CNF Satisfiability Problem

Andrew Makhorin <mao@gnu.org>

August 2011

1 Introduction

The *Satisfiability Problem (SAT)* is a classic combinatorial problem. Given a Boolean formula of n variables

$$f(x_1, x_2, \ldots, x_n), \tag{1.1}$$

this problem is to find such values of the variables, on which the formula takes on the value *true*.

The *CNF Satisfiability Problem (CNF-SAT)* is a version of the Satisfiability Problem, where the Boolean formula (1.1) is specified in the *Conjunctive Normal Form (CNF)*, that means that it is a conjunction of *clauses*, where a clause is a disjunction of *literals*, and a literal is a variable or its negation. For example:

$$(x_1 \vee x_2) \,\&\, (\neg x_2 \vee x_3 \vee \neg x_4) \,\&\, (\neg x_1 \vee x_4). \tag{1.2}$$

Here x_1, x_2, x_3, x_4 are Boolean variables to be assigned, $\neg$ means negation (logical *not*), $\vee$ means disjunction (logical *or*), and & means conjunction (logical *and*). One may note that the formula (1.2) is *satisfiable*, because on $x_1 = true$, $x_2 = false$, $x_3 = false$, and $x_4 = true$ it takes on the value *true*. If a formula is not satisfiable, it is called *unsatisfiable*, that means that it takes on the value *false* on any values of its variables.

Any CNF-SAT problem can be easily translated to a 0-1 programming problem as follows. A Boolean variable x can be modeled by a binary variable in a natural way: $x = 1$ means that x takes on the value *true*, and $x = 0$ means that x takes on the value *false*. Then, if a literal is a negated variable, i.e. $t = \neg x$, it can be expressed as $t = 1 - x$. Since a formula in CNF is a conjunction of clauses, to provide its satisfiability we should

require all its clauses to take on the value *true*. A particular clause is a disjunction of literals:

$$t \vee t' \vee t'' \ldots, \tag{1.3}$$

so it takes on the value *true* iff at least one of its literals takes on the value *true*, that can be expressed as the following inequality constraint:

$$t + t' + t'' + \ldots \geq 1. \tag{1.4}$$

Note that no objective function is used in this case, because only a feasible solution needs to be found.

For example, the formula (1.2) can be translated to the following constraints:

$$\begin{aligned} x_1 \quad + \quad x_2 \qquad\qquad\qquad\quad &\geq 1 \\ (1 - x_2) + x_3 + (1 - x_4) &\geq 1 \\ (1 - x_1) \qquad + \quad x_4 \quad &\geq 1 \end{aligned}$$

$$x_1, x_2, x_3, x_4 \in \{0, 1\}$$

Carrying out all constant terms to the right-hand side gives corresponding 0-1 programming problem in the standard format:

$$\begin{aligned} x_1 + x_2 \qquad\qquad &\geq \quad 1 \\ -x_2 + x_3 - x_4 &\geq -1 \\ -x_1 \qquad + x_4 &\geq \quad 0 \end{aligned}$$

$$x_1, x_2, x_3, x_4 \in \{0, 1\}$$

In general case translation of a CNF-SAT problem results in the following 0-1 programming problem:

$$\sum_{j \in J_i^+} x_j - \sum_{j \in J_i^-} x_j \geq 1 - |J_i^-|, \quad i = 1, \ldots, m \tag{1.5}$$

$$x_j \in \{0, 1\}, \quad j = 1, \ldots, n \tag{1.6}$$

where n is the number of variables, m is the number of clauses (inequality constraints), $J_i^+ \subseteq \{1, \ldots, n\}$ is a subset of variables, whose literals in i-th clause do not have negation, and $J_i^- \subseteq \{1, \ldots, n\}$ is a subset of variables, whose literals in i-th clause are negations of that variables. It is assumed that $J_i^+ \cap J_i^- = \emptyset$ for all i.

