Package: satire (via r-universe)

June 3, 2026

Type Package

Title Define and Solve Boolean Satisfiability ('SAT') Problems

Version 1.0.0.9000

Maintainer Mike Cheng <mikefc@coolbutuseless.com>

URL https://github.com/coolbutuseless/satire

BugReports https://github.com/coolbutuseless/satire/issues

Description Boolean satisifiability ('SAT') tests whether there are sets of variables which satisfy a Boolean formula. This package provides tools for defining 'SAT' problems and includes a 'DPLL'-based 'SAT' solver based upon Davis & Putnam (1960) <doi:10.1145/321033.321034>. Built-in capabilities for Tseytin transformation (Tseytin (1968) <doi:10.1007/978-3-642-81955-1_28>) allows for the input of complex Boolean formula. Cardinality constraints are implemented using commander variables (Klieber & Kwon (2007) <https://www.cs.cmu.edu/~wklieber/papers/2007_efficient-cnf-encoding-for-selecting-1.pdf>) and sequential counters (Sinz (2005) <doi:10.1007/11564751_73>). Tools for reading and writing 'DIMACS' files are also included.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.2

Imports stringr

Suggests knitr, rmarkdown, testthat (>= 3.0.0), ggplot2, purrr, glue

Config/testthat/edition 3

VignetteBuilder knitr

Config/pak/sysreqs libicu-dev

Repository https://coolbutuseless.r-universe.dev

Date/Publication 2025-04-07 00:17:36 UTC

RemoteUrl https://github.com/coolbutuseless/satire

RemoteRef HEAD

RemoteSha 2bb32cf1a319d6f95a7a67330ebccaaff741de8d

Contents

as_cnf . . . 2
is_cnf . . . 3
literals_to_dimacs . . . 4
print.sat_dimacs . . . 4
print.sat_prob . . . 5
read_dimacs . . . 6
sat_add_exprs . . . 6
sat_add_literals . . . 7
sat_block_solution . . . 8
sat_card_exactly_k . . . 9
sat_card_exactly_one . . . 10
sat_copy . . . 11
sat_literals_to_lgl . . . 12
sat_new . . . 12
sat_solve_dpll . . . 13
sat_solve_naive . . . 14
tseytin_transform . . . 15

Index **16**

`as_cnf` *Convert an arbitrary Boolean expression to conjunctive normal form (CNF)*

Description

Expression may include nesting with brackets, `&`, `|`, `!`, `->` (implication) and `==` (equivalence). All other elements in the expression must be names. Names must only contain a-z, A-Z, 0-9 and underscore.

Usage

```
as_cnf(expr, dummy_idx = 1L)
```

Arguments

`expr`	Boolean expression
`dummy_idx`	starting index for dummy numbering. If the supplied expression is not in CNF, then Tseytin transform may be applied, and this often generates extra dummy variables. In order to avoid variable name clashes it is necessary for the caller to provide a `dummy_idx` (unique within a given SAT problem) to number these dummy variables.

Value

string containing an expression in CNF

Examples

```
as_cnf("a | b")     # Already CNF. returns the argument unchanged
as_cnf("a -> b")    # Rewrite via substitution
as_cnf("!!a & b")   # Rewrite via substitution
as_cnf("!(a | b)")  # Uses Tseytin transform
```

---

is_cnf *Determine if the expression (in a string) is in conjunctive normal form*

Description

Determine if the expression (in a string) is in conjunctive normal form

Usage

```
is_cnf(s)
```

Arguments

s	single string containing an expression

Value

Logical value. TRUE if expression is a CNF. Return FALSE if expression is a valid boolean formula but is not in CNF. Otherwise an error is raised.

Examples

```
# Expect TRUE for these CNF expressions
is_cnf("a | !b")
is_cnf("(a | !b) & (c | d | e)")

# Expect FALSE for these expressions
is_cnf("!!a")
is_cnf("!(a & b)")
is_cnf("!(a | b)")
is_cnf("a & (b | (d & e))")
```

literals_to_dimacs *Convert literals to a DIMACS-formatted string*

Description

Convert literals to a DIMACS-formatted string

Usage

```
literals_to_dimacs(literals)
```

Arguments

literals	integer vector of positive and negative literal values. Clauses should be separated using zero (0). If the vector is not terminated by a zero (0), then a zero will be added.

