CNFgen Documentation

Release 70e5b68

Massimo Lauria

Apr 17, 2021

Table of contents

The entry point to `cnfgen` library is `cnfgen.CNF`, which is the data structure representing CNF formulas. Variables can have text names but in general each variable is an integer from 1 to n, where n is the number of variables.

```
>>> from cnfgen import CNF
>>> F = CNF()
>>> x = F.new_variable("X")
>>> y = F.new_variable("Y")
>>> z = F.new_variable("Z")
>>> print(x,y,z)
1 2 3
```

A clause is a list of literals, and each literal is `+v` or `-v` where `v` is the number corresponding to a variable. The user can interleave the addition of variables and clauses. Notice that the method **:py:method:‘new_variable‘** return the numeric id of the newly added variable, which can be optionally used to build clauses.

```
>>> F.add_clause([-x, y])
>>> w = F.new_variable("W")
>>> w == 4
True
>>> F.add_clause([-z, 4])
>>> F.number_of_variables()
4
>>> F.number_of_clauses()
2
```

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

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

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.

```
>>> G = CNF()
>>> G.number_of_variables()
0
>>> G.add_clause([-1, 2])
>>> G.number_of_variables()
2
>>> list(G.variables())
[1, 2]
```

Note: By default the **:py:method:‘cnfgen.CNF.add_clause‘** checks that all literals in the clauses are non-zero integers. Furthermore if there are new variables mentioned in the clause, the number of variables of the formula is automatically updated. This checks makes adding clauses a bit expensive, and that’s an issue for very large formulas where millions of clauses are added. It is possible to avoid such checks but then it is resposibility of the user to keep things consistent.

See also **:py:method:‘cnfgen.CNF.debug‘**, which in turn can be also used to check the presence of literal repetitions and opposite literals.

It is possible to add clauses directly at the CNF construction. The code

```
>>> H = CNF([ [1,2,-3], [-2,4] ])
```

is essentially equivalent to

```
>>> H = CNF()
>>> H.add_clauses_from([ [1,2,-3], [-2,4] ])
```

or

```
>>> H = CNF()
>>> H.add_clause([1,2,-3])
>>> H.add_clause([-2,4])
```

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 `cnfgen.CNF.to_dimacs()`.

```
>>> c=CNF([ [1,2,-3], [-2,4] ])
>>> print( c.to_dimacs() )
p cnf 4 2
1 2 -3 0
-2 4 0
<BLANKLINE>
>>> c.add_clause( [-3,4,-5] )
>>> print( c.to_dimacs() )
p cnf 5 3
1 2 -3 0
-2 4 0
-3 4 -5 0
<BLANKLINE>
```

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.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps

CHAPTER 2

Exporting formulas to LaTeX

It is possible to use `cnfgen.CNF.to_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. By default variables `i` has the assigned name `x_{i}`.

```
>>> c=CNF([[-1, 2, -3], [-2,-4], [2,3,-4]])
>>> print(c.to_latex())
\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

$$\begin{align} & \left(\overline{x}_1 \lor x_2 \lor \overline{x}_3 \right) \tag{2.1} \\ & \land \left(\overline{x}_2 \lor \overline{x}_4 \right) \tag{2.2} \\ & \land \left(x_2 \lor x_3 \lor \overline{x}_4 \right) \tag{2.3} \end{align}$$

Instead of outputting just the LaTeX rendering of the formula it is possible to produce a full LaTeX document by using `cnfgen.CNF.to_latex_document()`. 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 `cnfgen.CNF` instance we can use the `cnfgen.CNF.is_satisfiable()` and `cnfgen.CNF.solve()` methods. The former just gives a boolean answer, with the latter provides more options about how to run the solver and returns a satisfying assignment found in the process.

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 cnfgen import CNF
>>> F = CNF([ [1,-2], [-1] ])
>>> outcome,assignment = F.solve()
>>> outcome
True
>>> assignment == [-1,-2]
True
>>> F.add_clause([2])
>>> F.is_satisfiable()
False
```

