MiFuMaX – a Literate MaxSAT Solver

Mikoláš Janota

July 8, 2013

Contents

Chapter 1

Common

1.1 Introduction

Throughout this documentation, terminology commonly used in SAT and MaxSAT community will be used. In particular, we are going to be solving problems containing *Boolean variables* (commonly denoted x,y, etc.). A *literal* is a variable parts negation, e.g. x, $\neg x$. A *clause* is a disjunction of literals (possibly none). A formula in *conjunctive normal form* (CNF) is a conjunction of clauses.

Since conjunction and disjunction are both associative and commutative, it is common to treat a clause as a set of literals and a formula in CNF as set of clauses. Note that the *empty clause*, i.e. the empty disjunction of literals, is semantically equivalent to *false*. Analogously, the empty set of clauses is semantically equivalent *true*.

1.2 Types

The objective was to rely on standard (STL) datatypes and since we will be communicating to `minisat`, we reuse some of its datatypes as well. Note that a CNF (`ClauseVector`) is represented as a vector of vectors. This was done in order to avoid explicit memory allocations and thus simplify the code. In general, however, this might be a bad idea because adding new clauses to the vector might cause lot of copying. We made the assumption that CNFs that we will be handling grow seldom and moreover the SAT calls are far more computationally expensive.

1 ⟨*MiFuMaXTypes.hh* 1⟩≡

```
#ifndef MIFUMAXTYPES_HH_18651
#define MIFUMAXTYPES_HH_18651
#include <vector>
#include <unordered_map>
#include "core/SolverTypes.h"
namespace Mifumax {
```

```
using std::unordered_map;
using std::vector;
using std::pair;
using Minisat::Var;
using Minisat::Lit;
typedef long WeightType;
typedef vector<vector<Lit> > ClauseVector;
typedef pair<WeightType,vector<Lit> > WeightedClause;
typedef vector<WeightedClause> WeightedClauseVector;
typedef unordered_map<Var,size_t> Var2Index;
}
#endif
```

Root chunk (not used in this document).

1.3 Utils

Here we define small utility functions that will be used in the program.

2a ⟨*Utils.hh* 2a⟩≡

```
#ifndef UTILS_HH_46471
#define UTILS_HH_46471
#include <sys/time.h>
#include <sys/resource.h>
#include <vector>
#include "MiFuMaXTypes.hh"
#include "core/Solver.h"
namespace Mifumax {
⟨Auxiliary functions 3a⟩
}
#endif
```

Root chunk (not used in this document).

2b ⟨*Utils.cc* 2b⟩≡

```
#include "Utils.hh"
using namespace Mifumax;
⟨Auxiliary functions implementation 3b⟩
```

Root chunk (not used in this document).

Allocate variables in a minisat solver so that it has at least a variable with the ID `max_id`.

3a ⟨*Auxiliary functions* 3a⟩≡

```
inline void new_variables(Minisat::Solver& solver, Minisat::Var max_id) {
    while (solver.nVars() ≤ max_id) solver.newVar();
}
```

This definition is continued in chunk 4.
This code is used in chunk 2a.

Add given clauses to a minisat solver.

3b ⟨*Auxiliary functions implementation* 3b⟩≡

```
bool Mifumax::add_all(Minisat::Solver& solver, const ClauseVector& cnf) {
  bool okay = true;
  Minisat::vec<Lit> literals; // temporary literal vector
  for (size_t i=0; i<cnf.size(); ++i) {
    const std::vector<Lit>& clause = cnf[i];
    literals.clear();
    literals.growTo(clause.size(), Minisat::lit_Undef);
    for (size_t literal_index = 0; literal_index<clause.size(); ++literal_index)
      literals[(int)literal_index]=clause[literal_index];
    okay &= solver.addClause_(literals);
  }
 return okay;
}
```

This definition is continued in chunk 3c.
This code is used in chunk 2b.

Add weighted clauses to the given minisat solver. The weights are ignored.

3c ⟨*Auxiliary functions implementation* 3b⟩+≡

```
bool Mifumax::add_all(Minisat::Solver& solver, const WeightedClauseVector& wcnf) {
  bool okay = true;
  Minisat::vec<Lit> literals; // temporary literal vector
  for (size_t i=0; i<wcnf.size(); ++i) {
    const vector<Lit>& clause = wcnf[i].second;
    literals.clear();
    literals.growTo(clause.size(), Minisat::lit_Undef);
    for (size_t literal_index = 0; literal_index<clause.size(); ++literal_index)
      literals[(int)literal_index]=clause[literal_index];
    okay &= solver.addClause_(literals);
  }
 return okay;
}
```

4a ⟨*Auxiliary functions* 3a⟩+≡

```
inline double read_cpu_time() {
  struct rusage ru;
  getrusage(RUSAGE_SELF, &ru);
  return (double)ru.ru_utime.tv_sec + (double)ru.ru_utime.tv_usec / 1000000;
}
```

4b ⟨*Auxiliary functions* 3a⟩+≡

```
template<typename K, typename V>
bool contains(const unordered_map<K,V>& m, const K& key)
{return m.find(key)≠m.end();}
```

4c ⟨*Auxiliary functions* 3a⟩+≡

```
bool add_all(Minisat::Solver& solver, const ClauseVector& cnf);
bool add_all(Minisat::Solver& solver, const WeightedClauseVector& wcnf);
```

Chapter 2

MiFuMax

2.1 Introduction

MaxSAT problem instances are given as set of clauses and a solution is an assignment that maximizes the number of satisfied clauses. MaxSAT has several variations. A simple extension is the *partial MaxSAT* where clauses are split into *hard* and *soft*. A solution is an assignment that satisfies all the hard clauses and maximizes the number of satisfied soft clauses.