Value

Character string holding the DIMACS representation

Examples

```
sat <- sat_new()
sat_add_exprs(sat, "!a | b")
sat_add_exprs(sat, "!b | c")
sat_add_exprs(sat, "!c | a")
literals_to_dimacs(sat$literals)
```

print.sat_dimacs *Print a DIMACs representation returned from* literals_to_dimacs()

Description

Print a DIMACs representation returned from literals_to_dimacs()

Usage

```
## S3 method for class 'sat_dimacs'
print(x, ...)
```

Arguments

x	sat_dimacs object
...	ignored

Value

None

Examples

```
literals <- c(1, 2, 3, 0, -2, 4, 0, 5, 6, 7, 0)
dimacs <- literals_to_dimacs(literals)
dimacs
```

`print.sat_prob` *Print 'sat' object*

Description

Print 'sat' object

Usage

```
## S3 method for class 'sat_prob'
print(x, ...)
```

Arguments

`x`	'sat' object created by `sat_new()`
`...`	ignored

Value

Invisible copy of original `x`

Examples

```
sat <- sat_new()
print(sat)
```

read_dimacs *Read a DIMACS file as a* sat *object.*

Description

This function reads SAT data from standard DIMACS files. Any comment lines (lines starting with 'c') are ignored.

Usage

```
read_dimacs(filename)
```

Arguments

filename	filename with CNF in DIMACS format

Value

sat object

Examples

```
filename <- system.file("sudoku.cnf", package = 'satire', mustWork = TRUE)
sat <- read_dimacs(filename = filename)
sat
sat$literals[1:100]
```

sat_add_exprs *Add Boolean expressions to a SAT problem*

Description

Note: Adding clauses to a SAT problem can only be via adding integer literals (via sat_add_literals()) or adding named expressions (via sat_add_exprs()) . The two methods cannot be mixed within a single SAT problem.

Usage

```
sat_add_exprs(sat, exprs, verbosity = 0L)
```

Arguments

sat	sat object as created by sat_new()
exprs	a character vector of Boolean expressions - liberal use brackets is encouraged to signify intent. If any expression is not in CNF, an attempt will be made to convert it to CNF using Tseytin transformation. Allowed syntax: ! Negation | OR & AND -> Implication. a -> b will be rewritten as !a | b == Equivalence. a == b will be rewritten as (a -> b) & (b -> a) !! Double negation. !!a will be simplified to a **alpha-numeric names** names must only contain a-z, A-Z, 0-9 and underscore
verbosity	Verbosity level. Default: 0

Value

None

See Also

Other ways of adding clauses: sat_add_literals()

Examples

```
# Solve a small SAT problem using strings
sat <- sat_new()
sat_add_exprs(sat, "!a | b")
sat_add_exprs(sat, "!b | c")
sat_add_exprs(sat, "!c | a")
sat_solve_naive(sat)

# Multiple string clauses may be added a single call.
sat <- sat_new()
sat_add_exprs(sat, c("!a | b", "!b | c", "!c | a"))
sat_solve_naive(sat)
```

---

sat_add_literals *Add integer literals to a SAT problem*

Description

Note: Adding clauses to a SAT problem can only be via adding integer literals (via sat_add_literals()) or adding named expressions (via sat_add_exprs()) . The two methods cannot be mixed within a single SAT problem.

Usage

```
sat_add_literals(sat, literals)
```

Arguments

sat	sat object as created by sat_new()
literals	integer vector of positive and negative literal values. Clauses should be separated using zero (0). If the vector is not terminated by a zero (0), then a zero will be added.

Value

None

See Also

Other ways of adding clauses: sat_add_exprs()

Examples

```
# Define a small SAT problem
sat <- sat_new()
sat_add_literals(sat, c(-1, 2))
sat_add_literals(sat, c(-2, 3))
sat_add_literals(sat, c(-3, 1))
sat_solve_naive(sat)


# Equivalent problem
sat <- sat_new()
sat_add_literals(sat, c(-1, 2, 0, -2, 3, 0, -3, 1, 0))
sat_solve_naive(sat)
```

---

sat_block_solution *Block a given result - use this for manually finding multiple solutions to a given problem.*

Description

Block a given result - use this for manually finding multiple solutions to a given problem.

