python-nnf

Dec 13, 2022

Contents:

`python-nnf` is a package for working with logical sentences written in the negation normal form.

CHAPTER 1

Installing

`python-nnf` can be installed with pip:

```
pip install --user nnf
```

CHAPTER 2

Module reference

2.1 Module contents

class nnf.**NNF**

Bases: `object`

Base class for all NNF sentences.

clause() → bool

The sentence is a clause.

Clauses are Or nodes with variable children that don't share names.

condition(*model: Dict[Hashable, bool]*) → nnf.NNF

Fill in all the values in the dictionary.

consistent() → bool

Some set of values exists that makes the sentence correct.

This method doesn't necessarily try to find an example, which can make it faster. It's decent at decomposable sentences and sentences in CNF, and bad at other sentences.

contradicts(*other: nnf.NNF*) → bool

There is no set of values that satisfies both sentences.

decision_node() → bool

The sentence is a valid binary decision diagram (BDD).

decomposable() → bool

The children of each And node don't share variables, recursively.

deduplicate() → T_NNF

Return a copy of the sentence without any duplicate objects.

If a node has multiple parents, it's possible for it to be represented by two separate objects. This method gets rid of that duplication.

It's better to avoid the duplication in the first place. This method is for diagnostic purposes, in combination with `object_count()`.

deterministic() → bool
The children of each Or node contradict each other.

May be very expensive.

entails(*other: nnf.NNF*) → bool
Return whether `other` is always true if the sentence is true.

This is faster if `self` is a term or `other` is a clause.

equivalent(*other: nnf.NNF*) → bool
Test whether two sentences have the same models.

If the sentences don't contain the same variables they are considered equivalent if the variables that aren't shared are independent, i.e. their value doesn't affect the value of the sentence.

flat() → bool
A sentence is flat if its height is at most 2.

That is, there are at most two layers below the root node.

forget(*names: Iterable[Hashable]*) → nnf.NNF
Forget a set of variables from the theory.

Has the effect of returning a theory without the variables provided, such that every model of the new theory has an extension (i.e., an assignment) to the forgotten variables that is a model of the original theory.

Parameters names – An iterable of the variable names to be forgotten

forget_aux() → nnf.NNF
Returns a theory that forgets all of the auxillary variables

height() → int
The number of edges between here and the furthest leaf.

implicants() → nnf.Or[nnf.And[nnf.Var]][nnf.And[nnf.Var][nnf.Var]]
Extract the prime implicants of the sentence.

Prime implicants are the minimal terms that imply the sentence. This method returns a disjunction of terms that's equivalent to the original sentence, and minimal, meaning that there are no terms that imply the sentence that are strict subsets of any of the terms in this representation, so no terms could be made smaller.

implicates() → nnf.And[nnf.Or[nnf.Var]][nnf.Or[nnf.Var][nnf.Var]]
Extract a prime implicate cover of the sentence.

Prime implicates are the minimal implied clauses. This method returns a conjunction of clauses that's equivalent to the original sentence, and minimal, meaning that there are no clauses implied by the sentence that are strict subsets of any of the clauses in this representation, so no clauses could be made smaller.

While `implicants()` returns all implicants, this method may only return some of the implicates.

implies(*other: nnf.NNF*) → bool
Return whether `other` is always true if the sentence is true.

This is faster if `self` is a term or `other` is a clause.

is_CNF(*strict: bool = False*) → bool
Return whether the sentence is in the Conjunctive Normal Form.

Parameters strict – If `True`, follow the definition of the Knowledge Compilation Map, requiring that a variable doesn't appear multiple times in a single clause.

is_DNF(*strict: bool = False*) → bool

Return whether the sentence is in the Disjunctive Normal Form.

Parameters **strict** – If `True`, follow the definition of the Knowledge Compilation Map, requiring that a variable doesn't appear multiple times in a single term.

is_MODS() → bool

Return whether the sentence is in MODS form.

MODS sentences are disjunctions of terms representing models, making the models trivial to enumerate.

leaf() → bool

True if the node doesn't have children.

