# CNFgen Documentation

*Release 0.7.3*

**Massimo Lauria**

**Aug 28, 2018**

# Table of contents

The entry point to `cnfformula` library is `cnfformula.CNF`, which is the data structure representing CNF formulas. Any string (actually any *hashable* object) is a valid name for a variable, that can be optionally documented with an additional description.

```
>>> from cnfformula import CNF
>>> F = CNF()
>>> F.add_variable("X",'this variable is the cause')
>>> F.add_variable("Y",'this variable is the effect')
>>> F.add_variable("Z") # no description here
```

Clauses are represented as lists of literals, where positive and negative literals are represented as `(True, <varname>)` or `(False, <varname>)`, respectively. The user can interleave the addition of variables and clauses.

```
>>> F.add_clause([(False,"X"),(True,"Y")])
>>> F.add_variable("W")
>>> F.add_clause([(False,"Z")])
```

The CNF object `F` in the example now encodes the formula

$$(\neg X \vee Y) \wedge (\neg Z)$$

over variables $X$, $Y$, $Z$ and $W$. It is perfectly fine to add variables that do not occur in any clause. Vice versa, it is possible to add clauses that mention variables never seen before. In that case any unknown variable is silently added to the formula (obviously without description).

```
>>> G = CNF()
>>> list(G.variables())
[]
>>> G.add_clause([(False,"X"),(True,"Y")])
>>> list(G.variables())
['X','Y']
```

---

**Note:** By default the `cnfformula.CNF.add_clause()` forbids a clauses to contain the same literal twice, or to have two opposite literals (i.e. a negative and a positive literal on the same variable). Furthermore, as mentioned before, it allows to add clauses with unknown variables.

It is possible to make change this behavior, for example it is possible to allow opposite literals, but simultaneously forbid unknown variables. See the documentation of `cnfformula.CNF.add_clause()` for further details.

---

Furthermore it is possible to add clauses directly while constructing the CNF object. The code

```
>>> H = CNF([ [(True,"x1"),(True,"x2"),(False,"x3")], [(False,"x2"),(True,"x4")] ])
```

is essentially equivalent to

```
>>> H = CNF()
>>> H.add_clause([(True,"x1"),(True,"x2"),(False,"x3")])
>>> H.add_clause([(False,"x2"),(True,"x4")])
```

# CHAPTER 1

## Exporting formulas to DIMACS

One of the main use of `CNFgen` is to produce formulas to be fed to SAT solvers. These solvers accept CNf formulas in DIMACS format[1], which can easily be obtained using `cnfformula.CNF.dimacs()`.

```
>>> c=CNF([ [(True,"x1"),(True,"x2"),(False,"x3")], [(False,"x2"),(True,"x4")] ])
>>> print( c.dimacs(export_header=False) )
p cnf 4 2
1 2 -3 0
-2 4 0
>>> c.add_clause( [(False,"x3"),(True,"x4"),(False,"x5")] )
>>> print( c.dimacs(export_header=False))
p cnf 5 3
1 2 -3 0
-2 4 0
-3 4 -5 0
```

The variables in the DIMACS representation are numbered according to the order of insertion. `CNFgen` does not guarantee anything about this order, unless variables are added explicitly.

[1] http://www.satlib.org/Benchmarks/SAT/satformat.ps

# CHAPTER 2

## Exporting formulas to LaTeX

It is possible to use `cnfformula.CNF.latex()` to get a LaTeX[2] encoding of the CNF to include in a document. In that case the variable names are included literally, therefore it is advisable to use variable names that would look good in Latex.

```
>>> c=CNF([[(False,"x_1"),(True,"x_2"),(False,"x_3")],\
... [(False,"x_2"),(False,"x_4")], \
... [(True,"x_2"),(True,"x_3"),(False,"x_4")]])
>>> print(c.latex(export_header=False))
\begin{align}
&       \left( {\overline{x}_1} \lor            {x_2} \lor {\overline{x}_3}␣
↪\right) \\
& \land \left( {\overline{x}_2} \lor {\overline{x}_4} \right) \\
& \land \left( {x_2} \lor {x_3} \lor {\overline{x}_4} \right)
\end{align}
```

which renders as

$$\begin{aligned} &(\overline{x}_1 \lor x_2 \lor \overline{x}_3) \\ \land\ &(\overline{x}_2 \lor \overline{x}_4) \\ \land\ &(x_2 \lor x_3 \lor \overline{x}_4) \end{aligned} \tag{2.1}$$

Instead of outputting just the LaTeX rendering of the formula it is possible to produce a full LaTeX document by setting the keyword argument `full_document` to `True` in the call to `cnfformula.CNF.latex()`. The document is ready to be compiled.

[2] http://www.latex-project.org/

CHAPTER 3

# Reference

# CHAPTER 4

## Testing satisfiability

To test the satisfiability of the CNF formula encoded in a `cnfformula.CNF` instance we can use the `cnfformula.CNF.is_satisfiable()` method. Testing satisfiability of a CNF is not at all considered to be an easy task. In full generality the problem is NP-hard[1], which essentially means that there are no fast algorithm to solve it.

In practice many formula that come from applications can be solved efficiently (i.e. it is possible to rapidly find a satisfying assignment). There is a whole community of clever software engineers and computer scientists that compete to write the fastest solver for CNF satisfiability (usually called a SAT solver)[2]. *CNFgen* does not implement a SAT solver, but uses behind the scenes the ones installed in the running environment. If the formula is satisfiable the value returned includes a satisfying assignment.

```
>>> from cnfformula import CNF
>>> F = CNF([ [(True,'X'),(False,'Y')], [(False,'X')] ])
>>> F.is_satisfiable()
(True, {'X': False, 'Y': False})
>>> F.add_clause([(True,'Y')])
>>> F.is_satisfiable()
(False, None)
```

It is always possible to force `CNFgen` to use a specific solver or a specific command line invocation using the `cmd` parameters for `cnfformula.CNF.is_satisfiable()`. `CNFgen` knows how to interface with several SAT solvers but when the command line invokes an unknown solver the parameter `sameas` can suggest the right interface to use.

```
>>> F.is_satisfiable(cmd='minisat -no-pre')
>>> F.is_satisfiable(cmd='glucose -pre')
>>> F.is_satisfiable(cmd='lingeling --plain')
>>> F.is_satisfiable(cmd='sat4j')
>>> F.is_satisfiable(cmd='my-hacked-minisat -pre',sameas='minisat')
>>> F.is_satisfiable(cmd='patched-lingeling',sameas='lingeling')
```

[1] NP-hardness is a fundamental concept coming from computational complexity, which is the mathematical study of how hard is to perform certain computations.
(https://en.wikipedia.org/wiki/NP-hardness)

[2] See http://www.satcompetition.org/ for SAT solver ranking.

CHAPTER 5

# Formula families

One of the most useful features of `CNFgen` is the implementation of several important families of CNF formulas, many of them either coming from the proof complexity literature or encoding some important problem from combinatorics. The formula are accessible through the `cnfformula` package. See for example this construction of the pigeohole principle formula with 10 pigeons and 9 holes.

```
>>> import cnfformula
>>> F = cnfformula.PigeonholePrinciple(10,9)
>>> F.is_satisfiable()
(False, None)
```

## 5.1 Included formula families

All formula generators are accessible from the `cnfformula` package, but their implementation (and documentation) is split across the following modules. This makes it easy to add new formula families.

