Logic Solver

Source code of released version | Source code of development version

---

Introduction

Logic Solver is a boolean satisfiability solver written in JavaScript. Given a problem expressed as logical constraints on boolean (true/false) variables, it either provides a possible solution, or tells you definitively that there is no possible assignment of the variables that satisfies the constraints.

Many kinds of logic problems can be expressed in terms of constraints on boolean variables, including Sudoku puzzles, scheduling problems, and the package dependency problem faced by package managers that automatically resolve version conflicts.

Logic Solver can handle complex problems with thousands of variables, and has some powerful features such as incremental solving and solving under temporary assumptions. It also supports small-integer sums and inequalities, and can minimize or maximize an integer expression.

Logic Solver contains a copy of MiniSat, an industrial-strength SAT solver, compiled from C++ to JavaScript using Emscripten. See About MiniSat for more information.

Logic Solver can solve a hard Sudoku in under a second in a web browser, with very clean-looking code compared to many constraint solvers. **Try this demo**

On NPM

```
var Logic = require('logic-solver');
```

https://www.npmjs.com/package/logic-solver

Table of Contents

Example: Dinner Guests

We are trying to decide what combination of Alice, Bob, and Charlie to invite over to dinner, subject to the following constraints:

- Don't invite both Alice and Bob
- Invite either Bob or Charlie

Setting up these constraints in code:

```
var solver = new Logic.Solver();
```

```
solver.require(Logic.atMostOne("Alice", "Bob"));
solver.require(Logic.or("Bob", "Charlie"));
```

Solving now will give us one possible solution, chosen arbitrarily:

```
var sol1 = solver.solve();
sol1.getTrueVars() // => ["Bob"]
```

Let's see what happens if we invite Alice. By using `solveAssuming`, we can look for a solution that makes an additional logical expression true over the ones we have required so far:

```
var sol2 = solver.solveAssuming("Alice");
sol2.getTrueVars() // => ["Alice", "Charlie"]
```

Aha! It seems that inviting Alice means we can't invite Bob, but then we must invite Charlie! If our reasoning is correct, it is impossible to invite Alice and not invite Charlie. We can confirm this:

```
solver.solveAssuming(Logic.and("Alice", "-Charlie")) // => null
```

(Note that `"-Charlie"` is shorthand for `Logic.not("Charlie")`.)

Let's write some code to list all possible solutions:

```
var solutions = [];
var curSol;
while ((curSol = solver.solve())) {
  solutions.push(curSol.getTrueVars());
  solver.forbid(curSol.getFormula()); // forbid the current solution
}

solutions
// => [["Alice", "Charlie"], ["Charlie"], ["Bob", "Charlie"], ["Bob"]]
```

As you can see, there are four possible solutions to the original problem.

After running the above code, all possible solutions are now forbidden, so the solver is in an unsatisfiable state. Calls to `solver.require` and `solver.forbid` are permanent, so we cannot return to a satisfiable state, and any call to `solve` or `solveAssuming` henceforth will return no solution:

```
solver.solve() // => null
```

It's informative to look at the clauses generated by Logic Solver during this example. In this notation, `v` is the boolean "OR" operator:

```
-Alice v -Bob   (at most one of Alice, Bob)
Bob v Charlie   (at least one of Bob, Charlie)

Alice v -$assump1   (solve assuming Alice)
```

```
$and1 v -$assump2  (solve assuming Alice and not Charlie)
Alice v -$and1
-Charlie v -$and1

-Alice v Bob v -Charlie  (forbid ["Alice", "Charlie"])
Alice v Bob v -Charlie   (forbid ["Charlie"])
Alice v -Bob v -Charlie  (etc.)
Alice v -Bob v Charlie
```

These clauses are sent to MiniSat using variable numbers in place of names, making the entire problem quite compact:

```
[[-3,-4], [4,5],
 [3,-6],
 [8,-7], [3,-8], [-5,-8],
 [-3,4,-5], [3,4,-5], [3,-4,-5], [3,-4,5]]
```

Example: Magic Squares

A 3x3 "magic square" is an arrangement of the digits 1 through 9 into a square such that the digits in each row, column, and diagonal add up to the same number. Here is an example from Wikipedia:

```
2 7 6
9 5 1
4 3 8
```

Each row, column, and three-digit diagonal adds up to 15, as you can verify. (There are many 3x3 magic squares, but the magic sum is always 15, because all the digits together add up to 45!)

