Sugar++: A SAT-Based MAX-CSP/COP Solver

Tomoya Tanjo[1], Naoyuki Tamura[2], and Mutsunori Banbara[2]

[1] Graduate School of Engineering, Kobe University, JAPAN
tanjo@stu.kobe-u.ac.jp
[2] Information Science and Technology Center, Kobe University, JAPAN
{tamura,banbara}@kobe-u.ac.jp

1 Introduction

This paper briefly describes some features of Sugar++, a SAT-based MAX-CSP/COP solver entering the Third International CSP Solver Competition.

In our approach, a MAX-CSP is translated into a Constraint Optimization Problem (COP), and then it is encoded into a SAT problem by the *order encoding* method [1]. SAT-encoded COP can be efficiently solved by repeatedly using the MiniSat solver [2].

The order encoding method encodes a comparison $x \leq a$ by a different boolean variable for each integer variable x and integer value a. Sugar [3] is a SAT-based CSP solver based on order encoding, and its effectiveness has been shown through application to Open-Shop Scheduling in [1].

A solution to COP can be obtained by repeatedly solving a certain number of CSPs. In Sugar, each CSP is solved by a different MiniSat process independently. Therefore the learnt clauses generated in each process disappear and can not be reused. This slows down the execution speed.

To solve this problem, we have developed a SAT-based MAX-CSP/COP solver Sugar++ that is an enhancement of Sugar by using the incremental search feature of MiniSat. Sugar++ invokes only one MiniSat process to solve MAX-CSP/COP and can reuse the learnt clauses generated during the search.

2 Translating MAX-CSP into COP

Definition 1 (CSP). A Constraint Satisfaction Problem (CSP) is defined as a tuple $(V, d, \mathcal{C})$ where

- $V = \{x_1, \ldots, x_m\}$ is a set of variables,
- d is a mapping from x_i to a finite domain D_i containing possible values x_i may take, and
- $\mathcal{C} = \{C_1, \ldots, C_n\}$ is a set of constraints.

A *solution* to a CSP is an assignment α, a mapping from every variable $x_i \in V$ to an element of D_i so that every constraint in $\mathcal{C}$ is satisfied.

Definition 2 (COP). A Constraint Optimization Problem (COP) is defined as a tuple $(V, d, \mathcal{C}, v)$ where

– $(V, d, \mathcal{C})$ is a CSP, and
– $v \in V$ is an objective variable to be minimized [3].

A *solution* to a COP$(V, d, \mathcal{C}, v)$ is an assignment α which is a solution of CSP$(V, d, \mathcal{C})$ taking the minimum $\alpha(v)$ value.

Given a CSP, the Max-CSP problem is to find an assignment that minimizes the number of violated constraints. Therefore, the Max-CSP problem for a CSP$(V, d, \mathcal{C})$ can be translated into a COP$(V^*, d^*, \mathcal{C}^*, \mathit{cost})$ where

– $V^* = V \cup \{c_1, \ldots, c_n, \mathit{cost}\}$
  Each c_i represents the penalty of the constraint C_i, and *cost* is the objective variable to be minimized.

– $d^*(x) = \begin{cases} \{0, 1\} & \text{if } x = c_i \quad (1 \le i \le n) \\ \{0, \ldots, n\} & \text{if } x = \mathit{cost} \\ d(x) & \text{otherwise} \end{cases}$

– $\mathcal{C}^* = \{(c_i \ge 1) \vee C_i \mid 1 \le i \le n\} \cup \{\mathit{cost} \ge \sum c_i\}$

It is also possible to translate Weighted-CSP into COP in the same way by replacing $d^*(\mathit{cost})$ with $\{0, \ldots, \sum w_i\}$ and $\mathit{cost} \ge \sum c_i$ with $\mathit{cost} \ge \sum w_i c_i$ where w_i is the given positive weight for the constraint C_i.