It is always possible to force `CNFgen` to use a specific solver or a specific command line invocation using the `cmd` parameters for `cnfgen.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') # doctest: +SKIP
>>> F.is_satisfiable(cmd='glucose -pre') # doctest: +SKIP
>>> F.is_satisfiable(cmd='lingeling --plain') # doctest: +SKIP
>>> F.is_satisfiable(cmd='sat4j') # doctest: +SKIP
```

(continues on next page)

[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.

(continued from previous page)

```
>>> F.is_satisfiable(cmd='my-hacked-minisat -pre',sameas='minisat') # doctest:␣
↪+SKIP
>>> F.is_satisfiable(cmd='patched-lingeling',sameas='lingeling') # doctest: +SKIP
```

CHAPTER 5

Formula families

The defining 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 `cnfgen` package. See for example this construction of the pigeohole principle formula with 5 pigeons and 4 holes.

```
>>> import cnfgen
>>> F = cnfgen.PigeonholePrinciple(5,4)
>>> F.is_satisfiable()
False
```

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 cnfgen.families.counting module

Implementation of counting/matching formulas

CountingPrinciple(M, p)

Counting principle

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

Parameters

M [non negative integer] size of the domain

p [positive integer] size of each part

Returns

cnfgen.CNF

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 cnfgen.families.coloring module

Formulas that encode coloring related problems

`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 [cnfgen.Graph] a simple undirected graph

colors [non negative int] the number of colors

functional: bool forbid a vertex to be mapped to multiple colors

Returns

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

5.1.3 cnfgen.families.graphisomorphism module

Graph isomorphimsm/automorphism formulas

`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*, *nontrivial=False*)
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

nontrivial: bool forbid identical mapping

Returns

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

are isomorphic.

5.1.4 cnfgen.families.ordering module

Implementation of the ordering principle formulas

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)

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)

5.1.5 cnfgen.families.pebbling module

Implementation of the pigeonhole principle formulas

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.

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`

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.

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]

5.1.6 cnfgen.families.pigeonhole module

Pigeonhole principle formulas

The pigeonhole principle PHP^m_n, written in conjunctive normal form, is a propositional formula which claims that it is possible to place m pigeons into n holes without collisions, whenever $m > n$.

Pigeonhole principle formulas are classic benchmarks for SAT solving and for Resolution proof systems. The module contains the implementation of several variants of this formulas.

The most classic pigeonhole principle formula PHP^{n+1}_{n} was the first CNF proved to be hard for resolution [H85].

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

The binary pigeonhole principle CNF formula claims that that it is possibile to place m pigeons into n holes without collisions. This is clearly impossible whenever $m > n$.

This formula encodes the principle using binary strings to identify the holes. Let b the smallest number of bits sufficient to encode in binary all values from 0 to $n - 1$. For every $i \in [m]$ there are b dedicated boolean variables encoding the hole where the pigeon i flies.

Parameters

pigeon: int number of pigeons (must be >=0).

hole: int number of holes (must be >=0).

Returns

cnfgen.formula.cnf.CNF A CNF formulas encoding binary the pigeonhole principle.

Raises

TypeError If either *pigeons* or *holes* is not an integer number.

ValueError If either *pigeons* or *holes* is less than zero.

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

The graph pigeonhole principle CNF formula, defined on a bipartite graph $G = (L, R, E)$, is a variant of the pigeonhole principle where the left vertices L are the pigeons, the right vertices R are the holes. The formula claims that there is a subset of edges $E' \subseteq E$ such that every vertex in $u \in L$ has at least one incident edge in E' and every $v \in R$ has at most one incident edge in E'.

The formula is satisfiable if and only if the graph has a matching of size $|L|$.

The formula is encoded with variables $p_{u,v}$ for $u \in L$ and $v \in R$ where the intended meaning is that $p_{u,v}$ is *True* when pigeon u flies into hole v. There are different variants of this formula, depending on the values of *functional* and *onto* argument.

- PHP(G): each $u \in L$ can fly to multiple $v \in R$
- FPHP(G): each $u \in L$ can fly to exactly one $v \in R$
- onto-PHP: each $v \in R$ must get a pigeon
- matching: E' must be a perfect matching

Parameter G can be either of type `cnfgen.graphs.BipartiteGraph` or of type a `networkx.graph`. In the latter case it must be a correct representation of a bipartite graph according to [NetworkX].

Parameters

G [`cnfgen.graphs.BipartiteGraph` or `networkx.graph`] the bipartite graph describing the possible pairings

functional: bool enforce at most one edge per left vertex

onto: bool enforce that any right vertex has one incident edge

Returns

cnfgen.formula.cnf.CNF A CNF formulas encoding the graph pigeonhole principle.

Raises

TypeError G is neither a `cnfgen.graphs.BipartiteGraph` nor a `networkx.graph`

ValueError G is not a proper bipartite graph

References

[Networkx] https://networkx.org/documentation/networkx-2.5/reference/algorithms/generated/networkx.algorithms.bipartite.basic.is_bipartite.html

PigeonholePrinciple(*pigeons*, *holes*, *functional=False*, *onto=False*)

Pigeonhole Principle CNF formula

The pigeonhole principle CNF formula claims that that it is possibile to place m pigeons into n holes without collisions. This is clearly impossible whenever $m > n$.

The formula is encoded with variables $p_{i,j}$ for $i \in [m]$ and $j \in [n]$ where the intended meaning is that $p_{i,j}$ is *True* when pigeon i flies into hole j. 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.

Parameters

pigeon: int number of pigeons (must be >=0).

hole: int number of holes (must be >=0).

functional: bool, optional enforce at most one hole per pigeon (default: False).

onto: bool, optional enforce that any hole must have a pigeon (default: False).

Returns

`cnfgen.formula.cnf.CNF` A CNF formulas encoding the pigeonhole principle.

Raises

TypeError If either *pigeons* or *holes* is not an integer number.

ValueError If either *pigeons* or *holes* is less than zero.

Examples

