Minimal Model Generation with respect to an Atom Set

Miyuki Koshimura[1]*, Hidetomo Nabeshima[2], Hiroshi Fujita[1], and Ryuzo Hasegawa[1]

[1] Kyushu University, Motooka 744, Nishi-ku, Fukuoka, 819-0395 Japan, {koshi,fujita,hasegawa}@ar.is.kyushu-u.ac.jp,
[2] University of Yamanashi, Takeda 4-3-1, Kofu, 400-8511 Japan, nabesima@yamanashi.ac.jp

Abstract. This paper studies minimal model generation for SAT instances. In this study, we minimize models with respect to an atom set, and not to the whole atom set. In order to enumerate minimal models, we use an arbitrary SAT solver as a subroutine which returns models of satisfiable SAT instances. In this way, we benefit from the year-by-year progress of efficient SAT solvers for generating minimal models. As an application, we try to solve job-shop scheduling problems by encoding them into SAT instances whose minimal models represent optimum solutions.

1 Introduction

The notion of minimal Herbrand models is important in a wide range of areas such as logic programming, deductive database, software verification, and hypothetical reasoning. Some applications would actually need to generate minimal models of a given formula.

In this work, we consider the problem of automating propositional minimal model generation with respect to an atom set. Some earlier works [3, 14, 9] considered minimal model generation with respect to the whole atom set.

Bry and Yahya [3] presented a sound and complete procedure for generating minimal models. They incorporate complement splitting and constrained search into positive unit hyper-resolution in order to reject nonminimal models. Niemelä [14] also gave a sound and complete procedure. His method is based on a generate and test method: generate a sequence of minimal model candidates and reject nonminimal models by groundedness test which passes minimal models. Hasegawa et al. [9] presented an minimal model generation method employing branching assumptions and lemmas so as to prune branches that lead to nonminimal models, and to reduce minimality tests on obtained models.

However, these earlier works do not make use of some pruning techniques such as non-chronological or intelligent backtracking, and generating lemmas. These techniques make reasoning systems practical ones. In recent years, the

* This work was supported by KAKENHI (20240003).

propositional satisfiability (SAT) problem has been studied actively [1]. Specially, many works for implementing efficient SAT solvers have been performed in the last decade. The state-of-the-art SAT solvers can solve SAT problems consisting of millions of clauses in a few minutes. Then, it has been realized that we solve several kinds of problems by encoding them into SAT problems [7, 2].

This paper shows a method to generate minimal models with a SAT solver. Thus, the method benefits from the year-by-year progress of SAT solvers implementing the pruning techniques efficiently. We also try to solve the job-shop scheduling problems (JSSP) in the minimal model generation framework, in which, minimal models represent optimum, namely, the shortest schedules.

The remaining part of this paper is organized as follows: First we present a characterization of minimal models that is the key to our method to handle minimal model generation. Section 3 gives minimal model inference procedures with a SAT solver. Section 4 describes the job-shop scheduling problem and encodes it as a SAT instance. Section 5 demonstrates that the procedures are successfully implemented with the SAT solver MiniSat 2 by solving several JSSPs. We end the paper with a short summary and a discussion of future works.

2 Properties of Minimal Models

Models of a propositional formula can be represented by a set of propositional variables (or atoms); namely, each model is represented by the set of propositional variables to which it assigns true. For example, the model assigning true to a, false to b, and true to c is represented by the set $\{a, c\}$. In this representation, we can compare two models by set inclusion. For example, model $\{a, c\}$ is smaller than model $\{a, b, c\}$. In this study, we focus on minimality of models in the representation.

Definition 1. *Let P, M_1 and M_2 be atom sets. Then, M_1 is said to be smaller than M_2 with respect to P if $M_1 \cap P$ is a proper subset of $M_2 \cap P$.*

Example 1. Let $M_1 = \{p_1, p_2, p_3, a\}$, $M_2 = \{p_1, p_3, b, c, e, f\}$ and $P = \{p_1, p_2, p_3\}$. Then, M_2 is smaller than M_1 with respect to P.

Definition 2 (Minimal model). *Let A be a propositional formula, P be an atom set, and M be a model of A. Then, M is said to be a minimal model of A with respect to P when there is no model smaller than M with respect to P.*

Example 2. Let A be a propositional formula and $P = \{p_1, p_2, p_3\}$. And, A has three models $M_1 = \{p_1, p_2, p_3, a\}$, $M_2 = \{p_1, p_2, b\}$, and $M_3 = \{p_3, c\}$. Then, M_2 and M_3 are minimal models with respect to P while M_1 is not minimal.

This definition is the same as that of circumscription $Circum(A(P, Z); P; Z)$ [10] when variable predicates $Z = \overline{P}$, i.e. no fixed predicate. Note that $\overline{P}$ denotes the set complement of P. In this sense, our study is a specialized one of circumscription. There is a little difference between our work and circumscription for the treatment of models. We are interested only in truth values of atoms in P.