Usage

```
sat_block_solution(sat, literals, verbosity = 0L)
```

Arguments

sat	sat object as created by sat_new()
literals	A solution. Vector of integer literals.
verbosity	verbosity

Value

None

Examples

```
# Setup a problem and solve
sat <- sat_new()
sat_add_exprs(sat, "!a | b")
sat_add_exprs(sat, "!b | c")
sat_add_exprs(sat, "!c | a")
sat_solve_naive(sat)

# Block the solution where all values are TRUE
soln1 <- "a & b & c"

# Block this solution from occurring
sat_block_solution(sat, soln1)

# Now only a single solution returned
sat_solve_naive(sat)
```

---

sat_card_exactly_k *Add to SAT problem the cardinality constraint that "exactly k", "at most k" or "at least k" of the given variables are true*

Description

This constraint is an implementation of Sinz's LTseq formulation.

Usage

```
sat_card_exactly_k(sat, nms, k)

sat_card_atleast_k(sat, nms, k)

sat_card_atmost_k(sat, nms, k)
```

Arguments

sat	sat object as created by sat_new()
nms	character vector of variable names
k	k limit

Value

a character vector of expression strings representing the cardinality constraint in conjunctive normal form

See Also

Other SAT cardinality constraints: `sat_card_exactly_one()`

Examples

```
sat <- sat_new()
sat_card_exactly_k(sat, c('a', 'b', 'c', 'd'), k = 2)
sat$exprs

sat <- sat_new()
sat_card_atleast_k(sat, c('a', 'b', 'c', 'd'), k = 2)
sat$exprs

sat <- sat_new()
sat_card_atmost_k(sat, c('a', 'b', 'c', 'd'), k = 2)
sat$exprs
```

---

sat_card_exactly_one *Add to SAT problem the cardinality constraint for "exactly one", "at most one" or "at least one" of the given variables is true*

Description

Add to SAT problem the cardinality constraint for "exactly one", "at most one" or "at least one" of the given variables is true

Usage

```
sat_card_exactly_one(sat, nms, method = "pairwise")

sat_card_atleast_one(sat, nms)

sat_card_atmost_one(sat, nms, method = "pairwise")
```

Arguments

`sat`	`sat` object as created by `sat_new()`
`nms`	character vector of variable names
`method`	nethod to use to encode this constraint. Default: 'pairwise'
	`pairwise` Generates zero new variables, but `O(n^2)` new clauses
	`commander` Divide-and-conquer approach

Value

a character vector of expression strings representing the cardinality constraint in conjunctive normal form

See Also

Other SAT cardinality constraints: `sat_card_exactly_k()`

Examples

```
sat <- sat_new()

sat_card_exactly_one(sat, c('a', 'b', 'c', 'd'))
sat$exprs

sat <- sat_new()
sat_card_atleast_one(sat, c('a', 'b', 'c', 'd'))
sat$exprs

sat <- sat_new()
sat_card_atmost_one(sat, c('a', 'b', 'c', 'd'))
sat$exprs
```

---

sat_copy *Copy a SAT problem*

Description

Copy a SAT problem

Usage

```
sat_copy(sat)
```

Arguments

`sat`	Original SAT problem

Value

Copy of SAT problem

Examples

```
sat <- sat_new()
sat_add_literals(sat, c(1, 2))
sat2 <- sat_copy(sat)
sat_add_literals(sat, c(3, 4))
sat
sat2
```

sat_literals_to_lgl *Convert a sequence of integer literals to named logical vector*

Description

This function is used when a set of literals (Usually corresponding to a solution) needs to be turned into a logical statement with named variables.

Usage

```
sat_literals_to_lgl(sat, literals, remove = "^dummy")
```

Arguments

sat	sat object as created by sat_new()
literals	literals
remove	regular expression for variables to remove when blocking solutions and assembling values to return. Default: "^dummy" will block all variables starting with the word "dummy" (as this is how the 'satire' package automatically creates dummy variables.) If NULL no variables will be removed.

Value

Named logical vector indicating the names and boolean values of the given literals

Examples

```
sat <- sat_new()
sat_add_exprs(sat, c("(a | !b) & (!c | !d)"))
sat$names
sat$named_literals
sat_literals_to_lgl(sat, c(1, 3, -4))
```

sat_new *Initialise a new SAT problem*

Description

Initialise a new SAT problem

Usage

```
sat_new()
```

Value

sat object

Examples

```
sat <- sat_new()
sat
sat_add_literals(sat, c(1, 2))
sat_add_literals(sat, c(2, -3))
sat
sat$names
sat$named_literals
sat$exprs
sat$dimacs
sat_solve_naive(sat)
```

---

sat_solve_dpll *SAT solver using DPLL technique*

Description

This pure R recursive solver uses DPLL methods (unit propagation and pure literal elimination). This solver works recursively and can be used to solve problems with 10s to 100s of variables. However, as all the code is in R, this is not as fast as using a compiled solver.