That is, if the node is a variable, or one of `true` and `false`.

make_pairwise() → nnf.NNF

Alter the sentence so that all internal nodes have two children.

This can be easier to handle in some cases.

make_smooth() → nnf.NNF

Transform the sentence into an equivalent smooth sentence.

mark_deterministic() → None

Declare for optimization that this sentence is deterministic.

Note that this goes by object identity, not equality. This may matter in obscure cases where you instantiate the same sentence multiple times.

marked_deterministic() → bool

Whether this sentence has been marked as deterministic.

model_count() → int

Return the number of models the sentence has.

This can be done efficiently for sentences that are decomposable and deterministic.

models() → Iterator[Dict[Hashable, bool]]

Yield all dictionaries of values that make the sentence correct.

Much faster on sentences that are decomposable. Even faster if they're also deterministic.

negate() → nnf.NNF

Return a new sentence that's true iff the original is false.

object_count() → int

Return the number of distinct node objects in the sentence.

project(*names: Iterable[Hashable]*) → nnf.NNF

Dual of `forget()`: will forget all variables not given

satisfiable() → bool

Some set of values exists that makes the sentence correct.

This method doesn't necessarily try to find an example, which can make it faster. It's decent at decomposable sentences and sentences in CNF, and bad at other sentences.

satisfied_by(*model: Dict[Hashable, bool]*) → bool

The given dictionary of values makes the sentence correct.

simplify(*merge_nodes: bool = True*) → nnf.NNF

Apply the following transformations to make the sentence simpler:

- If an And node has *false* as a child, replace it by *false*

- If an Or node has *true* as a child, replace it by *true*
- Remove children of And nodes that are *true*
- Remove children of Or nodes that are *false*
- If an And or Or node only has one child, replace it by that child
- If an And or Or node has a child of the same type, merge them

  Parameters merge_nodes – if False, don't merge internal nodes. In certain cases, merging them may increase the size of the sentence.

simply_conjunct() → bool
The children of And nodes are variables that don't share names.

simply_disjunct() → bool
The children of Or nodes are variables that don't share names.

size() → int
The number of edges in the sentence.

Note that sentences are rooted DAGs, not trees. If a node has multiple parents its edges will still be counted just once.

smooth() → bool
The children of each Or node all use the same variables.

solve() → Optional[Dict[Hashable, bool]]
Return a satisfying model, or None if unsatisfiable.

term() → bool
The sentence is a term.

Terms are And nodes with variable children that don't share names.

to_CNF(*simplify: bool = True*) → nnf.And[nnf.Or[nnf.Var]][nnf.Or[nnf.Var][nnf.Var]]
Compile theory to a semantically equivalent CNF formula.

Parameters simplify – If True, simplify clauses even if that means eliminating variables.

to_DOT(*, *color: bool = False*, *color_dict: Optional[Dict[str, str]] = None*, *label: str = 'text'*, *label_dict: Optional[Dict[str, str]] = None*) → str
Return a representation of the sentence in the DOT language.

DOT is a graph visualization language.

Parameters

- **color** – If True, color the nodes. This is a bit of an eyesore, but might make them easier to understand.
- **label** – If 'text', the default, label nodes with "AND", "OR", etcetera. If 'symbol', label them with unicode symbols like "" and "".
- **color_dict** – Use an alternative palette. This should be a dictionary with keys 'and', 'or', 'true', 'false', 'var' and 'neg'. Not all keys have to be included. Passing a dictionary implies color=True.
- **label_dict** – Use alternative labels for nodes. This should be a dictionary with keys 'and', 'or', 'true' and 'false'. Not all keys have to be included.

to_MODS() → nnf.Or[nnf.And[nnf.Var]][nnf.And[nnf.Var][nnf.Var]]
Convert the sentence to a MODS sentence.

to_model() → Dict[Hashable, bool]
If the sentence directly represents a model, convert it to that.

A sentence directly represents a model if it's a conjunction of (unique) variables, or a single variable.

valid() → bool
Check whether the sentence is valid (i.e. always true).