Even though the definition of MaxSAT may seem at first somewhat cryptic, it becomes far more intuitive once we observe that certain optimization problems can be encoded as MaxSAT. Consider for instance that we want to satisfy the formula $(x \vee y) \wedge (y \vee z)$ and at the same time *minimize* the number of variables set to true. Then we construct the following partial MaxSAT problem: let hard clauses be $\{x \vee y, y \vee z\}$ and soft clauses $\{\bar{x}, \bar{y}, \bar{z}\}$. The solution to this problem is $x = \text{false}, z = \text{false}, y = \text{true}$, unsatisfying one soft clause $(\bar{y})$. Note that this partial MaxSAT instance has only one solution but in general it may have many.

The algorithm we will use is based on an article by Fu&Malik [FM06]. An interesting property of the algorithm is that it starts from an over-constraint, i.e. unsatisfiable, problem and gradually relaxes it until it becomes satisfiable, which is when we have obtained a solution.

2.2 Public Interface

The algorithm is encapsulated in a class `MiFuMaX` relying on the types and utilities declared in `common`. Minisat [ES03] is used as the underlying SAT solver.

5 ⟨*MiFuMaX.hh* 5⟩≡

```
#ifndef MIFUMAX_HH_3876
#define MIFUMAX_HH_3876
#include "MiFuMaXTypes.hh"
#include "Utils.hh"
```

```
#include "core/Solver.h"
using Minisat::Solver;
using Minisat::vec;
using Minisat::lbool;
namespace Mifumax {
 class MiFuMaX {
 public:  ⟨Public members 6c⟩;
 private: ⟨Private members 9a⟩;
 }; }
#endif
```

Root chunk (not used in this document).

6a ⟨*MiFuMaX.cc* 6a⟩≡
⟨*Implementation includes* 6b⟩
⟨*Implementation* 8⟩

Root chunk (not used in this document).

6b ⟨*Implementation includes* 6b⟩≡

```
#include "MiFuMaX.hh"
using namespace Mifumax;
```

This definition is continued in chunks 13a, 20b, 27a, and 33a.
This code is used in chunks 6a and 20a.

The class provides two constructors. The classical *MaxSAT* constructor accepts the CNF for which we want to compute the solution.

6c ⟨*Public members* 6c⟩≡

```
MiFuMaX(Var max_id, const ClauseVector &cnf);
```

This definition is continued in chunks 6, 7, 20, and 21.
This code is used in chunks 5 and 19.

The *partial MaxSAT* constructor accepts two CNFs, which split the problem into hard and soft clauses.

6d ⟨*Public members* 6c⟩+≡

```
MiFuMaX(Var max_id,
        const ClauseVector& hard_cnf,
        const ClauseVector& soft_cnf);
```

To run the solver, one calls `solve`, which upon success returns whether or not the given problem has a solution, i.e. the hard clauses are satisfiable.

6e ⟨*Public members* 6c⟩+≡

```
bool solve();
```

If a solution was found, i.e. `solve()` returned `true`, the number of unsatisfied soft clauses in the solution can be obtained by the function `get_optimum` and the solution assignment by `get_solution`.

7a ⟨*Public members* 6c⟩+≡
```
size_t get_optimum () const {return optimum;}
void get_solution(vec<lbool>& output_model) const;
```

For advanced applications of the solver, some internal data structures are exposed. These methods should not really be used unless a good understanding of the implementation is had.

7b ⟨*Public members* 6c⟩+≡
```
const Var2Index& get_control2index() const { return control2index; }
const ClauseVector& get_hard_clauses() const {return hard_clauses;}
const ClauseVector& get_soft_clauses() const {return soft_clauses;}
const vector<bool>& get_removed_soft_clauses() const {return removed_soft_clauses;}
const vec<lbool>& get_model_internal() const {return model;}
const Var get_max_id() const {return max_id;}
inline const bool is_relaxation_variable(Var v) const {
  return (original_max_id<v) ∧ (v≤max_id);
}
```

2.3 Solving

First, let us look at the `solve` method, which is really the heart of the solver. Suppose that we are given hard clauses ϕ_H and soft clauses ϕ_S. Now if $\phi_H \wedge \phi_S$ are satisfiable by some assignment μ, we are done because μ is a solution where all the soft clauses are satisfied. If, however, this conjunct is unsatisfiable, at least one of the soft clauses must be unsatisfied in *any* solution. Moreover, if there is also some subset $\mathcal{C} \subseteq \phi_S$ for which $\phi_H \wedge \mathcal{C}$ is unsatisfiable, at least one of the clauses from $\mathcal{C}$ must be unsatisfied in any solution. We will refer to such set of clauses $\mathcal{C}$ as a *core*[1]. Subsequently, the algorithm *relaxes* the clauses in $\mathcal{C}$, i.e. it enables to unsatisfy one of the clauses (how that is done, is shown later). This process is repeated until the formula becomes satisfiable, at which point we have a solution.

8 ⟨*Implementation* 8⟩≡

```
bool MiFuMaX::solve() {
  ⟨Initialization solve 9b⟩
  while (true) {
    ⟨Initialize solve loop 9c⟩
    const bool formula_sat = ⟨Check if formula SAT 9d⟩;
    if (formula_sat) {
      return true;
    } else {
      ⟨Relax formula 10a⟩;
    }
  }
}
```

This definition is continued in chunks 12, 13, 15–17, 22, 25–30, and 32.
This code is used in chunks 6a and 20a.

[1] Technically speaking, here we are deviating from the standard terminology because a (unsatisfiable) core is typically defined as an unsatisfiable set of clauses. In our case that would be some unsatisfiable subset of $\phi_H \wedge \phi_S$. However, here we are not interested in the clauses of ϕ_H responsible for unsatisfiability.

Throughout the run of the `solve` method, the variable `optimum` tracks the number of times the formula was relaxed; this number represents a *lower bound* of the number of unsatisfied clauses in any solution. Consequently, when `solve` successfully terminates (it returns `true`), `optimum` is the number of unsatisfied clauses in the found solution.