Let's use Logic Solver to find magic squares. We could be fancy about it and write code that would generalize to NxN magic squares, but let's keep it simple and name the digit locations as follows:

```
A B C
D E F
G H I
```

Because each location holds an integer, we must use integer variables instead of boolean variables. An integer in Logic Solver is represented as a group of bits, where each bit is a boolean variable, or an entire boolean formula. Let's create a 4-bit group of variables for each digit location:

```
var A = Logic.variableBits('A', 4);
var B = Logic.variableBits('B', 4);
var C = Logic.variableBits('C', 4);
var D = Logic.variableBits('D', 4);
var E = Logic.variableBits('E', 4);
var F = Logic.variableBits('F', 4);
var G = Logic.variableBits('G', 4);
var H = Logic.variableBits('H', 4);
var I = Logic.variableBits('I', 4);

var locations = [A, B, C, D, E, F, G, H, I];
```

```
A.bits // => ["A$0", "A$1", "A$2", "A$3"]
```

Let's also assign the number 15, in bit form, to a variable for convenience.

```
var fifteen = Logic.constantBits(15);
fifteen.bits // => ["$T", "$T", "$T", "$T"]
```

The binary representation of 15 is "1111", so its bit form consists of four copies of `Logic.TRUE` or `"$T"`. We didn't have to know that, though, because `Logic.constantBits` generated it for us.

Now, we create a Solver and express our sum constraints:

```
var solver = new Logic.Solver();

_.each([[A,B,C], [D,E,F], [G,H,I], [A,D,G], [B,E,H], [C,F,I],
        [A,E,I], [G,E,C]],
       function (terms) {
         solver.require(Logic.equalBits(Logic.sum(terms), fifteen));
       });
```

Let's see what solution we get!

```
var sol1 = solver.solve();
sol1.evaluate(A) // => 3
sol1.evaluate(B) // => 10 (uh oh)
_.map(locations, function (loc) { return sol1.evaluate(loc); })
// => [3, 10, 2,
//     4,  5, 6,
//     8,  0, 7]
```

Oops, it looks like we forgot to specify that each "digit" is between 1 and 9! There is no harm done, because we have only underspecified the problem. We can continue to use the same `solver` instance.

Now we add inequalities to make each location A through I hold a number between 1 and 9 inclusive, and solve again:

```
_.each(locations, function (loc) {
  solver.require(Logic.greaterThanOrEqual(loc, Logic.constantBits(1)));
  solver.require(Logic.lessThanOrEqual(loc, Logic.constantBits(9)));
});

var sol2 = solver.solve();
_.map(locations, function (loc) { return sol2.evaluate(loc); })
// => [8, 1, 6,
//     3, 5, 7,
//     4, 9, 2]
```

Now we have a proper magic square!

However, it just so happens that we also forgot to specify that the numbers be distinct. To demonstrate that this is an important missing constraint, we can use `solveAssuming` to ask for a solution where A and B are equal:

```
var sol3 = solver.solveAssuming(Logic.equalBits(A, B));
_.map(locations, function (loc) { return sol3.evaluate(loc); })
// => [4, 4, 7,
//     8, 5, 2,
//     3, 6, 6]
```

Or where A, B, and C are equal:

```
var sol4 = solver.solveAssuming(Logic.and(Logic.equalBits(A, B),
                                          Logic.equalBits(B, C)));
_.map(locations, function (loc) { return sol4.evaluate(loc); })
// => [5, 5, 5,
//     5, 5, 5,
//     5, 5, 5]
```

A good way to enforce that all locations hold different digits is to generate a requirement about each pair of different locations:

```
_.each(locations, function (loc1, i) {
  _.each(locations, function (loc2, j) {
    if (i !== j) {
      solver.forbid(Logic.equalBits(loc1, loc2));
    }
  });
});
```

Solving now gives us a proper magic square again:

```
var sol5 = solver.solve();
_.map(locations, function (loc) { return sol5.evaluate(loc); })
// => [6, 7, 2,
//     1, 5, 9,
//     8, 3, 4]
```

If we wished to continue interrogating the solver, we could try asking for a magic square with a 1 in the upper-left corner, or proceed to enumerate a list of magic squares.