```
>>> print(PigeonholePrinciple(4,3).to_dimacs())
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
```

(continues on next page)

(continued from previous page)

```
-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
<BLANKLINE>
```

RelativizedPigeonholePrinciple (*pigeons*, *resting_places*, *holes*)
Relativized Pigeonhole Principle CNF formula

This formula is a variant of the pigeonhole principle. We consider m pigeons, r resting places, and n holes. The formula claims that pigeons can fly into holes with no conflicts, with the additional caveat that before landing in a hole, each pigeon stops in some resting place. No two pigeons can rest in the same place.

The formula is encoded with variables $p_{i,j}$ for $i \in [m]$ and $k \in [t]$, and variables $q_{k,j}$ for $k \in [t]$ and $j \in [n]$. The intended meaning is that $p_{i,k}$ is *True* when pigeon i rests into a resting place k, and $q_{k,j}$ is *True* when the pigeon resting at k flies into hole j. The formula is only satisfiable when $m \leq t \leq n$.

A more complete description of the formula can be found in [ALN16]

Parameters

pigeons: int number of pigeons (must be >=0).

resting_places: int number of resting places (must be >=0).

holes: int number of holes (must be >=0).

Returns

cnfgen.formula.cnf.CNF A CNF formulas encoding the pigeonhole principle.

Raises

TypeError If either *pigeons*, *resting_places*, or *holes* is not an integer number.

ValueError If either *pigeons*, *resting_places*, or *holes* is less than zero.

References

[ALN16]

5.1.7 cnfgen.families.pitfall module

Implementation of the Pitfall formula by Marc Vinyals, according to the paper [MV20].

PitfallFormula (*v*, *d*, *ny*, *nz*, *k*)
Pitfall Formula

The Pitfall formula was designed to be specifically easy for Resolution and hard for common CDCL heuristics. The formula is unsatisfiable and consists of three parts: an easy formula, a hard formula, and a pitfall misleading the solver into working with the hard part.

The hard part are several copies of an unsatisfiable Tseitin formula on a random regular graph. The pitfall part is made up of a few gadgets over (primarily) two sets of variables: pitfall variables, which point the solver towards the hard part after being assigned, and safety variables, which prevent the gadget from breaking even if a few other variables are assigned.

For more details, see the corresponding paper [1].

Parameters

v [positive integer] number of vertices of the Tseitin graph

d [positive integer] degree of the Tseitin graph

ny [positive integer] number of pitfall variables

nz [positive integer] number of safety variables

k [positive, even integer] number of copies of the hard and pitfall parts; controls how easy the easy part is

Returns

A CNF object

Raises

ValueError The is no d-regular graph when *v* < *d* or *d***v* are odd.

References

[1]

5.1.8 cnfgen.families.ramsey module

CNF Formulas for Ramsey-like statements

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

TypeError Parameters are not integers

References

[1]

RamseyNumber (s, k, N)

Ramsey number r(s,k) > N

This formula, given s, k, and N, claims that there is some graph with N vertices which has neither independent sets of size s nor cliques of size k.

It turns out that there is a number $r(s, k)$ so that every graph with at least $r(s, k)$ vertices must contain either one or the other. Hence the generated formula is satisfiable if and only if

$$r(s, k) > N$$

Parameters

s [int] independent set size

k [int] clique size

N [int] number of vertices

Returns

A CNF object

Raises

ValueError Parameters are not positive

TypeError Parameters are not integers

VanDerWaerden (*N*, *k1*, *k2*, **ks*)

Formula claims that van der Waerden number vdw(k1,k2,k3,k4,...) > N

Consider a coloring the of integers from 1 to N, with d colors. The coloring has an arithmetic progression of color c of length k if there are i and d so that all numbers

$$i, i+d, i+2d, \ldots, i+(k-1)d$$

have color c. In fact, given any number of lengths k_1, k_2,..., k_C, there is some value of N large enough so that no matter how the integers $1, \ldots, N$ are colored with C colors, such coloring must have one arithmetic progression of color c and length k_c.

The smallest N such that it is impossible to avoid the arithmetic progression regardless of the coloring is called van der Waerden number and is denotes as

$$VDW(k_1, k_2, \ldots, k_C)$$

The formula, given N and :math‘k_1‘, :math‘k_2‘ , ldots, :math‘k_C‘, is the CNF encoding of the claim

$$VDW(k_1, k_2, \ldots, k_C) > N$$

which is expressed, more concretely, as a CNF which forbids, for each color c between 1 and C, all arithmetic progressions of length k_C