### 5.1.1 cnfformula.families.counting module

Implementation of counting/matching formulas

**class CountingCmdHelper**
Command line helper for Counting Principle formulas

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build an Counting Principle formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for Counting Principle formula |

**static build_cnf**(*args*)
Build an Counting Principle formula according to the arguments

Arguments: - *args*: command line options

**description = 'counting principle'**

**name = 'count'**

**static setup_command_line**(*parser*)
Setup the command line options for Counting Principle formula

Arguments: - *parser*: parser to load with options.

**CountingPrinciple**(*M*, *p*)
Generates the clauses for the counting matching principle.

The principle claims that there is a way to partition M in sets of size p each.

Arguments: - *M* : size of the domain - *p* : size of each class

**class PMatchingCmdHelper**
Bases: object

Command line helper for Perfect Matching Principle formulas

### Methods

| | |
|---|---|
| *setup_command_line*(parser) | Setup the command line options for Perfect Matching Principle formula |

**build_cnf**

**static build_cnf**(*args*)

**description = 'perfect matching principle'**

**name = 'matching'**

**static setup_command_line**(*parser*)
Setup the command line options for Perfect Matching Principle formula

Arguments: - *parser*: parser to load with options.

**class ParityCmdHelper**
Bases: object

Command line helper for Parity Principle formulas

### Methods

| | |
|---|---|
| *setup_command_line*(parser) | Setup the command line options for Parity Principle formula |

**build_cnf**

**static build_cnf**(*args*)

**description = 'parity principle'**

**name = 'parity'**

**static setup_command_line**(*parser*)
Setup the command line options for Parity Principle formula

Arguments: - *parser*: parser to load with options.

**PerfectMatchingPrinciple**(*G*)
Generates the clauses for the graph perfect matching principle.

The principle claims that there is a way to select edges to such that all vertices have exactly one incident edge set to 1.

**Parameters**

**G** [undirected graph]

### 5.1.2 cnfformula.families.coloring module

Formulas that encode coloring related problems

**class ECCmdHelper**
Bases: object

#### Methods

| | |
|---|---|
| **build_cnf** | |
| **setup_command_line** | |

**static build_cnf**(*args*)

**description = 'even coloring formulas'**

**name = 'ec'**

**static setup_command_line**(*parser*)

**EvenColoringFormula**(*G*)
Even coloring formula

The formula is defined on a graph $G$ and claims that it is possible to split the edges of the graph in two parts, so that each vertex has an equal number of incident edges in each part.

The formula is defined on graphs where all vertices have even degree. The formula is satisfiable only on those graphs with an even number of vertices in each connected component *[1]*.

**Returns**

**CNF object**

**Raises**

**ValueError** if the graph in input has a vertex with odd degree

#### References

*[1]*

**GraphColoringFormula**(*G*, *colors*, *functional=True*)
Generates the clauses for colorability formula

The formula encodes the fact that the graph $G$ has a coloring with color set `colors`. This means that it is possible to assign one among the elements in ‘‘colors‘‘to that each vertex of the graph such that no two adjacent vertices get the same color.

**Parameters**

**G** [networkx.Graph] a simple undirected graph

**colors** [list or positive int] a list of colors or a number of colors

**Returns**

**CNF** the CNF encoding of the coloring problem on graph `G`

**class KColorCmdHelper**

Bases: `object`

Command line helper for k-color formula

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a k-colorability formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for k-color formula |

**static build_cnf**(*args*)

Build a k-colorability formula according to the arguments

Arguments: - *args*: command line options

**description = 'k-colorability formula'**

**name = 'kcolor'**

**static setup_command_line**(*parser*)

Setup the command line options for k-color formula

Arguments: - *parser*: parser to load with options.

### 5.1.3 cnfformula.families.graphisomorphism module

Graph isomorphimsm/automorphism formulas

**class GAutoCmdHelper**

Bases: `object`

Command line helper for Graph Automorphism formula

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a graph automorphism formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for graph automorphism formula |

**static build_cnf**(*args*)

Build a graph automorphism formula according to the arguments

Arguments: - *args*: command line options

**description = 'graph automorphism formula'**

**name = 'gauto'**

**static setup_command_line**(*parser*)

Setup the command line options for graph automorphism formula

Arguments: - *parser*: parser to load with options.

**class GIsoCmdHelper**

Bases: object

Command line helper for Graph Isomorphism formula

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a graph automorphism formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for graph isomorphism formula |

**static build_cnf**(*args*)
Build a graph automorphism formula according to the arguments

Arguments: - *args*: command line options

**description = 'graph isomorphism formula'**

**name = 'giso'**

**static setup_command_line**(*parser*)
Setup the command line options for graph isomorphism formula

Arguments: - *parser*: parser to load with options.

**GraphAutomorphism**(*G*)
Graph Automorphism formula

The formula is the CNF encoding of the statement that a graph G has a nontrivial automorphism, i.e. an automorphism different from the idential one.

**Returns**

**A CNF formula which is satiafiable if and only if graph G has a**

**nontrivial automorphism.**

**GraphIsomorphism**(*G1*, *G2*)
Graph Isomorphism formula

The formula is the CNF encoding of the statement that two simple graphs G1 and G2 are isomorphic.

**Parameters**

**G1** [networkx.Graph] an undirected graph object

**G2** [networkx.Graph] an undirected graph object

**Returns**

**A CNF formula which is satiafiable if and only if graphs G1 and G2**

**are isomorphic.**

### 5.1.4 cnfformula.families.ordering module

Implementation of the ordering principle formulas

**class GOPCmdHelper**
Bases: object

Command line helper for Graph Ordering principle formulas

### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a Graph ordering principle formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for Graph ordering principle formula |

**static build_cnf**(*args*)
Build a Graph ordering principle formula according to the arguments

Arguments: - *args*: command line options

**description = 'graph ordering principle'**

**name = 'gop'**

**static setup_command_line**(*parser*)
Setup the command line options for Graph ordering principle formula

Arguments: - *parser*: parser to load with options.

**GraphOrderingPrinciple**(*graph*, *total=False*, *smart=False*, *plant=False*, *knuth=0*)
Generates the clauses for graph ordering principle

Arguments: - *graph* : undirected graph - *total* : add totality axioms (i.e. "x < y" or "x > y") - *smart* : "x < y" and "x > y" are represented by a single variable (implies *total*) - *plant* : allow last element to be minimum (and could make the formula SAT) - *knuth* : Don Knuth variants 2 or 3 of the formula (anything else suppress it)

**class OPCmdHelper**
Bases: object

Command line helper for Ordering principle formulas

### Methods

| | |
|---|---|
| *build_cnf*(args) | Build an Ordering principle formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for Ordering principle formula |

**static build_cnf**(*args*)
Build an Ordering principle formula according to the arguments

Arguments: - *args*: command line options

**description = 'ordering principle'**

**name = 'op'**

**static setup_command_line**(*parser*)
Setup the command line options for Ordering principle formula

Arguments: - *parser*: parser to load with options.