3 Solving COP by using the incremental search of SAT solver

The `Sugar` constraint solver [3] can solve a finite linear COP by encoding it into a SAT problem based on the *order encoding* method [1], and then a SAT-encoded COP is solved by using the MiniSat solver [2]. The main feature of order encoding can be the natural representation of order relation on integers by encoding a comparison $x \le a$ into a different boolean variable for each integer variable x and integer value a. In the followings, "$p(x, a)$" is used to represent the boolean variable for the comparison $x \le a$.

The optimal value of COP$(V, d, \mathcal{C}, v)$ can be obtained by repeatedly solving CSPs.

$$\min \{a \in d(v) \mid \mathit{CSP}(V, d, \mathcal{C} \cup \{v \le a\}) \textit{ has a solution}\}$$

A solution to COP can be efficiently found by bisection search with varying a as proposed in previous works [4–6].

Let P be a COP whose objective variable is v. Fig. 1 shows the minimization procedure of `Sugar` to find the optimal value of P. The outline of `minimize(`P`, `v`)` is as follows:

(1) Encodes P into a SAT problem and sets `S` to it. The `S` represents a SAT file in the actual implementation.

[3] Without the loss of generality, we assume COPs as minimization problems.

```
procedure minimize(P, v);
begin
  S := SAT encoding of P;
  found := false;
  lb := lower bound of v;
  ub := upper bound of v + 1;
  while lb < ub do
    a := ⌊(lb+ub)/2⌋;
    c := {p(v, a)};
    result := execute MiniSat for S ∪ {c};
    if result is satisfiable then
      found := true;
      ub := a;
    else
      lb := a + 1;
    end if
  end while
  if found then
    OPTIMUM FOUND;
  else
    UNSATISFIABLE;
  end if
end
```

Fig. 1. The minimization procedure of `Sugar`

(2) Sets `found` to `false`. The `found` is a flag indicating whether a solution is found or not.
(3) Sets `lb` and `ub` to v's lower bound and v's upper bound + 1 respectively.
(4) If `lb` $<$ `ub` does not hold, goes to the step (10).
(5) Sets `a` to the value of $\lfloor$(`lb`+`ub`)$/2\rfloor$.
(6) Sets `c` to a unit clause $\{p(v, \texttt{a})\}$ where $p(v, \texttt{a})$ represents the boolean variable for the comparison $v \leq$ `a`.
(7) Executes the MiniSat with `S` $\cup \{$`c`$\}$ as an input SAT problem.
(8) Updates `ub` to the value of `a` and also sets `found` to `true` if the result is satisfiable. Otherwise updates `lb` to the value of `a`+1.
(9) Goes back to the step (4).
(10) If `found` is `true`, this procedure succeeds to find the optimal value of P. Otherwise fails.

`Sugar` encodes a COP into a SAT once at first, and repeatedly modifies only the clause corresponding $v \leq a$. However there has still remained some points to be improved.

- A number of MiniSat processes are invoked until the optimal value is found.
- The learnt clauses generated in each MiniSat process are not reused.

Reusing learnt clauses is effective to significantly reduce the search space. It is therefore very important to reuse learnt clauses. To solve this problem, we take advantage of the incremental search of MiniSat [2].

First, we modify the MiniSat so that it can deal with the following three commands from the standard input, where L_i means a literal:

- `add` $L_1\ L_2\ \cdots\ L_n$
  This command adds a clause $\{L_1, L_2, \ldots, L_n\}$ to the SAT clause database. The clause is never removed, and the conflict analysis in the further search uses this clause as well as those in the initial database.
- `solve` $L_1\ L_2\ \cdots\ L_m$
  This command makes the MiniSat to solve the SAT problem with an assumption $\{L_1\ L_2\ \cdots\ L_m\}$. This assumption is passed as an argument to the solve method of MiniSat and temporarily assumed to be true during the search. After solving the problem, this assumption is undone. It is noted that the assumption does not affect learnt clauses to be generated since the MiniSat's conflict detecting mechanism is independent of it.
- `exit`
  This command makes the MiniSat terminate.