This can be done efficiently for sentences that are decomposable and deterministic.

vars() → FrozenSet[Hashable]
The names of all variables that appear in the sentence.

walk() → Iterator[nnf.NNF]
Yield all nodes in the sentence, depth-first.

Nodes with multiple parents are yielded only once.

class nnf.**Internal**(*children: Iterable[T_NNF_co] = ()*)
Bases: *nnf.NNF*, typing.Generic

Base class for internal nodes, i.e. And and Or nodes.

children

leaf() → bool
True if the node doesn't have children.

That is, if the node is a variable, or one of `true` and `false`.

map(*func: Callable[[T_NNF_co], U_NNF]*) → nnf.Internal[~U_NNF][U_NNF]
Apply `func` to all of the node's children.

class nnf.**And**(*children: Iterable[T_NNF_co] = ()*)
Bases: *nnf.Internal*

Conjunction nodes, which are only true if all of their children are.

decision_node() → bool
The sentence is a valid binary decision diagram (BDD).

class nnf.**Or**(*children: Iterable[T_NNF_co] = ()*)
Bases: *nnf.Internal*

Disjunction nodes, which are true if any of their children are.

decision_node() → bool
The sentence is a valid binary decision diagram (BDD).

class nnf.**Var**(*name: Hashable, true: bool = True*)
Bases: *nnf.NNF*

A variable, or its negation.

If its name is a string, its repr will use that name directly. Otherwise it will use more ordinary constructor syntax.

```
>>> a = Var('a')
>>> a
a
>>> ~a
~a
>>> b = Var('b')
>>> a | ~b == Or({Var('a', True), Var('b', False)})
True
>>> Var(10)
```

(continues on next page)

(continued from previous page)

```
Var(10)
>>> Var(('a', 'b'), False)
~Var(('a', 'b'))
```

static aux() → nnf.Var
Create an auxiliary variable with a unique label.

decision_node() → bool
The sentence is a valid binary decision diagram (BDD).

name

true

class nnf.**Aux**(*hex=None*, *bytes=None*, *bytes_le=None*, *fields=None*, *int=None*, *version=None*, *, *is_safe=<SafeUUID.unknown: None>*)
Bases: `uuid.UUID`

Unique UUID labels for auxiliary variables.

Don't instantiate directly, call `Var.aux()` instead.

nnf.**all_models**(*names: Iterable[Hashable]*) → Iterator[Dict[Hashable, bool]]
Yield dictionaries with all possible boolean values for the names.

```
>>> list(all_models(["a", "b"]))
[{'a': False, 'b': False}, {'a': False, 'b': True}, ...
```

nnf.**complete_models**(*models: Iterable[Dict[Hashable, bool]], names: Iterable[Hashable]*) → Iterator[Dict[Hashable, bool]]

nnf.**decision**(*var: nnf.Var*, *if_true: T_NNF*, *if_false: U_NNF*) → nnf.Or[typing.Union[nnf.And[typing.Union[nnf.Var, ~T_NNF]], nnf.And[typing.Union[nnf.Var, ~U_NNF]]]][Union[nnf.And[typing.Union[nnf.Var, ~T_NNF]][Union[nnf.Var, T_NNF]], nnf.And[typing.Union[nnf.Var, ~U_NNF]][Union[nnf.Var, U_NNF]]]]
Create a decision node with a variable and two branches.

Parameters

- **var** – The variable node to decide on.
- **if_true** – The branch if the decision is true.
- **if_false** – The branch if the decision is false.

nnf.**true = true**
A node that's always true. Technically an And node without children.

nnf.**false = false**
A node that's always false. Technically an Or node without children.

2.2 Submodules

2.3 nnf.operators module

Convenience functions for logical relationships that are not part of NNF.

These functions will simulate those relationships, often by doubling sentences or altering their structure to negate them. This makes them inefficient.

`nnf.operators.`**`xor`**(*a: nnf.NNF*, *b: nnf.NNF*) → nnf.Or[nnf.And[nnf.NNF]][nnf.And[nnf.NNF][nnf.NNF]]
Exactly one of the operands is true.