**OrderingPrinciple**(*size*, *total=False*, *smart=False*, *plant=False*, *knuth=0*)
Generates the clauses for ordering principle

Arguments: - *size* : size of the domain - *total* : add totality axioms (i.e. "x < y" or "x > y") - *smart* : "x < y" and "x > y" are represented by a single variable (implies totality) - *plant* : allow a single element to be minimum (could make the formula SAT) - *knuth* : Donald Knuth variant of the formula ver. 2 or 3 (anything else suppress it)

**varname**(*v1*, *v2*)

### 5.1.5 cnfformula.families.pebbling module

Implementation of the pigeonhole principle formulas

**class PebblingCmdHelper**
Command line helper for pebbling formulas

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build the pebbling formula |
| *setup_command_line*(parser) | Setup the command line options for pebbling formulas |

**static build_cnf**(*args*)
Build the pebbling formula

Arguments: - *args*: command line options

**description = 'pebbling formula'**

**name = 'peb'**

**static setup_command_line**(*parser*)
Setup the command line options for pebbling formulas

Arguments: - *parser*: parser to load with options.

**PebblingFormula**(*digraph*)
Pebbling formula

Build a pebbling formula from the directed graph. If the graph has an *ordered_vertices* attribute, then it is used to enumerate the vertices (and the corresponding variables).

Arguments: - *digraph*: directed acyclic graph.

**class SparseStoneCmdHelper**
Sparse Stone formulas

This is a variant of the *StoneFormula()*. See that for a description of the formula. This variant is such that each vertex has only a small selection of which stone can go to that vertex. In particular which stones are allowed on each vertex is specified by a bipartite graph $B$ on which the left vertices represent the vertices of DAG $D$ and the right vertices are the stones.

If a vertex of $D$ correspond to the left vertex $v$ in $B$, then its neighbors describe which stones are allowed for it.

The vertices in $D$ do not need to have the same name as the one on the left side of $B$. It is only important that the number of vertices in $D$ is the same as the vertices in the left side of $B$.

In that case the element at position $i$ in the ordered sequence `enumerate_vertices(D)` corresponds to the element of rank $i$ in the sequence of left side vertices of $B$ according to the output of `Left, Right = bipartite_sets(B)`.

Standard *StoneFormula()* is essentially equivalent to a sparse stone formula where $B$ is the complete graph.

**Parameters**

**D** [a directed acyclic graph] it should be a directed acyclic graph.

**B** [bipartite graph]

**Raises**

**ValueError** if $D$ is not a directed acyclic graph

**ValueError** if $B$ is not a bipartite graph

**ValueError** when size differs between $D$ and the left side of $B$

**See also:**

*StoneFormula*

### Methods

| | |
|---|---|
| *build_cnf*(args) | Build the pebbling formula |
| *setup_command_line*(parser) | Setup the command line options for stone formulas |

**static build_cnf**(*args*)
Build the pebbling formula

Arguments: - *args*: command line options

**description = 'stone formula (sparse version)'**

**name = 'stonesparse'**

**static setup_command_line**(*parser*)
Setup the command line options for stone formulas

Arguments: - *parser*: parser to load with options.

**SparseStoneFormula**(*D*, *B*)
Sparse Stone formulas

This is a variant of the *StoneFormula()*. See that for a description of the formula. This variant is such that each vertex has only a small selection of which stone can go to that vertex. In particular which stones are allowed on each vertex is specified by a bipartite graph $B$ on which the left vertices represent the vertices of DAG $D$ and the right vertices are the stones.

If a vertex of $D$ correspond to the left vertex $v$ in $B$, then its neighbors describe which stones are allowed for it.

The vertices in $D$ do not need to have the same name as the one on the left side of $B$. It is only important that the number of vertices in $D$ is the same as the vertices in the left side of $B$.

In that case the element at position $i$ in the ordered sequence `enumerate_vertices(D)` corresponds to the element of rank $i$ in the sequence of left side vertices of $B$ according to the output of `Left, Right = bipartite_sets(B)`.

Standard *StoneFormula()* is essentially equivalent to a sparse stone formula where $B$ is the complete graph.

**Parameters**

**D** [a directed acyclic graph] it should be a directed acyclic graph.

**B** [bipartite graph]

**Raises**

**ValueError** if $D$ is not a directed acyclic graph

**ValueError** if $B$ is not a bipartite graph

**ValueError** when size differs between $D$ and the left side of $B$

**See also:**

*StoneFormula*

**class StoneCmdHelper**

Stone formulas

The stone formulas have been introduced in *[2]* and generalized in *[1]*. They are one of the classic examples that separate regular resolutions from general resolution *[1]*.

A "Stones formula" from a directed acyclic graph $D$ claims that each vertex of the graph is associated with one on $s$ stones (not necessarily in an injective way). In particular for each vertex $v$ in $V(D)$ and each stone $j$ we have a variable $P_{v,j}$ that claims that stone $j$ is associated to vertex $v$.

Each stone can be either red or blue, and not both. The propositional variable $R_j$ if true when the stone $j$ is red and false otherwise.

The clauses of the formula encode the following constraints. If a stone is on a source vertex (i.e. a vertex with no incoming edges), then it must be red. If all stones on the predecessors of a vertex are red, then the stone of the vertex itself must be red.

The formula furthermore enforces that the stones on the sinks (i.e. vertices with no outgoing edges) are blue.

---

**Note:** The exact formula structure depends by the graph and on its topological order, which is determined by the `enumerate_vertices(D)`.

---

**Parameters**

**D** [a directed acyclic graph] it should be a directed acyclic graph.

**nstones** [int] the number of stones.

**Raises**

**ValueError** if $D$ is not a directed acyclic graph

**ValueError** if the number of stones is negative

### References

*[1]*, *[2]*

### Methods

| | |
|---|---|
| *build_cnf*(args) | Build the pebbling formula |
| *setup_command_line*(parser) | Setup the command line options for stone formulas |

**static build_cnf**(*args*)

Build the pebbling formula

Arguments: - *args*: command line options

**description = 'stone formula'**

**name = 'stone'**

**static setup_command_line**(*parser*)

Setup the command line options for stone formulas

Arguments: - *parser*: parser to load with options.

**StoneFormula**(*D*, *nstones*)

Stone formulas

The stone formulas have been introduced in *[2]* and generalized in *[1]*. They are one of the classic examples

that separate regular resolutions from general resolution *[1]*.

A "Stones formula" from a directed acyclic graph $D$ claims that each vertex of the graph is associated with one on $s$ stones (not necessarily in an injective way). In particular for each vertex $v$ in $V(D)$ and each stone $j$ we have a variable $P_{v,j}$ that claims that stone $j$ is associated to vertex $v$.

Each stone can be either red or blue, and not both. The propositional variable $R_j$ if true when the stone $j$ is red and false otherwise.

The clauses of the formula encode the following constraints. If a stone is on a source vertex (i.e. a vertex with no incoming edges), then it must be red. If all stones on the predecessors of a vertex are red, then the stone of the vertex itself must be red.

The formula furthermore enforces that the stones on the sinks (i.e. vertices with no outgoing edges) are blue.

---

**Note:** The exact formula structure depends by the graph and on its topological order, which is determined by the `enumerate_vertices(D)`.