Parameters

N [int] size of the interval

k1: int length of the arithmetic progressions of color 1

k2: int length of the arithmetic progressions of color 2

***ks** [optional] lengths of the arithmetic progressions of color >2

Returns

A CNF object

Raises

ValueError Parameters are not positive

TypeError Parameters are not integers

5.1.9 cnfgen.families.randomformulas module

Random CNF Formulas

RandomKCNF (*k*, *n*, *m*, *seed=None*, *planted_assignments=None*)

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(lists), optional] a set of total/partial assigments such that all clauses in the formula will be satisfied by all of them. Each partial assignment is a sequence of literals. Undefined behaviour if some assignment contains opposite literals.

Returns

a CNF object

Raises

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

all_clauses (*k*, *n*, *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*, *n*, *m*, *planted_assignments*)

Sample m random k-clauses on a set of n variables

First it tries sparse sampling: - samples with repetition which is fast - filters bad samples

If after enough samples we haven't got enough clauses we use dense sampling, namely we generare all possible clauses and pick at random m of them. This approach always succeeds, but is quite slower and wasteful for just few samples.

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

5.1.10 cnfgen.families.subgraph module

Implementation of formulas that check for subgraphs

BinaryCliqueFormula (*G*, *k*, *symbreak=True*)

Test whether a graph has a k-clique (binary encoding)

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 [cnfgen.Graph] a simple graph

k [a non negative integer] clique size

symbreak: bool force mapping to be non decreasing

Returns

a CNF object

CliqueFormula (*G*, *k*, *symbreak=True*)

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 [cnfgen.Graph] a simple graph

k [a non negative integer] clique size

symbreak: bool force mapping to be non decreasing

Returns

a CNF object

RamseyWitnessFormula (*G*, *k*, *s*, *symbreak=True*)

True if graph contains either k-clique or and s independent set

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 [cnfgen.Graph] a simple graph

k [a non negative integer] clique size

s [a non negative integer] independet set size

symbreak: bool force mapping to be non decreasing

Returns

a CNF object

SubgraphFormula (*G*, *H*, *induced=False*, *symbreak=False*)

Test whether a graph has a k-clique.

Given two graphs H and G, the CNF formula claims that H is an (induced) subgraph of G.

Parameters

G [cnfgen.Graph] a simple graph

H [cnfgen.Graph] the candidate subgraph

induced: bool test for induced containment

symbreak: bool force mapping to be non decreasing (this makes sense only if T is symmetric)

Returns

a CNF object

non_edges (*G*)

5.1.11 cnfgen.families.subsetcardinality module

Implementation of subset cardinality formulas

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 [cnfgen.graphs.BipartiteGraph] 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]

5.1.12 cnfgen.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]

5.1.13 cnfgen.families.tseitin module

Implementation of Tseitin formulas

TseitinFormula(*G*, *charges=None*)
Build a Tseitin formula based on the input graph.

By default, an odd charge is put on the first vertex, unless another pattern of charges are specified. The pattern is specified via a sequence of boolean values in the *charges* variable (True means odd). If the sequence is shorter than the sequence of vertices, it is padded with Falses. If it is longer, excessive values will be ignored. Any non-boolean value in *charges* is interpreted as boolean via *bool* cast.

Parameters

G [cnfgen.Graph or networkx.Graph]

charges: a sequence of boolean

5.1.14 cnfgen.families.cpls module

Implementation of Thapen's size-width tradeoff formula

CPLSFormula(*a*, *b*, *c*)
Thapen's size-width tradeoff formula

The formula is a propositional version of the coloured polynomial local search principle (CPLS). A description can be found in [1]. The difference with the formula in the paper is that here, unary indices start from 1 instead of 0. Binary strings stil counts from 0, therefore the mappings $f[i](x) = x'$ is actually represented in binary with the binary representation of $x' - 1$.

Parameters

a: integer number of levels

b: integer nodes per level (must be a power of 2)

c: integer number of colours (must be a power of 2)

References

[1]

intlog2(*x*)
Compute the ceiling of the log2(x)

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 5 pigeons and 4 holes. We can get the same formula in DIMACS format with the following command line.