`nnf.operators.`**`nand`**(*a: nnf.NNF*, *b: nnf.NNF*) → nnf.Or[nnf.NNF][nnf.NNF]
At least one of the operands is false.

`nnf.operators.`**`nor`**(*a: nnf.NNF*, *b: nnf.NNF*) → nnf.And[nnf.NNF][nnf.NNF]
Both of the operands are false.

`nnf.operators.`**`implies`**(*a: nnf.NNF*, *b: nnf.NNF*) → nnf.Or[nnf.NNF][nnf.NNF]
`b` is true whenever `a` is true.

`nnf.operators.`**`implied_by`**(*a: nnf.NNF*, *b: nnf.NNF*) → nnf.Or[nnf.NNF][nnf.NNF]
`a` is true whenever `b` is true.

`nnf.operators.`**`iff`**(*a: nnf.NNF*, *b: nnf.NNF*) → nnf.Or[nnf.And[nnf.NNF]][nnf.And[nnf.NNF][nnf.NNF]]
`a` is true if and only if `b` is true.

`nnf.operators.`**`and_`**(*a: T_NNF*, *b: U_NNF*) → nnf.And[typing.Union[~T_NNF, ~U_NNF]][Union[T_NNF, U_NNF]]
`a` and `b` are both true. Included for completeness.

`nnf.operators.`**`or_`**(*a: T_NNF*, *b: U_NNF*) → nnf.Or[typing.Union[~T_NNF, ~U_NNF]][Union[T_NNF, U_NNF]]
`a` or `b` is true. Included for completeness.

2.4 nnf.dimacs module

A parser and serializer for the DIMACS CNF and SAT formats.

`nnf.dimacs.`**`dump`**(*obj: nnf.NNF*, *fp: TextIO*, ***, *num_variables: Optional[int] = None*, *var_labels: Optional[Dict[Hashable, int]] = None*, *comment_header: Optional[str] = None*, *mode: str = 'sat'*) → None
Dump a sentence into an open file in a DIMACS format.

Variable names have to be integers. If the variables in the sentence you want to dump are not integers, you can pass a `var_labels` dictionary to map names to integers.

Parameters

- **`obj`** – The sentence to dump.
- **`fp`** – The open file.
- **`num_variables`** – Override the number of variables, in case there are variables that don't appear in the sentence.
- **`var_labels`** – A dictionary mapping variable names to integers, to rename non-integer variables.
- **`comment_header`** – A comment to include at the top of the file. May include newlines.
- **`mode`** – Either `'sat'` to dump in the general SAT format, or `'cnf'` to dump in the specialized CNF format.

`nnf.dimacs.`**`load`**(*fp: TextIO*) → Union[nnf.NNF, nnf.And[nnf.Or[nnf.Var]][nnf.Or[nnf.Var][nnf.Var]]]
Load a sentence from an open file.

The format is automatically detected.

`nnf.dimacs.`**`dumps`**(*obj: nnf.NNF*, ***, *num_variables: Optional[int] = None*, *var_labels: Optional[Dict[Hashable, int]] = None*, *comment_header: Optional[str] = None*, *mode: str = 'sat'*) → str

Like `dump()`, but to a string instead of to a file.

`nnf.dimacs.`**`loads`**(*s: str*) → Union[nnf.NNF, nnf.And[nnf.Or[nnf.Var]][nnf.Or[nnf.Var][nnf.Var]]]

Like `load()`, but from a string instead of from a file.

`exception` `nnf.dimacs.`**`DimacsError`**

Bases: `Exception`

`exception` `nnf.dimacs.`**`EncodeError`**

Bases: `nnf.dimacs.DimacsError`

`exception` `nnf.dimacs.`**`DecodeError`**

Bases: `nnf.dimacs.DimacsError`

2.5 nnf.dsharp module

Interoperability with DSHARP.

`load` and `loads` can be used to parse files created by DSHARP's `-Fnnf` option.