---

**Parameters**

**D** [a directed acyclic graph] it should be a directed acyclic graph.

**nstones** [int] the number of stones.

**Raises**

**ValueError** if $D$ is not a directed acyclic graph

**ValueError** if the number of stones is negative

### References

*[1]*, *[2]*

**stone_formula_helper**(*F*, *D*, *mapping*)

Stones formulas helper

Builds the clauses of a stone formula given the mapping object between stones and vertices of the DAG. THis is not supposed to be called by user code, and indeed this function assumes the following facts.

- $D$ is equal to *mapping.domain()*
- $D$ is a DAG

**Parameters**

**F** [CNF] the CNF which will contain the clauses of the stone formula

**D** [a directed acyclic graph] it should be a directed acyclic graph.

**mapping** [unary_mapping object] mapping between stones and graph vertices

**See also:**

***StoneFormula*** the classic stone formula

***SparseStoneFormula*** stone formula with sparse mapping

### 5.1.6 cnfformula.families.pigeonhole module

Implementation of the pigeonhole principle formulas

**class BPHPCmdHelper**
Bases: object

Command line helper for the Pigeonhole principle CNF

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a PHP formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for pigeonhole principle formula |

**static build_cnf**(*args*)
Build a PHP formula according to the arguments

Arguments: - *args*: command line options

**description = 'binary pigeonhole principle'**

**name = 'bphp'**

**static setup_command_line**(*parser*)
Setup the command line options for pigeonhole principle formula

Arguments: - *parser*: parser to load with options.

**BinaryPigeonholePrinciple**(*pigeons*, *holes*)
Binary Pigeonhole Principle CNF formula

The pigeonhole principle claims that no M pigeons can sit in N pigeonholes without collision if M>N. This formula encodes the principle using binary strings to identify the holes.

**Parameters**

**pigeon** [int] number of pigeons

**holes** [int] number of holes

**class GPHPCmdHelper**
Command line helper for the Pigeonhole principle on graphs

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a Graph PHP formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for pigeonhole principle formula over graphs |

**static build_cnf**(*args*)
Build a Graph PHP formula according to the arguments

Arguments: - *args*: command line options

**description = 'graph pigeonhole principle'**

**name = 'gphp'**

**static setup_command_line**(*parser*)
Setup the command line options for pigeonhole principle formula over graphs

Arguments: - *parser*: parser to load with options.

**GraphPigeonholePrinciple**(*graph*, *functional=False*, *onto=False*)
Graph Pigeonhole Principle CNF formula

The graph pigeonhole principle CNF formula, defined on a bipartite graph G=(L,R,E), claims that there is a subset E' of the edges such that every vertex on the left size L has at least one incident edge in E' and every edge on the right side R has at most one incident edge in E'.

This is possible only if the graph has a matching of size |L|.

There are different variants of this formula, depending on the values of *functional* and *onto* argument.

- PHP(G): each left vertex can be incident to multiple edges in E'
- FPHP(G): each left vertex must be incident to exaclty one edge in E'
- onto-PHP: all right vertices must be incident to some vertex
- matching: E' must be a perfect matching between L and R

Arguments: - *graph* : bipartite graph - *functional*: add clauses to enforce at most one edge per left vertex - *onto*: add clauses to enforce that any right vertex has one incident edge

Remark: the graph vertices must have the 'bipartite' attribute set. Left vertices must have it set to 0 and the right ones to 1. A KeyException is raised otherwise.

**class PHPCmdHelper**
Bases: object

Command line helper for the Pigeonhole principle CNF

### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a PHP formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for pigeonhole principle formula |

**static build_cnf**(*args*)
Build a PHP formula according to the arguments

Arguments: - *args*: command line options

**description = 'pigeonhole principle'**

**name = 'php'**

**static setup_command_line**(*parser*)
Setup the command line options for pigeonhole principle formula

Arguments: - *parser*: parser to load with options.

**PigeonholePrinciple**(*pigeons*, *holes*, *functional=False*, *onto=False*)
Pigeonhole Principle CNF formula

The pigeonhole principle claims that no M pigeons can sit in N pigeonholes without collision if M>N. The counterpositive CNF formulation requires such mapping to be satisfied. There are different variants of this formula, depending on the values of *functional* and *onto* argument.

- PHP: pigeon can sit in multiple holes
- FPHP: each pigeon sits in exactly one hole
- **onto-PHP: pigeon can sit in multiple holes, every hole must be** covered.
- Matching: one-to-one bijection between pigeons and holes.

Arguments: - *pigeon*: number of pigeons - *hole*: number of holes - *functional*: add clauses to enforce at most one hole per pigeon - *onto*: add clauses to enforce that any hole must have a pigeon

```
>>> print(PigeonholePrinciple(4,3).dimacs(export_header=False))
p cnf 12 22
1 2 3 0
4 5 6 0
7 8 9 0
10 11 12 0
-1 -4 0
-1 -7 0
-1 -10 0
-4 -7 0
-4 -10 0
-7 -10 0
-2 -5 0
-2 -8 0
-2 -11 0
-5 -8 0
-5 -11 0
-8 -11 0
-3 -6 0
-3 -9 0
-3 -12 0
-6 -9 0
-6 -12 0
-9 -12 0
```

### 5.1.7 cnfformula.families.ramsey module

CNF Formulas for Ramsey-like statements

**class PTNCmdHelper**

Bases: object

Command line helper for PTN formulas

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a Ramsey formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for PTN formula |

**static build_cnf**(*args*)

Build a Ramsey formula according to the arguments

Arguments: - *args*: command line options

**description = 'Bicoloring of N with no monochromatic Pythagorean Triples'**

**name = 'ptn'**

**static setup_command_line**(*parser*)

Setup the command line options for PTN formula

Arguments: - *parser*: parser to load with options.

**PythagoreanTriples**(*N*)

There is a Pythagorean triples free coloring on N

The formula claims that it is possible to bicolor the numbers from 1 to $N$ so that there is no monochromatic

triplet $(x, y, z)$ so that $x^2 + y^2 = z^2$.

**Parameters**

**N** [int] size of the interval

**Raises**

**ValueError** Parameters are not positive integers

#### References

*[1]*

**class RamseyCmdHelper**

Bases: object

Command line helper for RamseyNumber formulas

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a Ramsey formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for Ramsey formula |

**static build_cnf**(*args*)

Build a Ramsey formula according to the arguments

Arguments: - *args*: command line options

**description = 'ramsey number principle'**

**name = 'ram'**

**static setup_command_line**(*parser*)

Setup the command line options for Ramsey formula

Arguments: - *parser*: parser to load with options.

**RamseyLowerBoundFormula**(*s*, *k*, *N*)

Formula claiming that Ramsey number r(s,k) > N

Arguments: - *s*: independent set size - *k*: clique size - *N*: vertices

### 5.1.8 cnfformula.families.randomformulas module

Random CNF Formulas

**class RandCmdHelper**

Bases: object

Command line helper for random formulas

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a conjunction |
| *setup_command_line*(parser) | Setup the command line options for an and of literals |

**static build_cnf**(*args*)
Build a conjunction

Arguments: - *args*: command line options

**description = 'random k-CNF'**

**name = 'randkcnf'**

**static setup_command_line**(*parser*)
Setup the command line options for an and of literals

Arguments: - *parser*: parser to load with options.