Therefore, we regard two models M_1 and M_2 as equal when $M_1 \cap P = M_2 \cap P$, while these two are distinguished in the framework of circumscription when $M_1 \neq M_2$.

The following theorem is a straight extension of **Proposition 6** in Niemelä's work [14]. This theorem gives the basis of the computational treatment of minimal models as **Proposition 6** does.

Theorem 1. *Let A be a propositional formula, P be an atom set, and M be a model of A. Then, M is a minimal model of A with respect to P iff a formula $A \wedge \neg(a_1 \wedge a_2 \wedge \ldots \wedge a_m) \wedge \neg b_1 \wedge \neg b_2 \wedge \ldots \wedge \neg b_n$ is unsatisfiable, where $\{a_1, a_2, \ldots, a_m\} = M \cap P$ and $\{b_1, b_2, \ldots, b_n\} = \overline{M} \cap P$.*

Proof. Let G be $A \wedge \neg(a_1 \wedge a_2 \wedge \ldots \wedge a_m) \wedge \neg b_1 \wedge \neg b_2 \wedge \ldots \wedge \neg b_n$.
Assume that M is not a minimal model. Then, there is a model N smaller than M with respect to P. Thus, the following properties hold: $\forall j(1 \leq j \leq n)(N \models \neg b_j)$ and $\exists i(1 \leq i \leq m)(N \models \neg a_i)$. Of course, $N \models A$ because N is a model of A. Therefore, $N \models G$; namely G is satisfiable.
Conversely, we assume G is satisfiable. Then, there is a model N such that $\forall j(1 \leq j \leq n)(N \models \neg b_j)$ and $\exists i(1 \leq i \leq m)(N \models \neg a_i)$. This implies N is smaller than M with respect to P. That is, M is not a minimal model with respect to P.

Example 3. Let A be a propositional formula and $P = \{p_1, p_2, p_3, p_4\}$. Then, a model $\{p_1, p_4, c, d\}$ of A is minimal with respect to P iff $A \wedge \neg(p_1 \wedge p_4) \wedge \neg p_2 \wedge \neg p_3$ is unsatisfiable.

3 Procedures

This section gives procedures for generating minimal models of a SAT instance with a SAT solver based on the generate and test method: generating a sequence $M_1, \ldots, M_i, \ldots$ of models and performing minimality test on each M_i. In these procedures, we use a single SAT solver as both generator and tester where we assume the SAT solver returns a model of a satisfiable SAT instance. Almost all SAT solvers satisfy this assumption.

Figure 1 (a) shows a minimal model generation with respect to an atom set P which is implicitly given to the procedure. We call this *the naive version.* `A0` is a SAT instance to be proved. The function `solve(A)` denotes the core part of the SAT solver. The function returns `false` when a SAT instance `A` is unsatisfiable and `true` when `A` is satisfiable. In the latter case, a model M of `A` is obtained through an array from which we construct two formulas `F1` and `F2` for a minimality test on M with respect to P where `F1` $= \neg(a_1 \wedge \ldots \wedge a_m)$ and `F2` $= \neg b_1 \wedge \ldots \wedge \neg b_n$. A boolean variable `exhaustive` indicates whether the procedure generates all minimal models or only one minimal model. If `exhaustive` is set to *true*, all minimal models are generated.

If `solve(A)` in line (2) returns *true*, the body of the while statement is executed. In this case, as a model M of `A` is obtained, we perform a minimality test on M (in (4)). If the test passes, that is `solve(A)` in (4) returns $false$, we conclude M is minimal with respect to P. If the test fails or `exhaustive` is $true$, $\mathtt{F_1}$ is added to $\mathtt{A_1}$ as a conjunct in order to avoid generating the same model or larger models in succeeding search. Thus, the role of the conjunct $\mathtt{F_1}$ is pruning redundant models.

```
(1)  $A = A_0$; $A_1 = A_0$; // $A_0$: a SAT instance to be proved
(2)  while (solve(A)) { // Found a model $M$ where $M \cap P = \{a_1, \dots, a_m\}$
                          // and $\overline{M} \cap P = \{b_1, \dots, b_n\}$
(3)       $A = A \wedge F_1 \wedge F_2$;   //  $F_1 = \neg(a_1 \wedge \dots \wedge a_m)$, $F_2 = \neg b_1 \wedge \dots \wedge \neg b_n$
(4)       if (!solve(A)) { // Perform minimality test
(5)            "minimal model found";
(6)            if (!exhaustive) break;
(7)       }
(8)       $A_1 = A_1 \wedge F_1$; $A = A_1$; // continue searching minimal models
                                  // without generating larger models.
(9)  }
```