`compile` invokes DSHARP directly to compile a sentence. This requires having DSHARP installed.

The parser was derived by studying DSHARP's output and source code. This format might be some sort of established standard, in which case this parser might reject or misinterpret some valid files in the format.

DSHARP may not work properly for some (usually trivially) unsatisfiable sentences, incorrectly reporting there's a solution. This bug dates back to sharpSAT, on which DSHARP was based:

https://github.com/marcthurley/sharpSAT/issues/5

It was independently discovered by hypothesis during testing of this module.

`nnf.dsharp.`**`load`**(*fp: TextIO*, *var_labels: Optional[Dict[int, Hashable]] = None*) → nnf.NNF

Load a sentence from an open file.

An optional `var_labels` dictionary can map integers to other names.

`nnf.dsharp.`**`loads`**(*s: str*, *var_labels: Optional[Dict[int, Hashable]] = None*) → nnf.NNF

Load a sentence from a string.

`nnf.dsharp.`**`compile`**(*sentence: nnf.And[nnf.Or[nnf.Var]][nnf.Or[nnf.Var][nnf.Var]], executable: str = 'dsharp', smooth: bool = False, timeout: Optional[int] = None, extra_args: Sequence[str] = ()*) → nnf.NNF

Run DSHARP to compile a CNF sentence to (s)d-DNNF.

This requires having DSHARP installed.

The returned sentence will be marked as deterministic.

Parameters

- **`sentence`** – The CNF sentence to compile.
- **`executable`** – The path of the `dsharp` executable. If the executable is in your PATH there's no need to set this.
- **`smooth`** – Whether to produce a smooth sentence.
- **`timeout`** – Tell DSHARP to give up after a number of seconds.
- **`extra_args`** – Extra arguments to pass to DSHARP.

2.6 nnf.amc module

An implementation of algebraic model counting.

`nnf.amc.`**`eval`**(*node: nnf.NNF, add: Callable[[T, T], T], mul: Callable[[T, T], T], add_neut: T, mul_neut: T, labeling: Callable[[nnf.Var], T]*) → T

Execute an AMC technique, given a semiring and a labeling function.

Parameters

- **node** – The sentence to calculate the value of.
- **add** – The operator, to combine `nnf.Or` nodes.
- **mul** – The operator, to combine `nnf.And` nodes.
- **add_neut** – e^, the neutral element of the operator.
- **mul_neut** – e^, the neutral element of the operator.
- **labeling** – The labeling function, to assign a value to each variable node.

`nnf.amc.`**`reduce`**(*node: nnf.NNF, add_key: Optional[Callable[[T], Any]], mul: Callable[[T, T], T], add_neut: T, mul_neut: T, labeling: Callable[[nnf.Var], T]*) → nnf.NNF

Execute AMC reduction on a sentence.

In AMC reduction, the operator must be `max` on some total order, and the branches of the sentence that don't contribute to the maximum value are removed. This leaves a simpler sentence with only the models with a maximum value.

Parameters

- **node** – The sentence.
- **add_key** – A function given to `max`'s `key` argument to determine the total order of the operator. Pass `None` to use the default ordering.
- **mul** – See `eval()`.
- **add_neut** – See `eval()`.
- **mul_neut** – See `eval()`.
- **labeling** – See `eval()`.

Returns The transformed sentence.

`nnf.amc.`**`SAT`**(*node: nnf.NNF*) → bool

Determine whether a DNNF sentence is satisfiable.

`nnf.amc.`**`NUM_SAT`**(*node: nnf.NNF*) → int

Determine the number of models that satisfy a sd-DNNF sentence.

`nnf.amc.`**`WMC`**(*node: nnf.NNF, weights: Callable[[nnf.Var], float]*) → float

Model counting of sd-DNNF sentences, weighted by variables.

Parameters

- **node** – The sentence to measure.
- **weights** – A dictionary mapping variable nodes to weights.

`nnf.amc.`**`PROB`**(*node: nnf.NNF, probs: Dict[Hashable, float]*) → float

Model counting of d-DNNF sentences, weighted by probabilities.