The variable `model` stores the found solution (if found). Since both `optimum` and `model` should be accessible for the user of our class, they are declared as fields of the class.

9a ⟨*Private members* 9a⟩≡
```
size_t     optimum;
vec<lbool> model;
```

This definition is continued in chunks 11, 14, 17c, 29–31, and 33c.
This code is used in chunks 5 and 19.

The `solve` function reuses variables `core`, `relaxation_variables`, and `model`, which are cleared at the beginning of a each iteration of the loop.

9b ⟨*Initialization solve* 9b⟩≡
```
vector<size_t> core;
vector<Var>    relaxation_variables;
optimum = 0;
```

This code is used in chunk 8.

9c ⟨*Initialize solve loop* 9c⟩≡
```
core.clear();
relaxation_variables.clear();
model.clear();
```

This definition is continued in chunk 23b.
This code is used in chunks 8 and 22.

To check whether the current formula is satisfiable, we issue a SAT call. If the given formula is satisfiable, `model` will contain the satisfying assignment, and otherwise `core` will represent the core as a set of indices of the clauses in the core.

9d ⟨*Check if formula SAT* 9d⟩≡
```
sat_call(core, model)
```

This definition is continued in chunk 23c.
This code is used in chunks 8 and 22.

Before we can actually relax the formula, we need to make one check. And that is that the current core is nonempty. Because if it is empty, it means that the set of hard clauses is itself unsatisfiable and therefore the problem does not have a solution. (Note that this can only happen in the case of partial MaxSAT.)

10a ⟨*Relax formula* 10a⟩≡

```
if (core.empty()) return false;
```

This definition is continued in chunks 10b, 23d, and 24a.
This code is used in chunks 8 and 22.

Recall that now we want to say that one of the clauses from `core` can be unsatisfied. For that we introduce a fresh variable r_C for each clause C in the core and replace the clause C with the clause $r_C \vee C$; the variables r_C are called *relaxation variables*. Now any assignment that sets r_C to true does not need to satisfy C because the new, relaxed clause $r_C \vee C$ is satisfied due to r_C.

Adding relaxation variables as described above, would on its own *not* be correct. The reason is that we know that *at least one* clause from the core must be unsatisfied. But without any further restrictions, any number of clauses from the core could be unsatisfied by setting their corresponding relaxation variable to true. and thus potentially obtaining a satisfying assignment but with a suboptimal number of unsatisfied clauses. In order to cope with that, we add a constraint into hard clauses that *at most one* of the relaxation variables from the core can be true[2]

10b ⟨*Relax formula* 10a⟩+≡

```
for (auto iter=core.begin(); iter ≠ core.end(); ++iter) {
  const size_t clause_index = *iter;
  relaxation_variables.push_back(relax_clause(clause_index));
}
add_at_most_1(relaxation_variables);
```

[2]Note that a original clause C can be relaxed multiple times since cores can intersect.

2.4 Formula Representation

The algorithm is presented above, leaves open how to communicate with the SAT solver and how to obtain cores. To calculate cores, we use the *assumption-based method*. For a clause C we generate a fresh variable s_C, called the control variable. Instead of C, we give to the SAT solver the clause $\neg s_C \vee C$ and pass the literal s_C in the assumptions when calling the SAT solver. If, the SAT solver needed C to conclude that the given formula is unsatisfiable, s_C will appear in the *final conflict clause*. And this is how we calculate our core.

We keep to separate clause vectors for hard and soft clauses. Since we are not interested whether a hard clause participated in unsatisfility, hard clauses are stored unaltered. Soft clauses, however, are stored already containing their corresponding control variable.

While the number of soft clauses does not grow throughout the lifetime of the object, they get modified by relaxation. Hard clauses are not modified during the lifetime of the object but new hard clauses are being added to express cardinality constraints on relaxation variables.

11a ⟨*Private members* 9a⟩+≡
```
ClauseVector hard_clauses;
ClauseVector soft_clauses;
```

The variable `controls` maintains the control variables for soft clauses and it holds that `controls[i]` is the control variables for the clause `soft_clauses[i]` for `0<=i<soft_clauses.size()`. Is Will also be useful to know which control variable corresponds to which clause. This is stored in the map `control2index` for which it holds that for a control variable `v`, `control2index[v]` is the index of the corresponding clause in the vector `soft_clauses`.

11b ⟨*Private members* 9a⟩+≡
```
vector<Var> controls;
Var2Index control2index;
```

Since we will be generating fresh variables, we remember the highest variable ID appearing in the given problem and the highest ID of a including fresh variables.

11c ⟨*Private members* 9a⟩+≡
```
Var original_max_id;
Var max_id;
```

The data structures above are initialized in the constructor. We make a copy of the hard and soft clauses; generate control variables, and adorn soft clauses with control variables.

12a ⟨*Implementation 8*⟩+≡
```
MiFuMaX::MiFuMaX(Var _max_id,
  const ClauseVector &_hard_clauses, const ClauseVector &_soft_clauses)
: hard_clauses(_hard_clauses)
, soft_clauses(_soft_clauses)
, original_max_id(_max_id)
{ initialize(); }
```

12b ⟨*Implementation 8*⟩+≡
```
MiFuMaX::MiFuMaX(Var _max_id, const ClauseVector& _soft_clauses)
: soft_clauses(_soft_clauses)
, original_max_id(_max_id)
{ initialize(); }
```

The initialization procedure common to both constructors populates the required data structures according to their invariants.

12c ⟨*Implementation 8*⟩+≡
```
void MiFuMaX::initialize() {
  max_id = original_max_id;
  controls.resize(soft_clauses.size());
  for (size_t index = 0; index < soft_clauses.size(); ++index) {
    const Var control = ++max_id;
    soft_clauses[index].push_back(~mkLit(control));
    controls[index]=control;
    control2index[control] = index;
  }
}
```

Now when the data structures are in place, it is easy to relax a clause by simply adding a new relaxation variable to it.