**RandomKCNF**(*k*, *n*, *m*, *seed=None*, *planted_assignments=[]*)
Build a random k-CNF

Sample $m$ clauses over $n$ variables, each of width $k$, uniformly at random. The sampling is done without repetition, meaning that whenever a randomly picked clause is already in the CNF, it is sampled again.

**Parameters**

**k** [int] width of each clause

**n** [int] number of variables to choose from. The resulting CNF object will contain n variables even if some are not mentioned in the clauses.

**m** [int] number of clauses to generate

**seed** [hashable object] seed of the random generator

**planted_assignments** [iterable(dict), optional] a set of total/partial assigments such that all clauses in the formula will be satisfied by all of them.

**Returns**

**a CNF object**

**Raises**

**ValueError** when some paramenter is negative, or when k>n.

**all_clauses**(*k*, *indices*, *planted_assignments*)

**clause_satisfied**(*cls*, *assignments*)
Test whether a clause is satisfied by all assignments

Test if clauses *cls* is satisfied by all assigment in the list assignments.

**sample_clauses**(*k*, *indices*, *m*, *planted_assignments*)

**sample_clauses_dense**(*k*, *indices*, *m*, *planted_assignments*)

### 5.1.9 cnfformula.families.subgraph module

Implementation of formulas that check for subgraphs

**BinaryCliqueFormula**(*G*, *k*)
Test whether a graph has a k-clique.

Given a graph $G$ and a non negative value $k$, the CNF formula claims that $G$ contains a $k$-clique. This formula uses the binary encoding, in the sense that the clique elements are indexed by strings of bits.

**Parameters**

**G** [networkx.Graph] a simple graph

**k** [a non negative integer] clique size

**Returns**

**a CNF object**

**class BinaryKCliqueCmdHelper**

Bases: object

Command line helper for k-clique formula

### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a k-clique formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for k-clique formula |

**static build_cnf**(*args*)

Build a k-clique formula according to the arguments

Arguments: - *args*: command line options

**description = 'Binary k clique formula'**

**name = 'kcliquebin'**

**static setup_command_line**(*parser*)

Setup the command line options for k-clique formula

Arguments: - *parser*: parser to load with options.

**CliqueFormula**($G$, $k$)

Test whether a graph has a k-clique.

Given a graph $G$ and a non negative value $k$, the CNF formula claims that $G$ contains a $k$-clique.

**Parameters**

**G** [networkx.Graph] a simple graph

**k** [a non negative integer] clique size

**Returns**

**a CNF object**

**class KCliqueCmdHelper**

Bases: object

Command line helper for k-clique formula

### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a k-clique formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for k-clique formula |

**static build_cnf**(*args*)

Build a k-clique formula according to the arguments

Arguments: - *args*: command line options

**description = 'k clique formula'**

**name = 'kclique'**

**static setup_command_line**(*parser*)

Setup the command line options for k-clique formula

Arguments: - *parser*: parser to load with options.

**class RWCmdHelper**

Bases: object

Command line helper for ramsey graph formula

### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a formula to check that a graph is a ramsey number lower bound |
| *setup_command_line*(parser) | Setup the command line options for ramsey witness formula |

**static build_cnf**(*args*)

Build a formula to check that a graph is a ramsey number lower bound

Arguments: - *args*: command line options

**description = 'unsat if G witnesses that r(k,s)>|V(G)| (i.e. G has not k-clique nor**

**name = 'ramlb'**

**static setup_command_line**(*parser*)

Setup the command line options for ramsey witness formula

Arguments: - *parser*: parser to load with options.

**RamseyWitnessFormula**(*G*, *k*, *s*)

Test whether a graph contains one of the templates.

Given a graph $G$ and a non negative values $k$ and $s$, the CNF formula claims that $G$ contains a neither a $k$-clique nor an independet set of size $s$.

**Parameters**

**G** [networkx.Graph] a simple graph

**k** [a non negative integer] clique size

**s** [a non negative integer] independet set size

**Returns**

**a CNF object**

**class SubGraphCmdHelper**

Bases: object

Command line helper for Graph Isomorphism formula

### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a graph automorphism formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for graph isomorphism formula |

**static build_cnf**(*args*)

Build a graph automorphism formula according to the arguments

Arguments: - *args*: command line options

**description = 'subgraph formula'**

**name = 'subgraph'**

**static setup_command_line**(*parser*)
Setup the command line options for graph isomorphism formula

Arguments: - *parser*: parser to load with options.

**SubgraphFormula**(*graph*, *templates*, *symmetric=False*)
Test whether a graph contains one of the templates.

Given a graph $G$ and a sequence of template graphs $H_1$, $H_2$, ..., $H_t$, the CNF formula claims that $G$ contains an isomorphic copy of at least one of the template graphs.

E.g. when $H_1$ is the complete graph of $k$ vertices and it is the only template, the formula claims that $G$ contains a $k$-clique.

**Parameters**

**graph** [networkx.Graph] a simple graph

**templates** [list-like object] a sequence of graphs.

**symmetric:** all template graphs are symmetric wrt permutations of vertices. This allows some optimization in the search space of the assignments.

**induce:** force the subgraph to be induced (i.e. no additional edges are allowed)

**Returns**

**a CNF object**

### 5.1.10 cnfformula.families.subsetcardinality module

Implementation of subset cardinality formulas

**class SCCmdHelper**
Bases: object

#### Methods

| | |
|---|---|
| **build_cnf** | |
| **setup_command_line** | |

**static build_cnf**(*args*)

**description = 'subset cardinality formulas'**

**name = 'subsetcard'**

**static setup_command_line**(*parser*)

**SubsetCardinalityFormula**(*B*, *equalities=False*)
Consider a bipartite graph $B$. The CNF claims that at least half of the edges incident to each of the vertices on left side of $B$ must be zero, while at least half of the edges incident to each vertex on the left side must be one.

Variants of these formula on specific families of bipartite graphs have been studied in *[1]*, *[2]* and *[3]*, and turned out to be difficult for resolution based SAT-solvers.

Each variable of the formula is denoted as $x_{i,j}$ where $\{i, j\}$ is an edge of the bipartite graph. The clauses of the CNF encode the following constraints on the edge variables.

For every left vertex i with neighborhood $\Gamma(i)$

$$\sum_{j \in \Gamma(i)} x_{i,j} \geq \frac{|\Gamma(i)|}{2}$$

For every right vertex j with neighborhood $\Gamma(j)$

$$\sum_{i \in \Gamma(j)} x_{i,j} \leq \frac{|\Gamma(j)|}{2}.$$

If the `equalities` flag is true, the constraints are instead represented by equations.

$$\sum_{j \in \Gamma(i)} x_{i,j} = \left\lceil \frac{|\Gamma(i)|}{2} \right\rceil$$

$$\sum_{i \in \Gamma(j)} x_{i,j} = \left\lfloor \frac{|\Gamma(j)|}{2} \right\rfloor.$$

**Parameters**

**B** [networkx.Graph] the graph vertices must have the 'bipartite' attribute set. Left vertices must have it set to 0 and the right ones to 1. A KeyException is raised otherwise.