Parameters

- **node** – The sentence to measure.
- **probs** – A dictionary mapping variable names to probabilities.

nnf.amc.**GRAD**(*node: nnf.NNF, probs: Dict[Hashable, float], k: Optional[Hashable] = None*) → Tuple[float, float]

Calculate a gradient of a d-DNNF sentence being true depending on the value of a variable, given probabilities for all variables.

Parameters

- **node** – The sentence.
- **probs** – A dictionary mapping variable names to probabilities.
- **k** – The name of the variable to check relative to.

Returns A tuple of two floats (probability, gradient).

nnf.amc.**MPE**(*node: nnf.NNF, probs: Dict[Hashable, float]*) → float

nnf.amc.**maxplus_reduce**(*node: nnf.NNF, labels: Dict[nnf.Var, float]*) → nnf.NNF

Execute AMC reduction using the maxplus algebra.

Parameters

- **node** – The sentence.
- **labels** – A dictionary mapping variable nodes to numbers.

2.7 nnf.tseitin module

Transformations using the well-known Tseitin encoding.

The Tseitin transformation converts any arbitrary circuit to one in CNF in polynomial time/space. It does so at the cost of introducing new variables (one for each logical connective in the formula).

nnf.tseitin.**to_CNF**(*theory: nnf.NNF, simplify: bool = True*) → nnf.And[nnf.Or[nnf.Var]][nnf.Or[nnf.Var][nnf.Var]]

Convert an NNF into CNF using the Tseitin Encoding.

Parameters

- **theory** – Theory to convert.
- **simplify** – If True, simplify clauses even if that means eliminating variables.

2.8 nnf.kissat module

Interoperability with kissat.

`solve` invokes the SAT solver directly on the given theory.

nnf.kissat.**solve**(*sentence: nnf.And[nnf.Or[nnf.Var]][nnf.Or[nnf.Var][nnf.Var]], extra_args: Sequence[str] = ()*) → Optional[Dict[Hashable, bool]]

Run kissat to compute a satisfying assignment.

Parameters

- **sentence** – The CNF sentence to solve.
- **extra_args** – Extra arguments to pass to kissat.

2.9 nnf.pysat module

`nnf.pysat.satisfiable`(*sentence: nnf.And[nnf.Or[nnf.Var]][nnf.Or[nnf.Var][nnf.Var]]*) → bool
Return whether a CNF sentence is satisfiable.

`nnf.pysat.solve`(*sentence: nnf.And[nnf.Or[nnf.Var]][nnf.Or[nnf.Var][nnf.Var]]*) → Optional[Dict[Hashable, bool]]
Return a model of a CNF sentence, or `None` if unsatisfiable.

`nnf.pysat.models`(*sentence: nnf.And[nnf.Or[nnf.Var]][nnf.Or[nnf.Var][nnf.Var]]*) → Iterator[Dict[Hashable, bool]]
Yield all models of a CNF sentence.

`nnf.pysat.available = False`
Indicates whether the PySAT library is installed and available for use.

CHAPTER 3

Command line interface

Some of python-nnf's functionality is exposed through a command line tool. It can be invoked as pynnf or python3 -m nnf.

3.1 SAT solving

pynnf sat tests whether a sentence is satisfiable, while pynnf sharpsat counts how many solutions it has.

Add -v to get extra information about the sentence and the running time.

Example:

```
$ pynnf sat uf20-01.cnf
SATISFIABLE
```

Beware that it's much slower than dedicated solvers like MiniSat.

3.2 Sentence summary

pynnf info shows basic information about a sentence.

Examples:

```
$ pynnf info uf20-01.cnf
Sentence is in CNF.
Variables:   20
Size:        360
Clauses:     90
Clause size: 3

$ pynnf info uf100-016.cnf.nnf
Sentence is decomposable.
```

(continues on next page)

(continued from previous page)

```
Variables:   97
Size:        109
```

3.3 Visualizing sentences

`pynnf draw` converts sentences to a DOT representation, and either outputs that or feeds it to `dot` to immediately output an image.