12d ⟨*Implementation 8*⟩+≡
```
Var MiFuMaX::relax_clause(size_t clause_index) {
  assert(clause_index < soft_clauses.size());
  const Var relaxation_variable = ++max_id;
  vector<Lit>& clause = soft_clauses[clause_index];
  clause.push_back(mkLit(relaxation_variable));
  return relaxation_variable;
}
```

To encode the at-most-1 constraint, we use an external encoder that produces a CNF representation of the constraint. Here we need to be a little bit careful because the encoder also needs to generate some fresh variables. Here we rely on on the encoder to provide the new maximal ID by its function `get_max_id`.

13a ⟨*Implementation includes* 6b⟩+≡

```
#include "SeqCounter.hh"
#include "BitWise.hh"
```

13b ⟨*Implementation* 8⟩+≡

```
void MiFuMaX::add_at_most_1(const vector<Var>& relaxation_variables) {
  //[[SeqCounter enc(hard_clauses, relaxation_variables, 1, max_id);]]
  //[[BitWise enc(hard_clauses, relaxation_variables, max_id); ]]
  SeqCounter enc(hard_clauses, relaxation_variables, 1, max_id);
  enc.encode();
  assert(max_id≤enc.get_max_id());
  max_id=enc.get_max_id();
}
```

If a user of our class asks for a solution (provided one was found) we just need to supply him with the values of original variables.

13c ⟨*Implementation* 8⟩+≡

```
void MiFuMaX::get_solution(vec<lbool> &output_model) const {
  output_model.growTo(original_max_id+1, l_Undef);
  const Var maxv = std::min(model.size()-1,original_max_id);
  for (Var v=1; v≤maxv; ++v) output_model[v]=model[v];
}
```

2.5 SAT Solver Communication

Now let us look at how a SAT solver is invoked. The actual SAT solver used is minisat. Any other solver could be used as long as it supports solving with *assumptions* and provides the *final conflict clause*.

Recall that during solving, the purpose of the SAT solver is to determine whether the current formula is satisfiable or not. If it is satisfiable, it should provide a model; if it is unsatisfiable, it should provide a core of soft clauses (clauses responsible for unsatisfiability).

Here we make a small optimization not described in the original paper by Fu and Malik. Consider a core with a single clause, i.e. let there be a soft clause C such that for the hard clauses ϕ_H, the conjunct $\phi_H \wedge C$ is unsatisfiable. This means that C does not appear in any solution of the given problem. In other words, C can be simply removed from the set of soft clauses. At the same time, since $\phi_H \wedge C$ is unsatisfiable, $\phi_H \Rightarrow \neg C$, which means that the negations of all literals of C can be added to ϕ_H withoutmodifying the set of models of ϕ_H

In order to effectively remove a soft clause C, we add to the hard clauses the negation of its control variable $\neg s_C$ and mark it as removed. It is very important thatremove clauses are marked because their control variables must not be set to true by assumptions. Removed clauses are stored in a bitvector `removed_soft_clauses` whose size is equal to the size of the `soft_clauses` vector.

14 ⟨*Private members* 9a⟩+≡

```
vector<bool> removed_soft_clauses;
```

15 ⟨*Implementation* 8⟩+≡

```
void MiFuMaX::remove_soft_clause(size_t clause_index, Solver& sat_solver) {
  assert (clause_index < soft_clauses.size());
  const vector<Lit>& c = soft_clauses[clause_index];
  const Var cv=controls[clause_index];
  removed_soft_clauses[clause_index]=true; //marked as removed
  size_t osz=hard_clauses.size();
  hard_clauses.resize(osz+c.size()); // we add c.size hard cls
  hard_clauses[osz++].push_back(~mkLit(cv)); // satisfy control forever
  sat_solver.addClause(~mkLit(cv));

  bool found=false;// sanity check purpose
  for (size_t i=0; i<c.size(); ++i) {
    const Lit l = c[i];
    if (var(l)==cv) {// found control variable
      found = true;
      assert(sign(l));
      continue;
    }
    sat_solver.addClause(~l);
    hard_clauses[osz++].push_back(~l);
  }
  assert(found);// sanity check
}
```

The function `sat_call` builds a new SAT solver constructs assumptions for it and calls it. Depending on the result of the SAT call, a model or a core is built. If it so happens that a core has the size 1, the process is repeated but without constructing a new SAT solver because the only the set hard clauses is modified in the SAT solver was modified by `removed_soft_clause`.

16a ⟨*Implementation* 8⟩+≡

```
bool MiFuMaX::sat_call(vector<size_t> &core, vec<lbool> &model ) {
  Solver solver;
  model.clear();
  core.clear();
  if (removed_soft_clauses.empty()) {
    removed_soft_clauses.resize(soft_clauses.size(),false); }
  populate_solver(solver);
  vec<Lit> assumptions;
 again:
  ⟨Build assumptions 17a⟩
  const bool return_value = solver.solve(assumptions);
  analyze_sat_answer(solver, return_value, core);
  if (!return_value) {
     ++optimum; // update optimum
     std::cout≪"c LB: "≪optimum≪" CS: "≪core.size()≪std::endl;
  }
  if (core.size()==1) {
    remove_soft_clause(core[0],solver);
    core.clear();
    assumptions.clear();
    goto again;
  }
  return return_value;
}
```

In order to populate the solver, we just copy of our data structures into the solver. Recall that `soft_clauses` are not the original soft clauses but clauses that contain control variables and potentially relaxation variables.