```
$ cnfgen php 5 4
c description: Pigeonhole principle formula for 5 pigeons and 4 holes
c generator: CNFgen (0.8.6-5-g56a1e50)
c copyright: (C) 2012-2020 Massimo Lauria <massimo.lauria@uniroma1.it>
c url: https://massimolauria.net/cnfgen
c command line: cnfgen php 5 4
c
```

(continues on next page)

(continued from previous page)

```
p cnf 20 45
1 2 3 4 0
5 6 7 8 0
9 10 11 12 0
13 14 15 16 0
17 18 19 20 0
-1 -5 0
-1 -9 0
-1 -13 0
-1 -17 0
-5 -9 0
-5 -13 0
-5 -17 0
-9 -13 0
-9 -17 0
-13 -17 0
-2 -6 0
-2 -10 0
-2 -14 0
-2 -18 0
-6 -10 0
-6 -14 0
-6 -18 0
-10 -14 0
-10 -18 0
-14 -18 0
-3 -7 0
-3 -11 0
-3 -15 0
-3 -19 0
-7 -11 0
-7 -15 0
-7 -19 0
-11 -15 0
-11 -19 0
-15 -19 0
-4 -8 0
-4 -12 0
-4 -16 0
-4 -20 0
-8 -12 0
-8 -16 0
-8 -20 0
-12 -16 0
-12 -20 0
-16 -20 0
```

CHAPTER 6

Graph based formulas

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