**equalities** [boolean] use equations instead of inequalities to express the cardinality constraints. (default: False)

**Returns**

**A CNF object**

#### References

*[1]*, *[2]*, *[3]*

### 5.1.11 cnfformula.families.cliquecoloring module

Implementation of the clique-coloring formula

**CliqueColoring**($n$, $k$, $c$)

Clique-coloring CNF formula

The formula claims that a graph $G$ with $n$ vertices simultaneously contains a clique of size $k$ and a coloring of size $c$.

If $k = c + 1$ then the formula is clearly unsatisfiable, and it is the only known example of a formula hard for cutting planes proof system. *[1]*

Variables $e_{u,v}$ to encode the edges of the graph.

Variables $q_{i,v}$ encode a function from $[k]$ to $[n]$ that represents a clique.

Variables $r_{v,\ell}$ encode a function from $[n]$ to $[c]$ that represents a coloring.

**Parameters**

**n** [number of vertices in the graph]

**k** [size of the clique]

**c** [size of the coloring]

**Returns**

**A CNF object**

#### References

*[1]*

**class CliqueColoringCmdHelper**
Bases: object

Command line helper for the Clique-coclique CNF

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a Clique-coclique formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for clique-coloring formula |

**static build_cnf**(*args*)
Build a Clique-coclique formula according to the arguments

Arguments: - *args*: command line options

**description = 'There is a graph G with a k-clique and a c-coloring'**

**name = 'cliquecoloring'**

**static setup_command_line**(*parser*)
Setup the command line options for clique-coloring formula

Arguments: - *parser*: parser to load with options.

### 5.1.12 cnfformula.families.tseitin module

Implementation of Tseitin formulas

**class TseitinCmdHelper**
Bases: object

Command line helper for Tseitin formulas

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build Tseitin formula according to the arguments |
| *setup_command_line*(parser) | Setup the command line options for Tseitin formula |

**static build_cnf**(*args*)
Build Tseitin formula according to the arguments

Arguments: - *args*: command line options

**description = 'tseitin formula'**

**name = 'tseitin'**

**static setup_command_line**(*parser*)
Setup the command line options for Tseitin formula

Arguments: - *parser*: parser to load with options.

**TseitinFormula**(*graph*, *charges=None*)

Build a Tseitin formula based on the input graph.

Odd charge is put on the first vertex by default, unless other vertices are is specified in input.

Arguments: - *graph*: input graph - 'charges': odd or even charge for each vertex

### 5.1.13 cnfformula.families.simple module

Implementation of simple formulas

**class AND**

Bases: object

Command line helper for a 1-CNF (i.e. conjunction)

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a conjunction |
| *setup_command_line*(parser) | Setup the command line options for an and of literals |

**static build_cnf**(*args*)

Build a conjunction

Arguments: - *args*: command line options

**description = 'a single conjunction'**

**name = 'and'**

**static setup_command_line**(*parser*)

Setup the command line options for an and of literals

Arguments: - *parser*: parser to load with options.

**class EMPTY**

Bases: object

Command line helper for the empty CNF (no clauses)

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build an empty CNF formula |

| setup_command_line | |
|---|---|

**static build_cnf**(*args*)

Build an empty CNF formula

**Parameters**

**args** [ignored] command line options

**description = 'empty CNF formula'**

**name = 'empty'**

**static setup_command_line**(*parser*)

**class EMPTY_CLAUSE**

Bases: object

Command line helper for the contradiction (one empty clauses)

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build a CNF formula with an empty clause |

**setup_command_line**

**static build_cnf**(*args*)
Build a CNF formula with an empty clause

**Parameters**

**args** [ignored] command line options

**description = 'one empty clause'**

**name = 'emptyclause'**

**static setup_command_line**(*parser*)

**class OR**
Bases: object

Command line helper for a single clause formula

#### Methods

| | |
|---|---|
| *build_cnf*(args) | Build an disjunction |
| *setup_command_line*(parser) | Setup the command line options for single or of literals |

**static build_cnf**(*args*)
Build an disjunction

Arguments: - *args*: command line options

**description = 'a single disjunction'**

**name = 'or'**

**static setup_command_line**(*parser*)
Setup the command line options for single or of literals

Arguments: - *parser*: parser to load with options.

## 5.2 Command line invocation

Furthermore it is possible to generate the formulas directly from the command line. To list all formula families accessible from the command line just run the command `cnfgen --help`. To get information about the specific command line parameters for a formula generator run the command `cnfgen <generator_name> --help`.

Recall the example above, in hich we produced a pigeohole principle formula for 10 pigeons and 9 holes. We can get the same formula in DIMACS format with the following command line.

```
$ cnfgen php 10 9
c Pigeonhole principle formula for 10 pigeons and 9 holes
```

(continues on next page)

(continued from previous page)

```
c Generated with `cnfgen` (C) 2012-2016 Massimo Lauria <lauria.massimo@gmail.com>
c https://massimolauria.github.io/cnfgen
c
p cnf 90 415
1 2 3 4 5 6 7 8 9 0
10 11 12 13 14 15 16 17 18 0
19 20 21 22 23 24 25 26 27 0
28 29 30 31 32 33 34 35 36 0
37 38 39 40 41 42 43 44 45 0
46 47 48 49 50 51 52 53 54 0
55 56 57 58 59 60 61 62 63 0
64 65 66 67 68 69 70 71 72 0
73 74 75 76 77 78 79 80 81 0
82 83 84 85 86 87 88 89 90 0
-1 -10 0
-1 -19 0
-1 -28 0
-1 -37 0
-1 -46 0
-1 -55 0
-1 -64 0
-1 -73 0
-1 -82 0
-10 -19 0
-10 -28 0
-10 -37 0
-10 -46 0
...
-72 -81 0
-72 -90 0
-81 -90 0
```

# CHAPTER 6

# Graph based formulas

The most interesting benchmark formulas have a graph structure. See the following example, where `cnfformula.TseitinFormula()` is realized over a star graph with five arms.

```
>>> import cnfformula
>>> import networkx as nx
>>> G = nx.star_graph(5)
>>> G.edges()
>>> [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5)]
>>> F = cnfformula.TseitinFormula(G,charges=[0,1,1,0,1,1])
>>> F.is_satisfiable()
(True,
 {'E_{0,1}': True,
  'E_{0,2}': True,
  'E_{0,3}': False,
  'E_{0,4}': True,
  'E_{0,5}': True})
```

Tseitin formulas can be really hard for if the graph has large edge expansion. Indeed the unsatisfiable version of this formula requires exponential running time in any resolution based SAT solver[1].

In the previous example the structure of the CNF was a simple undirected graph, but in `CNFgen` we have formulas built around four different types of graphs.

| | | |
|---|---|---|
| `simple` | simple graph | default graph |
| `bipartite` | bipartite graph | vertices split in two inpedendent sets |
| `digraph` | directed graph | each edge has an orientation |
| `dag` | directed acyclic graph | no cycles, edges induce a partial ordering |

To manipulate graphs `CNFgen` does not reinvent the wheel, but uses the famous NetworkX library behind the scene. `NetworkX` supports reading and writing graph from/to files, and `CNFgen` leverages on that. Furthermore `CNFgen` implements graph I/O in few other file formats. The function `cnfformula.graphs.supported_formats()` lists the file formats available for each graph type.

```
>>> from cnfformula.graphs import supported_formats
>>> supported_formats()
{'bipartite': ['matrix', 'gml', 'dot'],
```

(continues on next page)

[1] A. Urquhart. *Hard examples for resolution*. Journal of the ACM (1987) http://dx.doi.org/10.1145/48014.48016

(continued from previous page)

```
 'dag': ['kthlist', 'gml', 'dot'],
 'digraph': ['kthlist', 'gml', 'dot', 'dimacs'],
 'simple': ['kthlist', 'gml', 'dot', 'dimacs']}
```

The `dot` format is from Graphviz and it is available only if the optional `pydot2` python package is installed in the system. The Graph Modelling Language (GML) `gml` is the current standard in graph representation. The DIMACS (`dimacs`) format[2] is used sometimes for programming competitions or in the theoretical computer science community. The `kthlist` and `matrix` formats are defined and implemented inside `CNFgen`.