Finally, let's demonstrate that our "integers" are really just groups of boolean variables:

```
sol5.getTrueVars()
// => ["A$1", "A$2", "B$0", "B$1", "B$2", "C$1", "D$0", "E$0", "E$2",
//     "F$0", "F$3", "G$3", "H$0", "H$1", "I$2"]

_.map(A.bits, function (v) { return sol5.evaluate(v); })
// => [false, true, true, false]
```

You may be wondering whether it's bad that we generated 72 constraints as part of finding a 3x3 magic square. While there are certainly much faster ways to calculate magic squares, it is perfectly reasonable when setting up a logic problem to generate a complete set of pairwise constraints over N variables. In fact, having more constraints often improves performance in real-world problems, so it is worth generating extra constraints even when they are technically redundant. More constraints means more deductions can be made at each step, meaning fewer possibilities need to be tried that ultimately won't work out. In this case, it's important that when the solver assigns a digit to a particular location, it immediately be able to deduce that the same number does not appear at any other location.

Variables

Variable names are Strings which can contain spaces and punctuation:

```
Logic.implies('it is raining', 'take an umbrella');

Logic.exactlyOne("1,1", "1,2", "1,3")
```

Restrictions: A variable name must not be empty, consist of only the characters `0` through `9`, or start with `-`. Variable names that start with `$` are reserved for internal use.

You do not need to declare or create your variables before using them in formulas passed to `require` and `forbid`.

When you pass a variable name to a Solver for the first time, a variable number is allocated, and that name and number become synonymous for that Solver instance. You don't need to know about variable numbers to use Logic Solver, but you can always use a variable number in place of a variable name in terms and formulas, in case that is useful. (It is useful internally, and would probably be useful if you were to wrap Logic Solver in another library.) Examples of Solver methods that may allocate new variables are `require`, `forbid`, `solveAssuming`, and `getVarNum`.

If you want to add a free variable to a Solver but not require anything about it, you can use `getVarNum` to cause the variable to be allocated. It will then appear in solutions.

Methods

Logic.Solver#getVarNum(variableName, [noCreate])

Returns the variable number for a variable name, allocating a number if this is the first time this Solver has seen `variableName`.

PARAMETERS

- `variableName` - String - A valid variable name.
- `noCreate` - Boolean - Optional. If true, this method will return 0 instead of allocating a new variable number if `variableName` is new.

RETURNS

Integer - A positive integer variable number, or 0 if `noCreate` is true and there is no variable number allocated for `variableName`.

Logic.Solver#getVarName(variableNum)

Returns the variable name for a given variable number. An error is thrown if `variableNum` is not an allocated variable number.

PARAMETERS

- `variableNum` - Integer - An allocated variable number.

RETURNS

String - A variable name.

Terms

A Term is a variable name or number, optionally negated. To negate a string Term, prefix it with `"-"`. Examples of valid Terms are `"foo"`, `"-foo"`, `5`, and `-5`. In other solvers and papers, you may see Terms referred to as "literals."

The following are equivalent:

```
solver.require("-A");
solver.require(Logic.not("A"));
solver.forbid("A");
```

In fact, `Logic.not("A")` returns `"-A"`. It is valid to have more than one `-` in a Term (`"---A"`), and the meaning will be what you'd expect, but `Logic.not` will never return you such a Term, so in practice this case does not come up. `Logic.not("-A")` returns `"A"`.

String Terms are called NameTerms, and numeric Terms are called NumTerms. You will not normally need to use numeric Terms, but if you do, note that it doesn't make sense to share them across Solver instances, because each Solver has its own variable numbers. See the Variables section for more information.

Constants

Logic.FALSE, Logic.TRUE

These Terms represent the constant boolean values false and true. You may seem them appear as the internal variables `$F` and `$T` or `1` and `2`, which are automatically pinned to false and true.

Methods

Logic.isTerm(value)

Returns whether `value` is a valid Term. A valid Term is either a String consisting of a valid variable name preceded by zero or more `-` characters, or a non-zero integer.

PARAMETERS

- `value` - Any

RETURNS

Boolean

Logic.isNameTerm(value)

Returns whether `value` is a valid NameTerm (a Term that is a String).

PARAMETERS

- `value` - Any

RETURNS

Boolean

Logic.isNumTerm(value)

Returns whether `value` is a valid NumTerm (a Term that is a Number).

PARAMETERS

- `value` - Any

RETURNS

Boolean

Logic.Solver#toNameTerm(term)

Converts a Term to a NameTerm if it isn't already. If `term` is a NumTerm, the variable number is translated into a variable name. An error is thrown if the variable number is not an allocated variable number of this Solver.

PARAMETERS

- `term` - Term - The Term to convert, which may be a NameTerm or NumTerm.