```
>>> import cnfgen
>>> from pprint import pprint
>>> G = cnfgen.Graph.star_graph(5)
>>> list(G.edges())
[(1, 6), (2, 6), (3, 6), (4, 6), (5, 6)]
>>> F = cnfgen.TseitinFormula(G,charges=[0,1,0,0,1,0])
>>> pprint(F.solve())
(True, [-1, 2, -3, -4, 5])
```

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

Internally, vertices of these graphs are identified as integer starting from 1. Edges are pairs of integers and in general the data structure is such that edge lists and neighborhoods are given in a sorted fashion. - `cnfgen.Graph` to represent undirected graphs `simple`. - `cnfgen.DirectedGraph`: to represent directed graphs

> `digraph` and `dag` (directed acyclic graphs). A DAG is a *DirectedGraph* where all edges go from vertices with loweer id to vertices with higher id. Therefore the ids of the vertices must represent a topological order of the DAG. In particular a directed graph maybe acyclic but yet not considered a dag in `CNFgen`. The method `cnfgen.DirectedGraph.is_dag()` checks that the directed graph is indeed a DAG according to this standard.

- `cnfgen.BipartiteGraph` represents graph of `bipartite` type. The vertices are divided in two parts (left and right) and the vertices in each part are enumerated from 1. For example in a graph with 10 vertices on the left side and 4 vertices on the right side, the edge `(6,3)` connects vertex `6` on the left with vertex `4` on the right. Similarly edge `(2,2)` connects vertex 2 on the left to vertex 2 on the right.

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

6.1 Directed Acyclic Graphs

In CNFgen a DAG is an object of type `cnfgen.DirectedGraph` which furthermore passes the test `cnfgen.DirectedGraph.is_dag()`. We stress that the vertices numeric id must induce a topological order for the graph to be a dag.

```
>>> from cnfgen import DirectedGraph
>>> G = DirectedGraph(3)
>>> G.add_edges_from([(1,2),(2,3),(3,1)])
>>> G.is_dag()
False
>>> H = DirectedGraph(4)
>>> H.add_edges_from([(1,2),(2,3),(3,4)])
>>> H.is_dag()
True
>>> Z = DirectedGraph(4)
>>> Z.add_edges_from([(1,2),(3,2)])
>>> Z.is_dag()
False
```

6.2 Bipartite Graphs

We represent bipartite graphs using `cnfgen.BipartiteGraph`.

```
>>> B = cnfgen.graphs.BipartiteGraph(2,3)
>>> B.left_order()
2
>>> B.right_order()
3
>>> B.order()
5
>>> B.add_edges_from([(1,2),(2,1),(2,3)])
>>> B.number_of_edges()
3
>>> F = cnfgen.GraphPigeonholePrinciple(B)
>>> sorted(F.all_variable_labels())
['p_{1,2}', 'p_{2,1}', 'p_{2,3}']
```

6.3 Graph I/O

Furthermore CNFgen allows graphs I/O on files, in few formats. The function `cnfgen.supported_graph_formats()` lists the file formats available for each graph type.

```
>>> from cnfgen import supported_graph_formats
>>> from pprint import pprint
>>> pprint(supported_graph_formats())
{'bipartite': ['kthlist', 'gml', 'dot', 'matrix'],
 'dag': ['kthlist', 'gml', 'dot', 'dimacs'],
 'digraph': ['kthlist', 'gml', 'dot', 'dimacs'],
 'simple': ['kthlist', 'gml', 'dot', 'dimacs']}
```

The `dot` and `gml` formats are read using NetworkX library, which is a powerful library for graph manipulation. The support for the other formats is natively implemented.

The `dot` format is is from Graphviz and it is available only if the optional `pydot` python package is installed in the system. The Graph Modelling Language (GML) `gml` is a modern industrial standard in graph representation.

The DIMACS (dimacs) format[2] is used sometimes for programming competitions or in the theoretical computer science community. For more informations about kthlist and matrix formats you can refer to the User Documentation.

To facilitate graph I/O CNFgen has to functions cnfgen.graphs.readGraph() and cnfgen.graphs.writeGraph().

Both readGraph and writeGraph operate either on filenames, encoded as *str*, or on file-like objects such as

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

```
>>> import sys
>>> from io import BytesIO
>>> import networkx as nx
>>> from cnfgen import readGraph, writeGraph, BipartiteGraph
```

