['none', 'tile', 'random'] = 'random', randomize_order: bool = False, proposal_acceptance_criteria: str = 'Metropolis', **kwargs*) → SampleSet [source]

Sample from a binary quadratic model.

**Parameters:**

- **bqm** – Binary quadratic model to be sampled.
- **beta_range** – A 2-tuple or list defining the beginning and end of the $\beta$[1] schedule. The schedule is interpolated within this range according to the value specified by `beta_schedule_type`. Default range is set based on the total bias associated with each node.
- **num_reads** – Number of reads. Each read is generated by one run of the simulated annealing algorithm. If `num_reads` is not explicitly given, it is selected to match the number of initial states given. If initial states are not provided, only one read is performed.
- **num_sweeps** – Number of sweeps used in annealing. If no value is provided and `beta_schedule` is None, the value defaults to 1000.
- **num_sweeps_per_beta** (*int, optional, default=1*) – Number of sweeps to perform at each $\beta$. One sweep consists of a sequential Metropolis update of all spins.
- **beta_schedule_type** –

  $\beta$ schedule type, or how the $\beta$ values are interpolated between the given `beta_range`. Supported values are:

  - "linear"
  - "geometric"
  - "custom"

  "custom" is recommended for high-performance applications, which typically require optimizing $\beta$ schedules beyond those of the "linear" and "geometric" options, with bounds beyond those provided by default. `num_sweeps_per_beta` and `beta_schedule` fully specify a custom schedule.
- **beta_schedule** – Sequence of $\beta$ values swept. Format must be compatible with `numpy.array(beta_schedule, dtype=float)`. Values should be non-negative.
- **seed** – Seed to use for the PRNG. Specifying a particular seed with a constant set of parameters produces identical results. If not provided, a random seed is chosen.
- **initial_states** – One or more samples, each defining an initial state for all

the problem variables. Initial states are given one per read, but if fewer than `num_reads` initial states are defined, additional values are generated as specified by `initial_states_generator`. See `as_samples()` for a description of “samples-like”.

- **initial_states_generator** –

  Defines the expansion of `initial_states` if fewer than `num_reads` are specified:

  - **”none”:**

    If the number of initial states specified is smaller than `num_reads`, raises `ValueError`.

  - **”tile”:**

    Reuses the specified initial states if fewer than `num_reads` or truncates if greater.

  - **”random”:**

    Expands the specified initial states with randomly generated states if fewer than `num_reads` or truncates if greater.

- **randomize_order** –

  When *True*, each spin update selects a variable uniformly at random. This method is ergodic, obeys detailed balance and preserves symmetries of the model.

  When *False*, updates proceed sequentially through the labeled variables on each sweep so that all variables are updated once per sweep. This method:

  - can be non-ergodic in special cases when used with `proposal_acceptance_critera=="Metropolis"`.
  - can introduce a dynamical bias as a function of variable order.
  - has faster per spin update than the True method.

- **proposal_acceptance_criteria** – When “Gibbs”, each spin flip proposal is accepted according to the Gibbs criteria. When “Metropolis”, each spin flip proposal is accepted according to the Metropolis-Hastings criteria.
- **interrupt_function** (*function, optional*) – A function called with no parameters between each sample of simulated annealing. If the function returns True, simulated annealing terminates and returns with all of the samples and energies found so far.

**Returns:**

A *dimod.SampleSet* for the binary quadratic model.

The *info* field of the sample set contains information about the sampling procedure: 1. the beta range used, 2. the beta schedule type used, and 3. timing information (details below).

Timing information is categorized into three: preprocessing, sampling, and postprocessing time. All timings are reported in units of nanoseconds. Preprocessing time is the total time spent converting the BQM variable type (if required), parsing input arguments, and determining an annealing schedule. Sampling time is the total time the algorithm spent in sampling states of the binary quadratic model. Postprocessing time is the total time spent reverting variable type and creating a *dimod.SampleSet*.

**Examples**

This example runs simulated annealing on a binary quadratic model with various input parameters.

```
>>> import dimod
>>> from dwave.samplers import SimulatedAnnealingSampler
...
>>> sampler = SimulatedAnnealingSampler()
>>> bqm = dimod.BinaryQuadraticModel({'a': .5, 'b': -.5},
...                                  {('a', 'b'): -1}, 0.0,
...                                  dimod.SPIN)
>>> # Run with default parameters
>>> sampleset = sampler.sample(bqm)
>>> # Run with specified parameters
>>> sampleset = sampler.sample(bqm, seed=1234,
...                            beta_range=[0.1, 4.2],
...                            num_sweeps=20,
...                            beta_schedule_type='geometric')
>>> # Reuse a seed
>>> a1 = next((sampler.sample(bqm, seed=88)).samples())['a']
>>> a2 = next((sampler.sample(bqm, seed=88)).samples())['a']
```

```
>>> a1 == a2
True
```

[1] $\beta$ represents the inverse temperature, $1/(k_B T)$, of a Boltzmann distribution where $T$ is the thermodynamic temperature in kelvin and $k_B$ is Boltzmann's constant.