**Note:** More information about the supported graph file formats is in the User Documentation for the `cnfgen` command line too. In particular there is a description of `kthlist` and `matrix` formats.

## 6.1 Directed Acyclic Graphs and Bipartite Graphs

`NetworkX` does not have a specific data structure for a directed acyclic graphs (DAGs). In `CNFgen` a DAG is any object which is either a `networkx.DiGraph` or `networkx.MultiDiGraph` instance, and which furthermore passes the test `cnfformula.graphs.is_dag()`.

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_cycle([1,2,3])
>>> cnfformula.graphs.is_dag(G)
False
>>> H = nx.DiGraph()
>>> H.add_path([1,2,3])
>>> cnfformula.graphs.is_dag(H)
True
```

In the same way `NetworkX` does not have a particular data structure for bipartite graphs ( `networkx.Graph` and `networkx.MultiGraph` are possible choices), but it follows the convention that all vertices in the graph have the `bipartite` attribute that gets values $\{0,1\}$[3]. The CNF formula constructions that make use of bipartite graphs usually associate the 0 part as the left side, and the 1 part to the right side. The function `cnfformula.graphs.has_bipartition()` tests whether this bipartition exists in a graph.

```
>>> import networkx as nx
>>> G=nx.bipartite.random_graph(3,2,0.5)
>>> cnfformula.graphs.has_biparition(G)
True
>>> G.node
{0: {'bipartite': 0},
 1: {'bipartite': 0},
 2: {'bipartite': 0},
 3: {'bipartite': 1},
 4: {'bipartite': 1}}
>>> G.edges()
[(0, 4), (1, 3), (1, 4), (2, 3)]
>>> F = cnfformula.GraphPigeonholePrinciple(G)
>>> list(F.variables())
['p_{0,4}', 'p_{1,3}', 'p_{1,4}', 'p_{2,3}']
```

[2] Beware. Here we are talking about the DIMACS format for graphs, not the DIMACS file format for CNF formulas.
[3] This convention is describe in http://networkx.readthedocs.org/en/latest/reference/algorithms.bipartite.html

## 6.2 Graph I/O

The `cnfformula.graphs` module implements a graph reader `cnfformula.graphs.readGraph` and a graph writer `cnfformula.graphs.writeGraph` to facilitate graph I/O. .. Both `readGraph` and `writeGraph` operate either on filenames, encoded as `str` or `unicode`, or otherwise on file-like objects such as

- standard file objects (including `sys.stdin` and `sys.stdout`);
- string buffers of type `StringIO.StringIO`;
- in-memory text streams that inherit from `io.TextIOBase`.

```
>>> import sys
>>> import networkx as nx
>>> from cnfformula.graphs import readGraph, writeGraph
```

```
>>> G = nx.bipartite.random_graph(3,2,0.5)
>>> writeGraph(G,sys.stdout,graph_type='bipartite',file_format='gml')
strict graph "fast_gnp_random_graph(3,2,0.5)" {
 0    [bipartite=0];
      4        [bipartite=1];
      0 -- 4;
      1        [bipartite=0];
      3        [bipartite=1];
      1 -- 3;
      2        [bipartite=0];
      2 -- 4;
}
```

```
>>> from StringIO import StringIO
>>> buffer = StringIO("graph X { 1 -- 2 -- 3 }")
>>> G = readGraph(buffer, graph_type='simple', file_format='dot')
>>> G.edges()
[('1', '2'), ('3', '2')]
```

There are several advantages with using those functions, instead of the reader/writer implemented `NextowrkX`. First of all the reader always verifies that when reading a graph of a certain type, the actual input actually matches the type. For example if the graph is supposed to be a DAG, then `cnfformula.graphs.readGraph()` would check that.

```
>>> buffer = StringIO('digraph A { 1 -- 2 -- 3 -- 1}')
>>> readGraph(buffer,graph_type='dag',file_format='dot')
[...]
ValueError: Input graph must be acyclic
```

When the file object has an associated file name, it is possible to omit the `file_format` argument. In this latter case the appropriate choice of format will be guessed by the file extension.

```
>>> with open("example.dot","w") as f: print >> f , "digraph A {1->2->3}"
>>> G = readGraph("example.dot",graph_type='dag')
>>> G.edges()
[('1', '2'), ('2', '3')]
```

```
>>> with open("example.gml","w") as f: print >> f , "digraph A {1->2->3}"
>>> G = readGraph("example.gml",graph_type='dag',file_format='dot')
>>> G.edges()
[('1', '2'), ('2', '3')]
```

```
>>> with open("example.gml","w") as f: print >> f , "digraph A {1->2->3}"
>>> G = readGraph("example.gml",graph_type='dag')
ValueError: [Parse error in GML input] expected ...
```

## 6.3 Graph generators

**Note:** See the documentation of the module `cnfformula.graphs` for more information about the `CNFgen` support code for graphs.

## 6.4 References

CHAPTER 7

# Post-process a CNF formula

After you produce a `cnfformula.CNF`, maybe using one of the generators included, it is still possible to modify it. One simple ways is to add new clauses but there are ways to make the formula harder with some structured transformations. Usually this technique is employed to produce interesting formulas for proof complexity applications or to benchmark SAT solvers.

## 7.1 Example: `OR` substitution

As an example of formula post-processing, we transform a formula by substituting every variable with the logical disjunction of, says, 3 fresh variables. Consider the following CNF as starting point.

$$(\neg X \vee Y) \wedge (\neg Z)$$

After the substitution the new formula is still expressed as a CNF and it is

$$\begin{gathered}(\neg X_1 \vee Y_1 \vee Y_2 \vee Y_3) \wedge \\ (\neg X_2 \vee Y_1 \vee Y_2 \vee Y_3) \wedge \\ (\neg X_3 \vee Y_1 \vee Y_2 \vee Y_3) \wedge \\ (\neg Z_1) \wedge (\neg Z_2) \wedge (\neg Z_3)\end{gathered}$$

There are many other transformation methods than `OR` substitution. Each method comes with a *rank* parameter that controls the hardness after the substitution. In the previous example the parameter would be the number of variables used in the disjunction to substitute the original variables.

## 7.2 Using CNF transformations

To get a list of the available transformations you can run