```
>>> G = BipartiteGraph(3,3,name='a bipartite graph')
>>> G.add_edges_from([[1,1],[1,2],[2,3]])
>>> G.number_of_edges()
3
>>> writeGraph(G,sys.stdout,graph_type='bipartite',file_format='gml')
graph [
  name "a bipartite graph"
  node [
    id 0
    label "1"
    bipartite 0
  ]
  node [
    id 1
    label "2"
    bipartite 0
  ]
  node [
    id 2
    label "3"
    bipartite 0
  ]
  node [
    id 3
    label "4"
    bipartite 1
  ]
  node [
    id 4
    label "5"
    bipartite 1
  ]
  node [
    id 5
    label "6"
    bipartite 1
  ]
  edge [
    source 0
    target 3
  ]
  edge [
```

(continues on next page)

[2] Beware. Here we are talking about the DIMACS format for graphs, not the DIMACS file format for CNF formulas.

(continued from previous page)

```
    source 0
    target 4
  ]
  edge [
    source 1
    target 5
  ]
]
<BLANKLINE>
>>> from io import StringIO
>>> textbuffer = StringIO("graph X { 1 -- 2 -- 3 }")
>>> G = readGraph(textbuffer, graph_type='simple', file_format='dot')
>>> E = G.edges()
>>> (1, 2) in E
True
>>> (2, 3) in E
True
>>> (1, 3) in E
False
```

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 `cnfgen.graphs.readGraph()` would check that.

```
>>> buffer = StringIO('digraph A { 1 -- 2 -- 3 -- 1}')
>>> readGraph(buffer,graph_type='dag',file_format='dot')
Traceback (most recent call last):
...
ValueError: [Input error] Graph must be explicitly 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(tmpdir+"example_dag1.dot","w") as f:
...     print("digraph A {1->2->3}",file=f)
>>> G = readGraph(tmpdir+"example_dag1.dot",graph_type='dag')
>>> list(G.edges())
[(1, 2), (2, 3)]
```

is equivalent to

```
>>> with open(tmpdir+"example_dag2.gml","w") as f:
...     print("digraph A {1->2->3}",file=f)
>>> G = readGraph(tmpdir+"example_dag2.gml",graph_type='dag',file_format='dot')
>>> list(G.edges())
[(1, 2), (2, 3)]
```

Instead, if we omit the format and the file extension is misleading we would get and error.

```
>>> with open(tmpdir+"example_dag3.gml","w") as f:
...     print("digraph A {1->2->3}",file=f)
>>> G = readGraph(tmpdir+"example_dag3.gml",graph_type='dag')
Traceback (most recent call last):
...
ValueError: [Parse error in GML input] ...
```

This is an example of GML file.

```
>>> gml_text ="""graph [
...             node [
```

(continues on next page)

(continued from previous page)

```
...                 id 1
...                 label "a"
...               ]
...               node [
...                 id 2
...                 label "b"
...               ]
...               edge [
...                 source 1
...                 target 2
...               ]
...             ]"""
>>> with open(tmpdir+"example_ascii.gml","w",encoding='ascii') as f:
...     print(gml_text,file=f)
>>> G = readGraph(tmpdir+"example_ascii.gml",graph_type='simple')
>>> (1,2) in G.edges()
True
```

Recall that GML files are supposed to be ASCII encoded.

```
>>> gml_text2="""graph [
...               node [
...                 id 0
...                 label "à"
...               ]
...               node [
...                 id 1
...                 label "è"
...               ]
...               edge [
...                 source 0
...                 target 1
...               ]
...             ]"""
```