We have developed the `Sugar++` system that is an enhancement of `Sugar` to use the incremental version of MiniSat described above. The main features of `Sugar++` are as follows:

- Bi-directional IO is used to communicate between `Sugar++` and the incremental version of MiniSat.
- MiniSat is invoked only once, therefore, SAT file is parsed only once.
- Learnt clauses are reused during the search.

Fig. 2 shows the new minimization procedure of `Sugar++`. The outline of `minimize(`P`,` v`)` is as follows:

(1) Encodes P into a SAT problem and sets `S` to it. The `S` represents a SAT file in the actual implementation.
(2) Sets `found` to `false`. The `found` is a flag indicating whether a solution is found or not.
(3) Sets `lb` and `ub` to v's lower bound and v's upper bound + 1 respectively.
(4) Starts the incremental version of MiniSat process with `S`, and the process waits for the commands: `add`, `solve`, and `exit`.
(5) If `lb` $<$ `ub` does not hold, goes to the step (10).
(6) Sets `a` to the value of $\lfloor$(`lb`+`ub`)$/2\rfloor$.
(7) Sends the command '`solve` $p(v,$ `a`$)$' to the MiniSat process.
(8) Sends the command '`add` $p(v,$ `a`$)$' to the MiniSat process after updating `ub` to the value of `a` and setting `found` to `true` if the result is satisfiable. Otherwise updates `lb` to the value of `a`+1 and sends the command '`add -`$p(v,$ `a`$)$'.
(9) Goes back to the step (5).
(10) Sends the command '`exit`' to the MiniSat process.

```
procedure minimize(P, v);
begin
  S := SAT encoding of P;
  found := false;
  lb := lower bound of v;
  ub := upper bound of v + 1;
  Start a MiniSat process with S;
  while lb < ub do
    a := ⌊(lb+ub)/2⌋;
    Send 'solve p(v, a)' to MiniSat;
    result := receive the result from MiniSat;
    if result is satisfiable then
      found := true;
      ub := a;
      Send 'add p(v, a)' to MiniSat;
    else
      lb := a + 1;
      Send 'add -p(v, a)' to MiniSat;
    end if
  end while
  Send 'exit' to MiniSat;
  if found then
    OPTIMUM FOUND;
  else
    UNSATISFIABLE;
  end if
end
```

Fig. 2. The new minimization procedure of `Sugar++`

(11) If `found` is `true`, this procedure succeeds to find the optimal value of P. Otherwise fails.

Compared with `Sugar`, `Sugar++` succeeds not only in reducing the overhead of parsing SAT problems but also in reusing learnt clauses during the search.

4 Conclusion

In this paper, we have described some features of `Sugar++`, a SAT-based MAX-CSP/COP solver entering the Third International CSP Solver Competition. `Sugar++` solves a MAX-CSP by translating it into a COP, and encoding it into a SAT problem by the *order encoding* method. SAT-encoded COP is solved by using the incremental version of MiniSat solver.

Acknowledgments

We would like to give thanks to the competition organizers for their efforts.

References

1. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. In: Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP 2006). (2006) 590–603
2. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003). (2003) 502–518
3. Tamura, N., Banbara, M.: Sugar: A CSP to SAT translator based on order encoding. In: Proceedings of the Second International CSP Solver Competition. (2008) 65–69
4. Soh, T., Inoue, K., Banbara, M., Tamura, N.: Experimental results for solving job-shop scheduling problems with multiple SAT solvers. In: Proceedings of the 1st International Workshop on Distributed and Speculative Constraint Processing (DSCP'05). (2005)
5. Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M., Tamura, N.: A competitive and cooperative approach to propositional satisfiability. Discrete Applied Mathematics **154** (2006) 2291–2306
6. Nabeshima, H., Soh, T., Inoue, K., Iwanuma, K.: Lemma reusing for SAT based planning and scheduling. In: Proceedings of the International Conference on Automated Planning and Scheduling 2006 (ICAPS'06). (2006) 103–112