RETURNS

NameTerm

Logic.Solver#toNumTerm(term, [noCreate])

Converts a Term to a NumTerm if it isn't already. If `term` is a NameTerm, the variable name is translated into a variable number. A new variable number is allocated if the variable name has not been seen before by this Solver, unless you pass true for `noCreate`.

PARAMETERS

- `term` - Term - The Term to convert, which may be a NameTerm or NumTerm.
- `noCreate` - Boolean - Optional. If true, this method will not allocate a new variable number if it encounters a new variable name, but will return 0 instead.

RETURNS

NumTerm, or 0 (if `noCreate` is true and a new variable name is encountered)

Formulas

A Formula is an object representing a boolean expression. Conceptually, a Formula is built out of Terms and operations that combine Terms.

Here are some examples of Formulas:

```
// A and B
Logic.and("A", "B")

// If exactly one of (A, B, C) is true, then A does not equal D.
Logic.implies(Logic.exactlyOne("A", "B", "C"),
              Logic.not(Logic.equiv("A", "D")))

// More of (x1, x2, x3) are true than (y1, y2, y3)
var xs = ["x1", "x2", "x3"];
var ys = ["y1", "y2", "y3"];
Logic.greaterThan(Logic.sum(xs), Logic.sum(ys))
```

Formulas are immutable. To be on the safe side, do not mutate any arrays you use to create a Formula.

Formulas are Solver-independent. They can be created without a Solver, and although Solvers keep track of Formula objects and recognize them (to avoid compiling the same Formula twice), a Formula object never becomes tied to one Solver object and can always be reused, as long as it doesn't contain any explicit variable numbers (NumTerms).

A Term is not a Formula, but you can always pass a Term anywhere a Formula is required.

Functions such as `Logic.and` and `Logic.greaterThan` are called Formula constructor functions. One thing to note about them is that they do not always return Formulas, but may return Terms as well. `Logic.and("A")`, for example, returns `"A"`. Some constructor functions take any number of arguments, which may be nested in arrays, so that the following are equivalent:

```
Logic.and("A", "B", "C")
Logic.and(["A", "B", "C"])
Logic.and("A", [["B", "C"]], [])
```

To use a Formula, you must tell a Solver to `require` or `forbid` it. Otherwise, the Formula does not take effect.

```
var solver = new Logic.Solver();
solver.require("A");

Logic.exactlyOne("A", "B"); // no effect, just creates a Formula

solver.require(Logic.exactlyOne("A", "B")); // this works

var myFormula = Logic.exactlyOne("A", "B");
solver.require(myFormula); // this also works
```

You should save and reuse Formula objects whenever possible, because the Solver will recognize the Formula object and not recompile it. Internally, each Formula is replaced by a variable in the Solver, such as `$and1` for a `Logic.and`, and clauses are generated that relate the variable to the operands of the Formula. When you pass the same Formula object again, it is replaced by the same variable, and the Formula only needs to be compiled once.

Formulas that operate on integers are documented in the Bits section.

Methods

Logic.isFormula(value)

Returns true if `value` is a Formula object. (A Term is not a Formula.)

PARAMETERS

- `value` - Any

RETURNS

Boolean

Logic.not(operand)

Represents a boolean expression that is true when its operand is false, and vice versa.

When called on an operand that is a NameTerm, NumTerm, or Formula, returns a value of the same kind.

PARAMETERS

- `operand` - Formula or Term

RETURNS

Formula or Term (same kind as `operand`)

EXAMPLES

```
Logic.not("A") // => "-A"
Logic.not("-A") // => "A"
Logic.not(Logic.and("A", "B")) // => a Formula object
```

Logic.or(operands...)

Represents a boolean expression that is true when at least one of its operands is true.

PARAMETERS

- `operands...` - Zero or more Formulas, Terms, or Arrays

RETURNS

Formula or Term

Logic.and(operands...)

Represents a boolean expression that is true when all of its operands are true.

PARAMETERS

- `operands...` - Zero or more Formulas, Terms, or Arrays

RETURNS

Formula or Term

Logic.xor(operands...)

Represents a boolean expression that is true when an odd number of its operands are true.

PARAMETERS

- `operands...` - Zero or more Formulas, Terms, or Arrays

RETURNS

Formula or Term

Logic.implies(operand1, operand2)

Represents a boolean expression that is true unless `operand1` is true and `operand2` is false. In other words, if this Formula is required to be true, and `operand1` is true, then `operand2` must be true.