```
>>> with open(tmpdir+"example_utf8.gml","w",encoding='utf-8') as f:
...     print(gml_text2,file=f)
>>> G = readGraph(tmpdir+"example_utf8.gml",graph_type='dag')
Traceback (most recent call last):
...
ValueError: [Non-ascii chars in GML file] ...
```

6.4 Graph generators

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

6.5 References

CHAPTER 7

Post-process a CNF formula

After you produce a `cnfgen.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

We implement the following transformation methods. 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 `cnfgen.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.

Some method implemented as still missing from the list

Name	Description	Default rank	See documentation
`none`	leaves the formula alone	ignored	
`eq`	all variables equal	3	`cnfgen.AllEqualSubstitution`
`ite`	if x then y else z	ignored	`cnfgen.IfThenElseSubstitution`
`lift`	lifting	3	`cnfgen.FormulaLifting`
`maj`	Loose majority	3	`cnfgen.MajoritySubstitution`
`neq`	not all vars equal	3	`cnfgen.NotAllEqualSubstitution`
`one`	Exactly one	3	`cnfgen.ExactlyOneSubstitution`
`or`	OR substitution	2	`cnfgen.OrSubstitution`
`xor`	XOR substitution	2	`cnfgen.XorSubstitution`

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

```
>>> from cnfgen import CNF, XorSubstitution
>>> F = CNF([ [1,2,-3], [-2,4] ])
>>> G = XorSubstitution(F,2)
```

Here is the original formula.

```
>>> print( F.to_dimacs() )
p cnf 4 2
1 2 -3 0
-2 4 0
<BLANKLINE>
```

Here it is after the transformation.

```
>>> print( G.to_dimacs() )
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
<BLANKLINE>
```

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 `cnfgen` library and the `cnfgen` command line utility.

10.1 The `cnfgen` library

The `cnfgen` 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 `cnfgen.CNF` object. Let's see a simple example of its usage.

```
>>> from pprint import pprint
>>> import cnfgen
>>> F = cnfgen.CNF()
>>> F.add_clause([1,-2])
>>> F.add_clause([-1])
>>> outcome,assignment = F.solve() # outputs a pair
>>> outcome                        # is the formula SAT?
True
>>> pprint(assignment)             # a solution
[-1, -2]
>>> F.add_clause([2])
>>> F.solve()             # no solution
(False, None)
>>> print(F.to_dimacs())
p cnf 2 3
1 -2 0
-1 0
2 0
<BLANKLINE>
>>> print(F.to_latex())
\begin{align}
```

(continues on next page)

[1] http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps

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

[3] http://en.wikipedia.org/wiki/Proof_complexity

(continued from previous page)

```
&       \left(            {x_1} \lor {\overline{x}_2} \right) \\
& \land \left( {\overline{x}_1} \right) \\
& \land \left(            {x_2} \right)
\end{align}
```

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

```
>>> from cnfgen import PigeonholePrinciple
>>> F = PigeonholePrinciple(5,4)
>>> print(F.to_dimacs())
p cnf 20 45
1 2 3 4 0
5 6 7 8 0
...
-16 -20 0
<BLANKLINE>
>>> F.is_satisfiable()
False
```

10.2 The cnfgen command line tool

The command line tool is installed along `cnfgen` 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 description: Pigeonhole principle formula for 5 pigeons and 4 holes
c generator: CNFgen (0.8.5.post1-7-g4e234b7)
c copyright: (C) 2012-2020 Massimo Lauria <massimo.lauria@uniroma1.it>
c url: https://massimolauria.net/cnfgen
c command line: cnfgen php 5 4
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)

[H85] Haken, A. (1985). The intractability of resolution. Theoretical Computer Science, 39, 297–308.

[ALN16] Atserias, A., Lauria, M., & Nordstr"om, Jakob (2016). Narrow Proofs May Be Maximally Long. ACM Transactions on Computational Logic, 17(3), 19–1–19–30. http://dx.doi.org/10.1145/2898435

[MV20] Marc Vinyals. Hard examples for common variable decision heuristics. In Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI) 2020, pp. 1652–1659. https://doi.org/10.1609/aaai.v34i02.5527

[1] Marc Vinyals. Hard examples for common variable decision heuristics. In Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI) 2020, pp. 1652–1659. https://doi.org/10.1609/aaai.v34i02.5527

[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)

[1] N. Thapen (2016) Trade-offs between length and width in resolution. Theory of Computing, 12(1), 1–14.

Python Module Index

f

Index

A

B

C

E

G

I

N

O

P

R

S

T

V