16b ⟨*Implementation* 8⟩+≡

```
void MiFuMaX::populate_solver(Solver& solver) {
  new_variables(solver, max_id);
  add_all(solver, soft_clauses);
  add_all(solver, hard_clauses);
}
```

We also need to tell the SAT solver that all control variables must be set to true except for removed soft clauses.

17a ⟨*Build assumptions* 17a⟩≡
```
for (size_t index = 0; index < soft_clauses.size(); ++index) {
  if (!removed_soft_clauses[index])
    assumptions.push(mkLit(controls[index]));
}
```

This definition is continued in chunk 28b.
This code is used in chunks 16a and 28a.

If the SAT solver returned satisfiable, we copy the model from the solver into the global variable `model`. If, the problem is unsatisfiable, we construct a core based on the conflict clause in the solver.

17b ⟨*Implementation* 8⟩+≡
```
void MiFuMaX::analyze_sat_answer(Solver& solver,
                                 bool is_sat,
                                 vector<size_t>& core) {
  if (is_sat) {
     solver.model.copyTo(model);
  } else {
     const vec<Lit> &conflict_clause=solver.conflict;
     for (int index = 0; index< conflict_clause.size(); ++index) {
       const Var conflict_variable = var(conflict_clause[index]);
       assert(sign(conflict_clause[index])); // control forced to false
       assert(contains(control2index, conflict_variable));
       const size_t clause_index = control2index[conflict_variable];
       assert(clause_index<soft_clauses.size());
       assert(!removed_soft_clauses[clause_index]);
       core.push_back(clause_index);
     }
  }
}
```

17c ⟨*Private members* 9a⟩+≡
```
void initialize();
Var relax_clause(size_t clause_index);
void add_at_most_1(const vector<Var>& relaxation_variables);
void populate_solver(Solver& solver);
bool sat_call(vector<size_t> &core, vec<lbool> &model );
void remove_soft_clause(size_t clause_index, Solver& sat_solver);
void analyze_sat_answer(Solver& solver, bool is_sat, vector<size_t>& core);
```

Chapter 3

MiFuMaXWeighted

3.1 Introduction

The problem of *weighted MaxSAT* is very much similar to the (partial) MaxSAT and we will use a similar algorithm to solve it. As the name suggests, the difference between unweighted MaxSAT and weighted MaxSAT is that soft clauses are labeled with weights. We will write (w, C) to denote a clause C with a weight w. To solve the problem means to find a satisfiable subset of the given clauses such that the *total weight* of the clauses not in the solution is minimal. In the *partial* version of weighted MaxSAT some clauses are marked as hard, which means they must appear in any solution.

For illustration consider a partial weighted MaxSAT with a single hard clause $x \vee y$ and two soft clause $(2, \neg x)$ and $(10, \neg y)$, with their respective weights 2 and 10. The solution is $x = \text{False}, y = \text{True}$ (and it is the only solution in this case). Note that any instance of weighted MaxSAT can be converted to an instance of unweighted MaxSAT by setting all the weights of soft clauses to 1.

We will solve the weighted partial MaxSAT problem by an algorithm developed by Manquinho et al. [MSP09]. This algorithm is a natural extension of the algorithm proposed by Fu and Malik [FM06] so the reader is adviced to study that algorithm first.

3.2 Public Interface

The algorithm is encapsulated in a class `MiFuMaXWeighted` relying on the types and utilities declared in `common`. Minisat [ES03] is used as the underlying SAT solver. Similar constructs as in `MiFuMaX` are used.

19 ⟨*MiFuMaXWeighted.hh* 19⟩≡

```
#ifndef MIFUMAXWEIGHTED_W_24714
#define MIFUMAXWEIGHTED_W_24714
⟨Header includes 33b⟩;
namespace Mifumax {
```

```
class MiFuMaXWeighted {
public: ⟨Public members 6c⟩
private: ⟨Private members 9a⟩
};
}
#endif
```

Root chunk (not used in this document).

20a ⟨*MiFuMaXWeighted.cc* 20a⟩≡
⟨*Implementation includes* 6b⟩
⟨*Implementation* 8⟩

Root chunk (not used in this document).

20b ⟨*Implementation includes* 6b⟩+≡
```
#include "MiFuMaXWeighted.hh"
using namespace Mifumax;
```

The class provides a constructor accepting a vector of hard clauses and a vector of weighted soft clauses.

20c ⟨*Public members* 6c⟩+≡
```
MiFuMaXWeighted(Var max_id,
                const ClauseVector& hard_cnf,
                const WeightedClauseVector& soft_cnf);
```

To run the solver, one calls `solve`, which upon success returns whether or not the given problem has a solution, i.e. if the hard clauses are satisfiable.

20d ⟨*Public members* 6c⟩+≡
```
bool solve();
```

If a solution was found, i.e. `solve()` returned `true`, the total weight of the unsatisfied soft clauses by the solution can be obtained by the function `get_optimum` and the solution assignment by `get_solution`.

20e ⟨*Public members* 6c⟩+≡
```
WeightType get_optimum () const {return optimum;}
void get_solution(vec<lbool>& output_model) const;
```

For advanced applications of the solver, some internal data structures are exposed. These methods should be used only if a good understanding of the implementation is had.