PARAMETERS

- `operand1` - Formula or Term
- `operand2` - Formula or Term

RETURNS

Formula or Term

Logic.equiv(operand1, operand2)

Represents a boolean expression that is true when `operand1` and `operand2` are either both true or both false.

PARAMETERS

- `operand1` - Formula or Term
- `operand2` - Formula or Term

RETURNS

Formula or Term

PARAMETERS

- `operands...` - Zero or more Formulas, Terms, or Arrays

RETURNS

Formula or Term

Logic.exactlyOne(operands...)

Represents a boolean expression that is true when exactly one of its operands is true.

PARAMETERS

- `operands...` - Zero or more Formulas, Terms, or Arrays

RETURNS

Formula or Term

Logic.atMostOne(operands...)

Represents a boolean expression that is true when zero or one of its operands are true.

PARAMETERS

- `operands...` - Zero or more Formulas, Terms, or Arrays

RETURNS

Formula or Term

Logic.Solver

You create a Logic.Solver with `new Logic.Solver()`.

A Solver maintains a list of Formulas that must be true (or false), which you can think of as a list of constraints. Each Solver instance embeds a self-contained MiniSat instance, which learns and remembers facts that are derived from the constraints. At any time, you can ask the Solver for a solution that satisfies the current constraints, and it will either provide one (chosen arbitrarily) or report that none exists. You can then continue to add more constraints and solve again.

See Example: Dinner Guests for a good introduction to Solver.

Constraints are only ever added, never removed. If the current constraints are not satisfiable, then `solve()` will return null, and adding additional constraints cannot make the problem solvable again. However, using `solveAssuming`, you can look for a solution with a particular Formula temporarily in force. If `solveAssuming` returns null, there is no harm done, and you can continue to solve under other assumptions or add more constraints.

Sometimes `solve()` will take a long time! That is to be expected. The best thing to do is to try expressing the problem in a different way, with fewer variables, more sharing of common subexpressions, or more constraints between variables so that the solver can make important deductions in fewer steps. Also try wrapping your code in `Logic.disablingAssertions(function () { ... })` in case runtime type checks are slowing down Formula compilation.

If you need an extra speed boost in Node, you could help me create a binary npm package containing a native-compiled MiniSat.

Constructor

new Logic.Solver()

Methods

Logic.Solver#require(args...)

Requires that the Formulas and Terms listed in `args` be true in order for a solution to be valid.

PARAMETERS

- `args...` - Zero or more Formulas, Terms, or Arrays

Logic.Solver#forbid(args...)

Requires that the Formulas and Terms listed in `args` be *false* in order for a solution to be valid.

PARAMETERS

- `args...` - Zero or more Formulas, Terms, or Arrays

Logic.Solver#solve()

Finds a solution that satisfies all the constraints specified with `require` and `forbid`, or determines that no such solution is possible. A solution is an assignment of all the variables to boolean values.

To find more than one solution, you can forbid the first solution (using `solver.forbid(solution.getFormula())`, and solve again.

Solving is fully incremental, and each call to `solve()` has the benefit of everything learned by previous calls to `solve()`. Re-solving with one or two new constraints is typically very fast, because no work is repeated.

There is no guarantee of which solution is found if there are more than one. However, some statements can be made about what to expect:

- MiniSat starts by trying a solution where all variables are false, so underconstrained variables will tend to be set to false.
- Calling `solve()` repeatedly, with no intervening method calls, will in practice return the same solution each time. On the other hand, if you call `solve`, then `solveAssuming`, then `solve` again, the call to `solveAssuming` will affect the solution returned by the second `solve`.
- Logic Solver and MiniSat are deterministic, so the same series of calls on a new Solver will generally produce the same results. However, the results may not be stable across different versions of Logic Solver.

RETURNS

Logic.Solution, or null if no solution is possible

Logic.Solver#solveAssuming(assumption)

Like `solve()`, but looks for a solution that additionally satisfies `assumption`. This is especially useful for testing whether a new constraint would make the problem unsolvable before requiring it, or for "querying" the solver about different types of solutions.

Note that any solution returned by `solveAssuming` is also a valid solution for `solve` to return. If you call `solve`, then `solveAssuming`, then `solve` again, the second `solve` will typically return the same solution as `solveAssuming`, because the internal state of the solver has been changed (even though no new permanent constraints have been introduced).