2 GLPK API Routines

2.1 glp_read_cnfsat—read CNF-SAT problem data in DIMACS format

Synopsis

```
int glp_read_cnfsat(glp_prob *P, const char *fname);
```

Description

The routine `glp_read_cnfsat` reads the CNF-SAT problem data from a text file in DIMACS format and automatically translates the data to corresponding 0-1 programming problem instance (1.5)–(1.6).

The parameter `P` specifies the problem object, to which the 0-1 programming problem instance should be stored. Note that before reading data the current content of the problem object is completely erased with the routine `glp_erase_prob`.

The character string `fname` specifies the name of a text file to be read in. (If the file name ends with the suffix '`.gz`', the file is assumed to be compressed, in which case the routine decompresses it "on the fly".)

Returns

If the operation was successful, the routine returns zero. Otherwise, it prints an error message and returns non-zero.

DIMACS CNF-SAT problem format[1]

The DIMACS input file is a plain ASCII text file. It contains lines of several types described below. A line is terminated with an end-of-line character. Fields in each line are separated by at least one blank space.

Comment lines. Comment lines give human-readable information about the file and are ignored by programs. Comment lines can appear anywhere in the file. Each comment line begins with a lower-case character `c`.

```
c This is a comment line
```

[1] This material is based on the paper "Satisfiability Suggested Format", which is publicly available at `http://dimacs.rutgers.edu/`.

Problem line. There is one problem line per data file. The problem line must appear before any clause lines. It has the following format:

```
p cnf VARIABLES CLAUSES
```

The lower-case character `p` signifies that this is a problem line. The three character problem designator `cnf` identifies the file as containing specification information for the CNF-SAT problem. The `VARIABLES` field contains an integer value specifying n, the number of variables in the instance. The `CLAUSES` field contains an integer value specifying m, the number of clauses in the instance.

Clauses. The clauses appear immediately after the problem line. The variables are assumed to be numbered from 1 up to n. It is not necessary that every variable appears in the instance. Each clause is represented by a sequence of numbers separated by either a space, tab, or new-line character. The non-negated version of a variable j is represented by j; the negated version is represented by $-j$. Each clause is terminated by the value 0. Unlike many formats that represent the end of a clause by a new-line character, this format allows clauses to be on multiple lines.

Example. Below here is an example of the data file in DIMACS format corresponding to the CNF-SAT problem (1.2).

```
c sample.cnf
c
c This is an example of the CNF-SAT problem data
c in DIMACS format.
c
p cnf 4 3
1 2 0
-4 3
-2 0
-1 4 0
c
c eof
```

2.2 glp_check_cnfsat—check for CNF-SAT problem instance

Synopsis

```
int glp_check_cnfsat(glp_prob *P);
```

Description

The routine `glp_check_cnfsat` checks if the specified problem object `P` contains a 0-1 programming problem instance in the format (1.5)–(1.6) and therefore encodes a CNF-SAT problem instance.

Returns

If the specified problem object has the format (1.5)–(1.6), the routine returns zero, otherwise non-zero.

2.3 glp_write_cnfsat—write CNF-SAT problem data in DIMACS format

Synopsis

```
int glp_write_cnfsat(glp_prob *P, const char *fname);
```

Description

The routine `glp_write_cnfsat` automatically translates the specified 0-1 programming problem instance (1.5)–(1.6) to a CNF-SAT problem instance and writes the problem data to a text file in DIMACS format.

The parameter `P` is the problem object, which should specify a 0-1 programming problem instance in the format (1.5)–(1.6).

The character string `fname` specifies a name of the output text file to be written. (If the file name ends with suffix '`.gz`', the file is assumed to be compressed, in which case the routine performs automatic compression on writing that file.)

Returns

If the operation was successful, the routine returns zero. Otherwise, it prints an error message and returns non-zero.