21 ⟨*Public members* 6c⟩+≡

```
const Var2Index & get_control2index() const { return control2index; }
const ClauseVector & get_hard_clauses() const { return hard_clauses; }
const WeightedClauseVector & get_soft_clauses() const { return soft_clauses; }
const vector<bool>& get_removed_soft_clauses() const {return removed_soft_clauses;}
const vec<lbool>& get_model_internal() const { return model; }
const Var get_max_id() const { return max_id; }
const Var get_control(size_t soft_clause_index) const {
  assert(soft_clause_index<soft_clauses.size());
  return controls[soft_clause_index];
}
```

3.3 Solving

As noted before, one could solve a weighted MaxSAT by translating it to an *unweighted* MaxSAT making as many copies of each clause as its weight is. Let us pretend for a while that we do that. Now we find some unweighted core $\mathcal{C}$. Let w_m be the smallest weight appearing in the weighted counterpart of the core. This means that in the unweighted version there is at least w_m copies of each of the clauses in $\mathcal{C}$, or in another words, there is at least w_m disjoint copies of the core $\mathcal{C}$. Hence, what we could do is relax each of the cores separately right after we have found the core $\mathcal{C}$ without waiting for the other copies of the core to be computed. This gives us one important optimization. However, we make another important observation and that is that each copy of some clause $C \in \mathcal{C}$ is satisfied (respectively unsatisfied) by the same set of assignments. Hence, if a solution to the given problem removes some clause C, it should also remove all its copies. This means, that the same relaxation variable can be used for all the w_m copies of each clause $C \in \mathcal{C}$.

These observations enable an algorithm that does not require explicitly creating all the different copies of clauses. Whenever a core $\mathcal{C}$ is found, we compute the minimum weight w_m appearing in it. Subsequently, each clause $(w, C) \in \mathcal{C}$ is split into the clauses (w_m, C) and $(w - w_m, C)$. The clause with the weight w_m correspond to the w_m copies discussed above. The clause with weight $w - w_m$ correspond to the surplus copies of that clause (if $w - w_m = 0$, the clause is ignored). In accordance with the discussion above, the clause (w_m, C) is relaxed. The relaxation variables are constrained by the at-most-one constraint just as in the unweighted case.

22 ⟨*Implementation* 8⟩+≡

```
bool MiFuMaXWeighted::solve() {
  ⟨Initialize solve 23a⟩
  while (true) {
    ⟨Initialize solve loop 9c⟩
    const bool formula_sat = ⟨Check if formula SAT 9d⟩;
    if (formula_sat) {
      ⟨Print sat call statistics 24c⟩
      return_value = true;
      break;
    } else {
      ⟨Relax formula 10a⟩;
      ⟨Print core statistics 24b⟩;
    }
  }
  return return_value;
}
```

The `solve` function reuses the local variable `core`, and the object variable `model`, which are cleared at the beginning of each iteration of the loop. The variable `optimum` represents the lower bound for the value of the solution, i.e. the total weight of the unsatisfied clauses in a solution.

23a ⟨*Initialize solve* 23a⟩≡
```
vector<size_t> core;
bool return_value = false;
optimum = 0;
```

This code is used in chunk 22.

23b ⟨*Initialize solve loop* 9c⟩+≡
```
core.clear();
model.clear();
```

To check whether the current formula is satisfiable, we issue a SAT call. If the given formula is satisfiable, `model` will contain the satisfying assignment, and otherwise `core` will represent the core as a set of indices of the clauses in the core.

23c ⟨*Check if formula SAT* 9d⟩+≡
```
sat_call(core, model)
```

Before we relax the core, for the case of partial MaxSAT, we need to check if the core is not empty. If it is empty, it means that the hard clauses themselves are unsatisfiable.

23d ⟨*Relax formula* 10a⟩+≡
```
if (core.empty()) {
  return_value = false;
  break;
}
```

Following the main idea of the algorithm, we compute the minimum weight w_m of the core, split clauses in the core into two and relax those clauses corresponding to the w_m. The relaxation variables must be constrained by an at-most-one constraint for the same reason as in the unweighted case. Since we know that at least one of the clauses just relaxed has to be removed from the solution, we update our lower bound of the optimum by w_m (in the unweighted counterpart of the problem this would mean removing W_M copies of that clause).

24a ⟨*Relax formula* 10a⟩+≡

```
vector<Var> relaxation_variables;
const WeightType m = core_min(core);
optimum+=m;
for (auto iter=core.begin(); iter ≠ core.end(); ++iter) {
  const size_t clause_index = *iter;
  const Var relaxation_variable = split_clause(clause_index, m);
  relaxation_variables.push_back(relaxation_variable);
}
add_at_most_1(relaxation_variables);
```

24b ⟨*Print core statistics* 24b⟩≡

```
std::cerr≪"c Min unsat cost LB: "≪optimum
         ≪", ts: "≪read_cpu_time()
         ≪", CS: "≪core.size()≪std::endl;
```

This code is used in chunks 22 and 28a.

24c ⟨*Print sat call statistics* 24c⟩≡

```
std::cerr≪"c SAT found"
         ≪", ts: "≪read_cpu_time()≪std::endl;
```

This code is used in chunk 22.

3.3.1 Helpers of `solve`

The following function splits a clause (w, C) into the clauses $(w - \mathtt{m}, C)$ and $(\mathtt{m}, r \vee C)$ where $\mathtt{m}$ is the minimum weight appearing it in the core just found and r is a fresh relaxation variable. When $w = \mathtt{m}$, the clause with weight $w - \mathtt{m}$ is ignored. The function returns the new relaxation variable.

25 ⟨*Implementation 8*⟩+≡

```
Var MiFuMaXWeighted::split_clause(size_t clause_index,
                                  const WeightType& m) {
  assert(clause_index < soft_clauses.size());
  const WeightType  w = soft_clauses[clause_index].first;
  const Var relaxation_variable = ++max_id;
  if (w≠m) {
    ⟨Generate new soft clause 26a⟩;
  }
  auto& wclause = soft_clauses[clause_index];
  wclause.first=m;
  wclause.second.push_back(mkLit(relaxation_variable));
  return relaxation_variable;
}
```

Produce a new soft clause, which is a copy of the old one, $(\texttt{w}, C)$, but with the weight $(\texttt{w}-\texttt{m}, C)$. Besides resizing `soft_clauses`, the data structures `controls`, `control2index`, `removed_soft_clauses` need to be updated as well. Once a copy is made, the old control variable needs to be replaced with the new one.