PARAMETERS

- `assumption` - Formula or Term

RETURNS

Logic.Solution or null

Logic.disablingAssertions(func)

Calls `func()`, disabling runtime type checks and assertions for the duration. This speeds up the processing of complex Formulas, especially when integers or large numbers of variables are involved, at the price of not validating the arguments to most function calls. It doesn't affect the time spent in MiniSat.

PARAMETERS

- `func` - Function

RETURNS

Any - The return value of `func()`.

Logic.Solution

A Solution represents an assignment or mapping of the Solver variables to true/false values. Solution objects are returned by `Logic.Solver#solve` and `Logic.Solver#solveAssuming`.

(Variables internal to the Solver, which start with `$` and which you'd probably only encounter while poking around in internals, are not considered part of the assignment.)

Methods

Logic.Solution#getMap()

Returns a complete mapping of variables to their assigned values.

RETURNS

Object - Dictionary whose keys are variable names and whose values are booleans

Logic.Solution#getTrueVars()

Returns a list of all the variables that are assigned to true by this Solution.

RETURNS

Array of String - Names of the variables that are assigned to true

Logic.Solution#evaluate(expression)

Evaluates a Formula or Term under this Solution's assignment of the variables, returning a boolean value. For example:

```
solution.evaluate('A')
solution.evaluate('-A')
solution.evaluate(Logic.or('A', 'B'))
solution.evaluate(myFormula) // Formula given to the Solver earlier
```

If `expression` is a Bits, the result of evaluation is an integer:

```
var x = Logic.variableBits('x', 3); // 3-digit binary variable
var y = Logic.variableBits('y', 3);
```

```
var xySum = Logic.sum(x, y);
var five = Logic.constantBits(5);

var solver = new Logic.Solver;
solver.require(Logic.equalBits(xySum, five));
var solution = solver.solve();
solution.evaluate(x) // 2 (for example)
solution.evaluate(y) // 3 (for example)
solution.evaluate(five) // 5
```

It is an error to try to evaluate an unknown variable or a variable that did not exist at the time the Solution was created, unless you call `ignoreUnknownVariables()` first.

PARAMETERS

- `expression` - Formula, Term, or Bits

RETURNS

Boolean or Integer

Logic.Solution#getFormula()

Creates a Formula (or Term) which can be used to require, or forbid, that variables are assigned to the exact values they have in this Solution.

To find all solutions to a logic problem:

```
var solver = new Logic.Solver;
solver.require(Logic.or('A', 'B'));

var allSolutions = [];
var curSolution = null;
while ((curSolution = solver.solve())) {
  allSolutions.push(curSolution.getTrueVars());
  solver.forbid(curSolution.getFormula());
}

allSolutions // [["A"], ["A", "B"], ["B"]]
```

Adding a constraint and solving again in this way is quite efficient.

The Formula or Term returned may not be used with any other Solver instance besides the one that produced this Solution.

RETURNS

Formula or Term

Logic.Solution#getWeightedSum(formulas, weights)

Equivalent to `evaluate(Logic.weightedSum(formulas, weights))`, but much faster because the addition is done using integer arithmetic, not boolean logic. Rather than constructing a Bits and evaluating it, `getWeightedSum` simply evaluates each of the Formulas to a boolean value and then sums the weights corresponding to the Formulas that evaluate to true.

See `Logic.weightedSum`.

PARAMETERS

- `formulas` - Array of Formula or Term
- `weights` - Array of non-negative integers, or a single non-negative integer

RETURNS

Integer

Logic.Solution#ignoreUnknownVariables()

Causes all evaluation by this Solution instance, from now on, to treat variables that aren't part of this Solution as false instead of throwing an error. This includes unrecognized variable names and variables that were created after this Solution was created.

This method cannot be undone. Good style is to call it once when you first get the Solution object, or not at all.

Optimization

Logic Solver can perform basic integer optimization, using a combination of inequalities and incremental solving. The methods in this section are utilities for minimizing or maximizing the value of a weighted sum, which is a type of problem sometimes called pseudo-boolean optimization.

To understand how these methods work, remember that if you have one solution and want another solution to the same problem, a good technique is to forbid the current solution and then re-solve. In a similar vein, if you have one solution and want another solution that yields a larger or smaller value for an integer expression, you can simply express this new constraint as an inequality and re-solve. The final wrinkle is to use `solveAssuming` to test out each inequality before requiring it, so that when the minimum or maximum value is found, the solver is not put into an unsatisfiable state. The methods in this section implement this technique for you.