2.4 glp_minisat1—solve CNF-SAT problem instance with MiniSat solver

Synopsis

```
int glp_minisat1(glp_prob *P);
```

Description

The routine `glp_minisat1` is a driver to MiniSat, a CNF-SAT solver developed by Niklas Eén and Niklas Sörensson, Chalmers University of Technology, Sweden.[2]

It is assumed that the specified problem object `P` contains a 0-1 programming problem instance in the format (1.5)–(1.6) and therefore encodes a CNF-SAT problem instance.

If the problem instance has been successfully solved to the end, the routine `glp_minisat1` returns 0. In this case the routine `glp_mip_status` can be used to determine the solution status:

`GLP_OPT` means that the solver found an integer feasible solution and therefore the corresponding CNF-SAT instance is satisfiable;

`GLP_NOFEAS` means that no integer feasible solution exists and therefore the corresponding CNF-SAT instance is unsatisfiable.

If an integer feasible solution was found, corresponding values of binary variables can be retrieved with the routine `glp_mip_col_val`.

Returns

0	The MIP problem instance has been successfully solved. (This code does *not* necessarily mean that the solver has found feasible solution. It only means that the solution process was successful.)
`GLP_EDATA`	The specified problem object contains a MIP instance which does *not* have the format (1.5)–(1.6).
`GLP_EFAIL`	The solution process was unsuccessful because of the solver failure.

[2]The MiniSat software module is *not* part of GLPK, but is used with GLPK and included in the distribution.

2.5 glp_intfeas1—solve integer feasibility problem

Synopsis

```
int glp_intfeas1(glp_prob *P, int use_bound, int obj_bound);
```

Description

The routine `glp_intfeas1` is a tentative implementation of an integer feasibility solver based on a CNF-SAT solver (currently it is MiniSat; see Subsection 2.4).

If the parameter `use_bound` is zero, the routine searches for *any* integer feasibile solution to the specified integer programming problem. Note that in this case the objective function is ignored.

If the parameter `use_bound` is non-zero, the routine searches for an integer feasible solution, which provides a value of the objective function not worse than `obj_bound`. In other word, the parameter `obj_bound` specifies an upper (in case of minimization) or lower (in case of maximization) bound to the objective function.

If the specified problem has been successfully solved to the end, the routine `glp_intfeas1` returns 0. In this case the routine `glp_mip_status` can be used to determine the solution status:

- `GLP_FEAS` means that the solver found an integer feasible solution;
- `GLP_NOFEAS` means that the problem has no integer feasible solution (if `use_bound` is zero) or it has no integer feasible solution, which is not worse than `obj_bound` (if `use_bound` is non-zero).

If an integer feasible solution was found, corresponding values of variables (columns) can be retrieved with the routine `glp_mip_col_val`.

Usage Notes

The integer programming problem specified by the parameter `P` should satisfy to the following requirements:

1. All variables (columns) should be either binary (`GLP_BV`) or fixed at integer values (`GLP_FX`).
2. All constraint and objective coefficients should be integer numbers in the range $[-2^{31},\ +2^{31} - 1]$.

Though there are no special requirements to the constraints, currently the routine `glp_intfeas1` is efficient mainly for problems, where most constraints (rows) fall into the following three classes:

1. Covering inequalities
$$\sum_j t_j \geq 1,$$
where $t_j = x_j$ or $t_j = 1 - x_j$, x_j is a binary variable.

2. Packing inequalities
$$\sum_j t_j \leq 1.$$

3. Partitioning equalities (SOS1 constraints)
$$\sum_j t_j = 1.$$

Returns

0	The problem has been successfully solved. (This code does *not* necessarily mean that the solver has found an integer feasible solution. It only means that the solution process was successful.)
`GLP_EDATA`	The specified problem object does not satisfy to the requirements listed in Paragraph 'Usage Notes'.
`GLP_ERANGE`	An integer overflow occured on translating the specified problem to a CNF-SAT problem.
`GLP_EFAIL`	The solution process was unsuccessful because of the solver failure.