26a ⟨*Generate new soft clause* 26a⟩≡

```
assert(w>m);
const size_t new_clause_index=soft_clauses.size();
const Var old_control = controls[clause_index];
const Var new_control = ++max_id;
controls.push_back(new_control);
assert(controls.size()==(new_clause_index+1));
control2index[new_control]=new_clause_index;
soft_clauses.resize(new_clause_index+1);
removed_soft_clauses.resize(new_clause_index+1,false);
soft_clauses[new_clause_index].first=w-m;
soft_clauses[new_clause_index].second=soft_clauses[clause_index].second;
auto& new_cl=soft_clauses[new_clause_index].second;
bool found=false;// sanity check
for (size_t i=0; i<new_cl.size(); ++i) {
  if (var(new_cl[i])==old_control) {
    new_cl[i]=~mkLit(new_control);
    found=true;
    break;
  }
}
assert(found); // sanity check
```

This code is used in chunk 25.

The following function traverses the core and finds the minimum weight.

26b ⟨*Implementation* 8⟩+≡

```
WeightType MiFuMaXWeighted::core_min(vector<size_t>& core) {
  assert(core.size());
  WeightType min_weight = soft_clauses[core[0]].first;
  for (size_t i=1; i<core.size(); ++i) {
    const size_t clause_index = core[i];
    assert(clause_index<soft_clauses.size());
    const WeightType cw = soft_clauses[clause_index].first;
    if (cw<min_weight) min_weight=cw;
  }
  return min_weight;
}
```

The following function encodes the at-most-1 constraint into CNF, for which an external encoder is used. Here we need to be a little bit careful because the encoder also needs to generate some fresh variables. Here we rely on on the encoder to provide the new maximal ID by its function `get_max_id`.

27a ⟨*Implementation includes* 6b⟩+≡

```
#include "SeqCounter.hh"
```

27b ⟨*Implementation* 8⟩+≡

```
void MiFuMaXWeighted::add_at_most_1(const vector<Var>& relaxation_variables) {
  SeqCounter enc(hard_clauses, relaxation_variables, 1, max_id);
  enc.encode();
  assert(max_id≤enc.get_max_id());
  max_id=enc.get_max_id();
}
```

3.4 SAT Call

Here we will see how to call the SAT solver. Note that the SAT solver does not know anything about the weights associated with the clauses. As in `MiFuMaX` we will use the optimization that if a core with a single soft cause is found, that soft clause can be ignored from then on. In such case, the value of optimum also needs to be updated.

28a ⟨*Implementation* 8⟩+≡

```
bool MiFuMaXWeighted::sat_call(vector<size_t> &core,
                               vec<lbool> &model ) {
  Solver solver;
  model.clear();
  core.clear();
  if (removed_soft_clauses.empty()) { // initialize
    removed_soft_clauses.resize(soft_clauses.size(),false); }
  populate_solver(solver);
  vec<Lit> assumptions;
 again:
  ⟨Build assumptions 17a⟩
  const bool return_value = solver.solve(assumptions);
  analyze_sat_answer(solver, return_value, core);
  if (core.size()==1) {
    const size_t clause_index = core[0];
    optimum+=soft_clauses[clause_index].first;
    remove_soft_clause(clause_index,solver);
    ⟨Print core statistics 24b⟩;
    core.clear();
    assumptions.clear();
    goto again;
  }
  return return_value;
}
```

Through assumptions, we tell the SAT solver that all control variables must be set to true except for removed soft clauses (due to core with a single clause).

28b ⟨*Build assumptions* 17a⟩+≡

```
for (size_t index = 0; index < soft_clauses.size(); ++index) {
  if (!removed_soft_clauses[index])
    assumptions.push(mkLit(controls[index]));
}
```

When building a new solver, we simply allocate variables in the solver and copy both hard and soft clauses into it (recall that the relaxation variables are already part of the soft clauses).

29a ⟨*Implementation* 8⟩+≡

```
void MiFuMaXWeighted::populate_solver(Solver& solver) {
  new_variables(solver, max_id);
  add_all(solver, soft_clauses);
  add_all(solver, hard_clauses);
}
```

If the SAT solver returned satisfiable, we just need to copy the model from the solver into the variable `model`. If, the problem is unsatisfiable, we construct a core based off the conflict clause in the solver.

29b ⟨*Implementation* 8⟩+≡

```
void MiFuMaXWeighted::analyze_sat_answer(Solver& solver,
                                         bool is_sat,
                                         vector<size_t>& core) {
  if (is_sat) {
    solver.model.copyTo(model);
  } else {
    const vec<Lit>& conflict_clause=solver.conflict;
    for (int index = 0; index< conflict_clause.size(); ++index) {
      const Var conflict_variable = var(conflict_clause[index]);
      assert(contains(control2index, conflict_variable));
      core.push_back(control2index[conflict_variable]);
    }
  }
}
```

In order to remove a soft clause (w, C), we add to the hard clauses the negation of its control variable $\neg s_C$ and mark it as removed. It is very important that removed clauses are marked because their control variables must not be set to true by assumptions. Removed clauses are stored in a bitvector `removed_soft_clauses` whose size is equal to the size of the `soft_clauses` vector. Just as in `MiFuMaX`, $\neg C$ is added to hard clauses.