This approach to integer optimization works surprisingly well, even when it takes many iterations to achieve the optimum value of a large cost function. However, depending on the structure of your problem, it may be quite a time-consuming operation.

Methods

Logic.Solver#minimizeWeightedSum(solution, formulas, weights)

Finds a Solution that minimizes the value of `Logic.weightedSum(formulas, weights)`, and adds a requirement that this minimum value is obtained (in the sense of calling `Solver#require` on this Solver).

To determine this minimum value, call `solution.getWeightedSum(formulas, weights)` on the returned Solution.

A currently valid Solution must be passed in as a starting point. This starting Solution must have been obtained by calling `solve` or `solveAssuming` on this Solver, and in addition, being "currently valid" means that no calls to `require` or `forbid` have been made since the Solution was produced that conflict with its assignments.

Note that while this method may add constraints to the Solver, the Solver is always in a satisfiable state both before and after this method is called.

PARAMETERS

- `solution` - Logic.Solution - A currently valid Solution for this Solver.
- `formulas` - Array of Formula or Term
- `weights` - Array of non-negative integers, or a single non-negative integer

RETURNS

Logic.Solution - A valid Solution that achieves the minimum value of the weighted sum. It may be `solution` if no improvement on the original value of the weighted sum is possible.

Logic.Solver#maximizeWeightedSum(solution, formulas, weights)

Finds a Solution that maximizes the value of `Logic.weightedSum(formulas, weights)`, and adds a requirement that this maximum value is obtained (in the sense of calling `Solver#require` on this Solver).

To determine this maximum value, call `solution.getWeightedSum(formulas, weights)` on the returned Solution.

A currently valid Solution must be passed in as a starting point. This starting Solution must have been obtained by calling `solve` or `solveAssuming` on this Solver, and in addition, being "currently valid" means that no calls to `require` or `forbid` have been made since the Solution was produced that conflict with its assignments.

Note that while this method may add constraints to the Solver, the Solver is always in a satisfiable state both before and after this method is called.

PARAMETERS

- `solution` - Logic.Solution - A currently valid Solution for this Solver.
- `formulas` - Array of Formula or Term
- `weights` - Array of non-negative integers, or a single non-negative integer

RETURNS

Logic.Solution - A valid Solution that achieves the maximum value of the weighted sum. It may be `solution` if no improvement on the original value of the weighted sum is possible.

Bits (integers)

A Bits object represents an N-digit binary number (non-negative integer) as an array of N Formulas. That is, it has a Formula for the boolean value of each bit. The Formulas are stored in an array called `bits` with the least significant bit first, so `bits[0]` is the ones digit, `bits[1]` is the twos digit, `bits[2]` is the fours digit, and so on. (Note that this is the opposite order from how we usually write numbers! It's much more convenient because the index into the array is always the same as the power of two, with numbers growing to the right as they gain larger-valued digits.)

You usually don't construct a Bits using the constructor, but instead using `Logic.constantBits`, `Logic.variableBits`, or an operation on Formulas such as `Logic.sum`. When you create an integer variable using `Logic.variableBits`, you specify the number of bits N, but in other cases the number of bits is calculated automatically. For example, `Logic.sum()` with no arguments returns a 0-length Bits. `Logic.sum('A', 'B')` returns a 2-length Bits which is the equivalent of `new Logic.Bits([Logic.xor('A', 'B'), Logic.and('A', 'B')])`.

See Example: Magic Squares for a good example of using Bits.

To avoid confusion with NumTerms, there is no automatic promotion of integers to Bits. If you want to use a constant like 5, you must call `Logic.constantBits(5)` to get a Bits object.

There is currently no explicit subtraction nor any negative numbers in Logic Solver.

FIELDS

- `bits` - Array of Formula or Term - Read-only.

Constructor

new Logic.Bits(formulas)

As previously mentioned, it's more common to create a Bits object using `Logic.constantBits`, `Logic.variableBits`, `Logic.sum`, or `Logic.weightedSum` than using this constructor.

PARAMETERS

- `formulas` - Array of Formula or Term - Becomes the value of the `bits` property of this Bits. The array is not copied, so don't mutate the original array. Unlike many Logic Solver methods, this constructor does not take a variable number of arguments, but requires exactly one array.

Methods

Logic.isBits(value)

Returns true if `value` is a Bits object.

PARAMETERS

- `value` - Any

RETURNS

Boolean

Logic.constantBits(wholeNumber)

Creates a constant Bits representing the given number.