```
>>> import cnfformula
>>> cnfformula.available_transformations()
```

The following processing methods are implemented. The `none` method just leaves the formula alone. It is a null transformation in the sense that, contrary to the other methods, this one returns exactly the same `cnfformula.CNF` object that it gets in input. All the other methods would produce a new CNF object with the new formula. The old one is left untouched.

| *Name* | Description | Default rank | See documentation |
|---|---|---|---|
| `none` | leaves the formula alone | ignored | |
| `eq` | all variables equal | 3 | `cnfformula.transformation.Equality` |
| `ite` | if x then y else z | ignored | `cnfformula.transformation.IfThenElse` |
| `lift` | lifting | 3 | `cnfformula.transformation.Lifting` |
| `maj` | Loose majority | 3 | `cnfformula.transformation.Majority` |
| `neq` | not all vars equal | 3 | `cnfformula.transformation.NotEquality` |
| `one` | Exactly one | 3 | `cnfformula.transformation.One` |
| `or` | OR substitution | 2 | `cnfformula.transformation.InnerOr` |
| `xor` | XOR substitution | 2 | `cnfformula.transformation.InnerXor` |

Any `cnfformula.CNF` can be post-processed using the function `cnfformula.TransformFormula()`. For example to substitute each variable with a 2-XOR we can do

```
>>> from cnfformula import CNF, TransformFormula
>>> F = CNF([ [(True,"x1"),(True,"x2"),(False,"x3")], [(False,"x2"),(True,"x4")] ])
>>> G = TransformFormula(F,'xor',2)
```

Here is the original formula.

```
>>> print F.dimacs(export_header=False)
p cnf 4 2
1 2 -3 0
-2 4 0
```

Here it is after the transformation.

```
>>> print G.variables(export_header=False)
p cnf 8 12
1 2 3 4 5 -6 0
1 2 3 4 -5 6 0
1 2 -3 -4 5 -6 0
1 2 -3 -4 -5 6 0
-1 -2 3 4 5 -6 0
-1 -2 3 4 -5 6 0
-1 -2 -3 -4 5 -6 0
-1 -2 -3 -4 -5 6 0
3 -4 7 8 0
3 -4 -7 -8 0
-3 4 7 8 0
-3 4 -7 -8 0
```

It is possible to omit the rank parameter. In such case the default value is used.

CHAPTER 8

# The command line utility

Most people are likely to use `CNFgen` by command line. The command line has a powerful interface with many options and sensible defaults, so that the newcomer is not intimidated but it is still possible to generate nontrivial formula

CHAPTER 9

# Adding a formula family to CNFgen

# CHAPTER 10

# Welcome to CNFgen's documentation!

The main components of CNFgen are the `cnfformula` library and the `cnfgen` command line utility.

## 10.1 The `cnfformula` library

The `cnfformula` library is capable to generate Conjunctive Normal Form (CNF) formulas, manipulate them and, when there is a satisfiability (SAT) solver properly installed on your system, test their satisfiability. The CNFs can be saved on file in DIMACS format, which the standard input format for SAT solvers[1], or converted to LaTeX[2] to be included in a document. The library contains many generators for formulas that encode various combinatorial problems or that come from research in Proof Complexity[3].

The main entry point for the library is the `cnfformula.CNF` object. Let's see a simple example of its usage.

```
>>> import cnfformula
>>> F = cnfformula.CNF()
>>> F.add_clause([(True,"X"),(False,"Y")])
>>> F.add_clause([(False,"X")])
>>> F.is_satisfiable()
(True, {'Y':False, 'X':False})
>>> F.add_clause([(True,"Y")])
>>> F.is_satisfiable()
(False, None)
>>> print F.dimacs()
c Generated with `cnfgen` (C) 2012-2016 Massimo Lauria <lauria.massimo@gmail.com>
c https://github.com/MassimoLauria/cnfgen.git
c
p cnf 2 3
1 -2 0
-1 0
2 0
>>> print F.latex()
% Generated with `cnfgen` (C) 2012-2016 Massimo Lauria <lauria.massimo@gmail.com>
% https://github.com/MassimoLauria/cnfgen.git
%
```

(continues on next page)

[1] http://www.satlib.org/Benchmarks/SAT/satformat.ps
[2] http://www.latex-project.org/
[3] http://en.wikipedia.org/wiki/Proof_complexity

(continued from previous page)

```
\begin{align}
&       \left(     {X} \lor \neg{Y} \right) \\
& \land \left( \neg{X} \right) \\
& \land \left(     {Y} \right)
\end{align}
```

A typical unsatisfiable formula studied in Proof Complexity is the pigeonhole principle formula.

```
>>> from cnfformula import PigeonholePrinciple
>>> F = PigeonholePrinciple(5,4)
>>> print F.dimacs()
c Pigeonhole principle formula for 5 pigeons and 4 holes
c Generated with `cnfgen` (C) 2012-2016 Massimo Lauria <lauria.massimo@gmail.com>
c https://github.com/MassimoLauria/cnfgen.git
c
p cnf 20 45
1 2 3 4 0
5 6 7 8 0
...
-16 -20 0
>>> F.is_satisfiable()
(False, None)
```

## 10.2 The `cnfgen` command line tool

The command line tool is installed along `cnfformula` package, and provides a somehow limited interface to the library capabilities. It provides ways to produce formulas in DIMACS and LaTeX format from the command line. To produce a pigeonhole principle from 5 pigeons to 4 holes as in the previous example the command line is

```
$ cnfgen php 5 4
c Pigeonhole principle formula for 5 pigeons and 4 holes
c Generated with `cnfgen` (C) Massimo Lauria <lauria.massimo@gmail.com>
c https://github.com/MassimoLauria/cnfgen.git
c
p cnf 20 45
1 2 3 4 0
5 6 7 8 0
...
-16 -20 0
```

For a documentation on how to use `cnfgen` command please type `cnfgen --help` and for further documentation about a specific formula generator type `cnfgen <generator_name> --help`.

## 10.3 Reference

CHAPTER 11

# Indices and tables

- genindex
- modindex
- search

# Bibliography


[1] Locality and Hard SAT-instances, Klas Markstrom Journal on Satisfiability, Boolean Modeling and Computation 2 (2006) 221-228

[1] M. Alekhnovich, J. Johannsen, T. Pitassi and A. Urquhart An Exponential Separation between Regular and General Resolution. Theory of Computing (2007)

[2] R. Raz and P. McKenzie Separation of the monotone NC hierarchy. Combinatorica (1999)



[1] M. Alekhnovich, J. Johannsen, T. Pitassi and A. Urquhart An Exponential Separation between Regular and General Resolution. Theory of Computing (2007)

[2] R. Raz and P. McKenzie Separation of the monotone NC hierarchy. Combinatorica (1999)



[1] M. J. Heule, O. Kullmann, and V. W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. arXiv preprint arXiv:1605.00723, 2016.

[1] Mladen Miksa and Jakob Nordstrom Long proofs of (seemingly) simple formulas Theory and Applications of Satisfiability Testing–SAT 2014 (2014)

[2] Ivor Spence sgen1: A generator of small but difficult satisfiability benchmarks Journal of Experimental Algorithmics (2010)

[3] Allen Van Gelder and Ivor Spence Zero-One Designs Produce Small Hard SAT Instances Theory and Applications of Satisfiability Testing–SAT 2010(2010)

[1] Pavel Pudlak. Lower bounds for resolution and cutting plane proofs and monotone computations. Journal of Symbolic Logic (1997)

# Python Module Index

## f

# Index

## E



## G



## K



## N



## O



## P



## R

## S



## T



## V