29c ⟨*Private members* 9a⟩+≡

```
vector<bool> removed_soft_clauses;
```

30a ⟨*Implementation* 8⟩+≡

```
void MiFuMaXWeighted::remove_soft_clause(size_t clause_index,
                                         Solver& sat_solver) {
  assert (clause_index < soft_clauses.size());
  const vector<Lit>& c = soft_clauses[clause_index].second;
  const Var cv=controls[clause_index];
  removed_soft_clauses[clause_index]=true; //marked as removed
  size_t osz=hard_clauses.size();
  hard_clauses.resize(osz+c.size()); // we add c.size hard cls
  hard_clauses[osz++].push_back(~mkLit(cv)); // satisfy control forever
  sat_solver.addClause(~mkLit(cv));

  bool found=false;// sanity check purpose
  for (size_t i=0; i<c.size(); ++i) {
    const Lit l = c[i];
    if (var(l)==cv) {// found control variable
      found = true;
      assert(sign(l));
      continue;
    }
    sat_solver.addClause(~l);
    hard_clauses[osz++].push_back(~l);
  }
  assert(found);// sanity check
}
```

3.5 Internal State And Functions

The rest of the code is very similar to the one in `MiFuMaX` main difference is that weights of clauses also need to be bookkept.

The variable `original_max_id` is initialized at the beginning of the lifetime of an object and does not change afterwards and corresponds to the maximal variable found in the input formula. In contrast, `max_id` is increased throughout the lifetime of the object whenever a new variable is needed.

30b ⟨*Private members* 9a⟩+≡

```
Var original_max_id;
Var max_id;
```

Clauses are split into *hard* and *soft*. A hard clause is given to the SAT solver as it is. For each soft clause c we remember a *control* variable v_c. Each soft clause c then is maintained in the form $\neg v_c \vee c$. When the SAT solver is called, v_c is forced to true by assumptions. Like this, whenever the considered formula is shown unsatisfiable, the last conflict clause obtained from the SAT solver will contain those control variables that participated in the conflict, which lets us reconstruct the core.

The variable `soft_clauses` maintains soft clauses in the form they will be given to the SAT solver, i.e. they already contain the control variable. Hard clauses are maintained in the variable `hard_clauses`. New soft clauses are generated from old soft clauses by splitting during relaxation. Hard clauses are not modified during the lifetime of the object but new hard clauses are being added to express cardinality constraints on relaxation variables.

31a ⟨*Private members* 9a⟩+≡
```
ClauseVector             hard_clauses;
WeightedClauseVector     soft_clauses;
```

The variable `controls` maintains the control variables for soft clauses and it holds that `controls[i]` is the control variables for the clause `soft_clauses[i]` for `0<=i<soft_clauses.size()`. In order to reconstruct an unsatisfiable core from the conflict clause given by the SAT solver, we also need a map from control variables to the corresponding clauses. It holds that for control variable `v`, `control2index[v]` is the index of the corresponding clause in the vector `soft_clauses`.

31b ⟨*Private members* 9a⟩+≡
```
vector<Var> controls;
Var2Index control2index;
```

When `solve` is successful and the instance satisfiable, stores the optimum and the model.

31c ⟨*Private members* 9a⟩+≡
```
WeightType optimum;
vec<lbool> model;
```

Makes a copy of the hard and soft clauses. Soft clauses will be adorned with control variables later on in the constructor.

32a ⟨*Implementation* 8⟩+≡

```
MiFuMaXWeighted::MiFuMaXWeighted(Var _max_id,
  const ClauseVector &_hard_clauses,
  const WeightedClauseVector &_soft_clauses)
: original_max_id(_max_id)
, hard_clauses(_hard_clauses)
, soft_clauses(_soft_clauses)
{initialize();}
```

Implementation of initialization procedure used in the constructor.

32b ⟨*Implementation* 8⟩+≡

```
void MiFuMaXWeighted::initialize() {
  max_id = original_max_id; // set max ID to the max ID of the input formula
  ⟨Generate control variables 32c⟩;
}
```

32c ⟨*Generate control variables* 32c⟩≡

```
controls.resize(soft_clauses.size());
for (size_t index = 0; index < soft_clauses.size(); ++index) {
  const Var control = ++max_id;
  soft_clauses[index].second.push_back(~mkLit(control));
  controls[index]=control;
  control2index[control] = index;
}
```

This code is used in chunk 32b.

32d ⟨*Implementation* 8⟩+≡

```
void MiFuMaXWeighted::get_solution(vec<lbool> &output_model) const {
  assert(model.size()≥original_max_id+1);
  output_model.growTo(original_max_id+1, l_Undef);
  for (Var v=1; v≤original_max_id; ++v) output_model[v]=model[v];
}
```

33a ⟨*Implementation includes* 6b⟩+≡
```
#include "MiFuMaXWeighted.hh"
#include "SeqCounter.hh"
#include "BitWise.hh"
#include "Utils.hh"
#include "core/Solver.h"
#include <sys/time.h>
#include <sys/resource.h>
using Minisat::Solver;
using namespace Mifumax;
```

33b ⟨*Header includes* 33b⟩≡
```
#include <vector>
#include <unordered_map>
#include "core/SolverTypes.h"
#include "core/Solver.h"
#include "MiFuMaXTypes.hh"
#include "Utils.hh"
using std::unordered_map;
using std::vector;
using std::pair;
using Minisat::Var;
using Minisat::Lit;
using Minisat::vec;
using Minisat::lbool;
using Minisat::mkLit;
using Minisat::var;
```

This code is used in chunk 19.

33c ⟨*Private members* 9a⟩+≡
```
bool sat_call( vector<size_t> &core, vec<lbool> &model );
void initialize();
Var split_clause(size_t clause_index, const WeightType& m);
WeightType core_min(vector<size_t>& core);
void add_at_most_1(const vector<Var>& relaxation_variables);
void populate_solver(Minisat::Solver& solver);
void analyze_sat_answer(Minisat::Solver& solver, bool is_sat,
                                            vector<size_t>& core);
void remove_soft_clause(size_t clause_index,
                        Minisat::Solver& sat_solver);
```

Bibliography

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, pages 502–518, 2003.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In Armin Biere and Carla P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006.

[MSP09] Vasco M. Manquinho, João P. Marques Silva, and Jordi Planes. Algorithms for weighted boolean optimization. In Oliver Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 495–508. Springer, 2009.