Usage

```
sat_solve_dpll(sat, max_solutions = 1, remove = "^dummy", verbosity = 0L)
```

Arguments

sat	SAT problem as created by sat_new()
max_solutions	maximum number of solutions to return. Default: 1
remove	regular expression for variables to remove when blocking solutions and assembling values to return. Default: "^dummy" will block all variables starting with the word "dummy" (as this is how the 'satire' package automatically creates dummy variables.) If NULL no variables will be removed.
verbosity	verbosity level. Default: 0

Details

For larger problems >100 variables, it may better to export the problem as a DIMACS file and solve externally.

Value

data.frame of logical values. Columns correspond to the variable names within the problem. Each row defines a solution.

See Also

Other SAT solvers: `sat_solve_naive()`

Examples

```
sat <- sat_new()
sat_card_atmost_k(sat, letters[1:4], 3)
sat$exprs
sat$names
sat_solve_naive(sat)
sat_solve_dpll(sat, max_solutions = 20)
```

---

sat_solve_naive *Naive, brute-force SAT solver with up-front memory allocation*

Description

This solver is only suitable for small problems as memory use is exponential in the number of variables and all memory is pre-allocated. E.g. a problem with 21 variables will pre-allocate about a gigabyte of memory. Each extra variable doubles the memory required - e.g. a problem with 32 variables would require allocation of 520 GB of memory.

Usage

```
sat_solve_naive(sat, remove = "^dummy", mem_limit = 1024, verbosity = 0L)
```

Arguments

`sat`	SAT problem as created by `sat_new()`
`remove`	regular expression for variables to remove when blocking solutions and assembling values to return. Default: "^dummy" will block all variables starting with the word "dummy" (as this is how the 'satire' package automatically creates dummy variables.) If NULL no variables will be removed.
`mem_limit`	only run problems if the estimated memory allocation is less than this number of MB. Default: 1024. This is a guard to prevent catastrophic consequences of trying to allocate too much memory if given a large problem.
`verbosity`	verbosity level. Default: 0

Value

data.frame of logical values. Columns correspond to the variable names within the problem. Each row defines a solution. If problem is unsatisfiable, return NULL.

See Also

Other SAT solvers: `sat_solve_dpll()`

Examples

```
sat <- sat_new()
sat_card_atmost_k(sat, letters[1:4], 3)
sat$exprs
sat$names
sat_solve_naive(sat)
```

`tseytin_transform` *Convert a boolean formula to CNF using Tseytin's transformation*

Description

Convert a boolean formula to CNF using Tseytin's transformation

Usage

```
tseytin_transform(expr, dummy_idx = 1L)
```

Arguments

`expr`	expression (as string or language object)
`dummy_idx`	value to use for numbering the dummy variables.

Value

character vector of expressions in conjunctive normal form

Examples

```
tseytin_transform("a | (b & c)")
tseytin_transform("!(a | b)")
```

Index

∗ **SAT cardinality constraints**
    sat_card_exactly_k, 9
    sat_card_exactly_one, 10
∗ **SAT solvers**
    sat_solve_dpll, 13
    sat_solve_naive, 14
∗ **export functions**
    literals_to_dimacs, 4
∗ **ways of adding clauses**
    sat_add_exprs, 6
    sat_add_literals, 7

as_cnf, 2

is_cnf, 3

literals_to_dimacs, 4, *4*

print.sat_dimacs, 4
print.sat_prob, 5

read_dimacs, 6

sat_add_exprs, 6, *8*
sat_add_literals, 7, *7*
sat_block_solution, 8
sat_card_atleast_k
        *(*sat_card_exactly_k*)*, 9
sat_card_atleast_one
        *(*sat_card_exactly_one*)*, 10
sat_card_atmost_k *(*sat_card_exactly_k*)*,
        9
sat_card_atmost_one
        *(*sat_card_exactly_one*)*, 10
sat_card_exactly_k, 9, *11*
sat_card_exactly_one, 10, *10*
sat_copy, 11
sat_literals_to_lgl, 12
sat_new, *5*, *7–10*, 12, *12–14*
sat_solve_dpll, 13, *14*
sat_solve_naive, 14, *14*

tseytin_transform, 15