For example, `Logic.constantBits(4)` is equivalent to `new Logic.Bits([Logic.FALSE, Logic.FALSE, Logic.TRUE])`.

PARAMETERS

- `wholeNumber` - non-negative integer

RETURNS

Bits

Logic.variableBits(baseName, N)

Creates a Bits representing an N-digit integer variable.

For example, `Logic.variableBits('A', 3)` is equivalent to `new Logic.Bits(['A$0', 'A$1', 'A$2'])`.

PARAMETERS

- `baseName` - String
- `N` - non-negative integer

RETURNS

Bits

Logic.equalBits(bits1, bits2)

Represents a boolean expression that is true when `bits1` and `bits2` are the same integer.

PARAMETERS

- `bits1` - Bits
- `bits2` - Bits

RETURNS

Formula or Term

Logic.lessThan(bits1, bits2)

Represents a boolean expression that is true when `bits1` is less than `bits2`, interpreting each as a non-negative integer.

PARAMETERS

- `bits1` - Bits
- `bits2` - Bits

RETURNS

Formula or Term

Logic.lessThanOrEqual(bits1, bits2)

Represents a boolean expression that is true when `bits1` is less than or equal to `bits2`, interpreting each as a non-negative integer.

PARAMETERS

- `bits1` - Bits
- `bits2` - Bits

RETURNS

Formula or Term

Logic.greaterThan(bits1, bits2)

Represents a boolean expression that is true when `bits1` is greater than `bits2`, interpreting each as a non-negative integer.

PARAMETERS

- `bits1` - Bits
- `bits2` - Bits

RETURNS

Formula or Term

Logic.greaterThanOrEqual(bits1, bits2)

Represents a boolean expression that is true when `bits1` is greater than or equal to `bits2`, interpreting each as a non-negative integer.

PARAMETERS

- `bits1` - Bits
- `bits2` - Bits

RETURNS

Formula or Term

Logic.sum(operands...)

Represents an integer expression that is the sum of the values of all the operands. Bits are interpreted as integers, and booleans are interpreted as 1 or 0.

As with Formula constructor functions that take a variable number of arguments, the operands may be nested in arrays arbitrarily and arbitrarily deeply.

PARAMETERS

- `operands...` - Zero or more Formulas, Terms, Bits, or Arrays

RETURNS

Bits

Logic.weightedSum(formulas, weights)

Represents an integer expression that is a weighted sum of the given Formulas and Terms, after mapping false to 0 and true to 1.

In other words, the sum is: `(formulas[0] * weights[0]) + (formulas[1] * weights[1]) + ...`, where `formulas[0]` is replaced with 0 or 1 based on the boolean value of that Formula.

`weights` may either be an array of non-negative integers, or a single non-negative integer, in which case that weight is used for all formulas. If `weights` is an array, it must have the same length as `formulas`.

PARAMETERS

- `formulas` - Array of Formula or Term
- `weights` - Array of non-negative integers, or a single non-negative integer

RETURNS

Bits

About MiniSat

Solving satisfiability problems ("SAT-solving") is notoriously difficult from an algorithmic perspective, but solvers such as MiniSat implement advanced techniques that have come out of years of research. You can read more about MiniSat on its web page at http://minisat.se/.

MiniSat accepts input in "conjunctive normal form," which is a fairly low-level representation of a logic problem. Logic Solver's main job is to take arbitrary boolean formulas that you specify, such as "exactly one of A, B, and C is true," and compile them into a list of statements that must all be satisfied -- a conjunction of clauses -- each of which is a simple disjunction such as: "A or B or C." "Not A, or not B."

Although MiniSat operates on a low-level representation of the problem and has no explicit knowledge of its overall structure, it is able to use sophisticated techniques to derive new clauses that are implied by the existing clauses. A naive solver would try assigning values to some of the variables until a conflict occurs, and then backtrack, but not really learn anything from the conflict. Even custom solvers written for a particular problem often work this way. Solvers such as MiniSat, on the other hand, employ Conflict-Driven Clause Learning, which means that when they backtrack, they learn new clauses. These new clauses narrow the search space and cause subsequent trials to reach a conflict sooner, until the entire problem is found to be unsatisfiable or a valid assignment is found.

In principle, Logic Solver could be used as a clause generator for other SAT-solver backends besides MiniSat, or for a backend consisting of MiniSat compiled to native machine code instead of JavaScript.