Immediately outputting an image requires having `dot` installed. It's done when the output file has an image extension, or when a format is passed with the `-f` flag.

Examples:

```
$ pynnf draw uf20-01.nnf out.png  # Create a PNG image

$ pynnf draw uf20-01.nnf out.gv   # Create a DOT representation

$ pynnf draw uf20-01.nnf out.pdf  # Create a PDF vector image

$ pynnf draw -f png uf20-01.nnf - | convert -flip - out.png  # Output a PNG image to
↪be processed by imagemagick
```

See `pynnf draw --help` for more information.

CHAPTER 4

Caveats

There are a few things to keep in mind when using `python-nnf`.

4.1 Node duplication

If the same node occurs multiple times in a sentence, then it often pays to make sure that it isn't created multiple times.

Here's a (contrived) example of two ways to construct the same sentence:

```
>>> inefficient = And({
...     Or({A, B}),
...     And({A, Or({A, B})}),
... })
>>> dup_node = Or({A, B})
>>> efficient = And({
...     dup_node,
...     And({A, dup_node}),
... })
```

These objects behave identically, but the first one stores the node `Or({A, B})` twice, and the other stores it only once. That means the second one uses less memory.

For a lot of sentences, this isn't worth worrying about. But if you have many nodes that occur multiple times, and they descend from nodes that occur multiple times, you may end up using a lot more memory than necessary.

The `.object_count()` and `.deduplicate()` methods exist to diagnose this problem. `.object_count()` tells you how many actual objects are used to represent the sentence, and `.deduplicate()` returns a maximally compact copy.

If `.deduplicate()` changes the value of `.object_count()` a lot then the sentence could benefit from watching out not to create objects multiple times.

```
>>> inefficient.object_count()
6
```

(continues on next page)

(continued from previous page)

```
>>> inefficient.deduplicate().object_count()
5
```

In this case the difference is pretty small.

4.2 Decomposability and determinism

A lot of methods are much faster to perform on sentences that are decomposable or deterministic, such as model enumeration.

Decomposability is automatically detected.

Determinism is too expensive to automatically detect, but it can give a huge speedup. If you know a sentence to be deterministic, run `.mark_deterministic()` to enable the relevant optimizations.

A compiler like DSHARP may be able to convert some sentences into equivalent deterministic decomposable sentences. The output of DSHARP can be loaded using the `nnf.dsharp` module. Sentences returned by `nnf.dsharp.compile()` are automatically marked as deterministic.

4.3 Other duplication inefficiencies

Even when properly deduplicated, the kind of sentence that's vulnerable to node duplication might still be inefficient to work with for some operations.

A known offender is equality (==). Currently, if two of such sentences are compared that are equal but don't share any objects, it takes a very long time even if both sentences don't have any duplication within themselves.

CHAPTER 5

Introduction

Sentences are made up of nodes. To start with, define some variables:

```
>>> from nnf import Var
>>> A, B, C = Var('A'), Var('B'), Var('C')
```

Then, if you want to write the sentence "A or B":

```
>>> from nnf import Or
>>> sentence = Or({A, B})
>>> sentence = A | B  # alternative syntax
```

Or "B and not C":

```
>>> from nnf import And
>>> sentence = And({B, ~C})
>>> sentence = B & ~C
```

Of course you can nest these, for more interesting sentences:

```
>>> sentence = Or({And({A, B}), And({~B, C})})
```

You can ask queries, and perform transformations:

```
>>> sentence.decomposable()
True
>>> sentence.smooth()
False
>>> list(sentence.models())
[{'A': True, 'B': True, 'C': True}, {'A': True, 'B': False, ...
>>> new = sentence.condition({'B': True})
>>> new
Or({And({A, true}), And({false, C})})
>>> list(new.models())
[{'A': True, 'C': True}, {'A': True, 'C': False}]
```

(continues on next page)

(continued from previous page)

```
>>> new.simplify()
A
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index