(a) Naive version

```
(1)  $A = A_0$;
(2)  while (solve(A)) {       // Found a model M,
(3)       MM = minimize(A, M); // minimize M, and obtain a minimal model MM
(4)       if (!exhaustive) break;
(5)       $A = A \wedge F_1$;        // $MM \cap P = \{a_1, \dots, a_m\}$ and $F_1 = \neg(a_1 \wedge \dots \wedge a_m)$
(6)  }

(7)  function minimize(A,M) {
     // returns a minimal model small than or equal to M
     // where $M \cap P = \{a_1, \dots, a_m\}$ and $\overline{M} \cap P = \{b_1, \dots, b_n\}$
(8)       $A = A \wedge F_1 \wedge F_2$;   // $F_1 = \neg(a_1 \wedge \dots \wedge a_m)$, $F_2 = \neg b_1 \wedge \dots \wedge \neg b_n$
(9)       if(!solve(A)) { // Perform minimality test
(10)           return M;    // M is a minimal model
(11)      } else {                // Found a new model SM smaller than M
(12)           minimize(A, SM); // and minimize SM
(13) }    }
```

(b) Normal version

Fig. 1. Procedures for minimal model generation

Figure 1 (b) shows a modified procedure of the naive version. We call this *the normal version.* In this version, when a model is found, we minimize it with the function `minimize`. Its definition is shown from the line (7) to (13). This uses the result of the minimality test on `A` in (9). When the test fails, in other words, `solve(A)` in (9) returns *true*, we obtain a model `SM` of `A`. `SM` is smaller than `M` because of the conjuncts $\mathtt{F_1}$ and $\mathtt{F_2}$. Thus, `SM` is the next target of `minimize`. Note that $\exists i(1 \leq i \leq m)(a_i \notin \texttt{SM})$. Therefore, at least one current $\neg a_i$ participates in $\texttt{F}_2$ of the next `minimize`.

The major difference between the naive version and the normal version is the use of `SM` obtained from the minimality test. The naive version ignores it while the normal version uses it. Therefore, we expect that the normal version is more efficient than the naive version for enumerating minimal models.

3.1 Lemma Reusing

Many state-of-the-art SAT solvers learn *lemmas* called conflict clauses to prune redundant search space, but lemmas deduced from a certain SAT instance can not apply to solve other SAT instances. Therefore, a function call `solve(A)` in Figure 1 (both (a) and (b)) can not use lemmas deduced from previous `solve(A)` in general.

However, every SAT instance `A` in `solve(A)` satisfies the following lemma-reusability condition [13] if the conjunct $\texttt{F}_1$ is not added to `A`, when the SAT solver uses Chaff-like lemma generation mechanism [12].

Definition 3 (Lemma-reusability condition [13]). *Suppose that A and B are SAT instances. The lemma-reusability condition between A and B is as follows: If A includes a non-unit clause x, then B contains x.*

If both A and B satisfy the condition, we can use lemmas generated by `solve(A)` for `solve(B)`. This is justified by the following proposition which is a paraphrase of Theorem 1 in [13].

Proposition 1. *If A is a SAT instance and c is any lemma generated by `solve(A)`, then c is a logical consequence of a set of some non-unit clauses in A.*

This proposition is true when we use the SAT solver MiniSat for implementing `solve(A)`, because MiniSat does not use any unit clause for generating lemmas.

$\texttt{F}_1$ is a non-unit clause and violates the lemma-reusability condition. However, the only role of $\texttt{F}_1$ is excluding models larger than the model causing $\texttt{F}_1$. Then, lemmas depending on $\texttt{F}_1$ can be used for succeeding minimal model generation. It follows from what has been said that every call `solve(A)` shares lemmas each other.

3.2 An Implementation with MiniSat

We have implemented the minimal model generation procedures with the SAT solver MiniSat [5] version 2.1 which is written in C++. MiniSat 2.1 took the first place in the main track of SAT-Race 2008.

The `solve` method of MiniSat is declared as follows:

```
    bool solve(const vec<Lit>& assumps)
```

The method determines the satisfiability of a set of clauses under an assumption `assumps`. It returns `true` if the set is satisfiable; otherwise `false`. The clause set is realized by a vector `clauses` and initialized to a SAT instance ($\mathtt{A}_0$ in Figure 1). The assumption `assumps` is a vector of literals which means the conjunction of the literals.

In our implementation, the clause $\mathtt{F}_1$ is appended to `clauses` and the formula $\mathtt{F}_2$ is set to `assumps`. Then, the `solve` method is invoked. We don't need to remove $\mathtt{F}_1$ from `clauses` before the next `solve` invocation because the role of $\mathtt{F}_1$ is excluding models larger than the model causing $\mathtt{F}_1$. If $\mathtt{F}_2$ is appended to `clauses`, we need to remove $\mathtt{F}_2$ from `clauses` before the next invocation. Therefore, we add $\mathtt{F}_2$ to `assumps` instead of `clauses`. Thus, removing $\mathtt{F}_2$ is not necessary.

When we need only one minimal model[3] rather than all minimal models, we can append $\mathtt{F}_2$ to `clauses` without removing $\mathtt{F}_2$ afterward. We also implement such solver based on the normal version and call it *the single-solution version.*

4 Solving the JSSP

A JSSP consists of a set of jobs and a set of machines. Each job is a sequence of operations. Each operation requires the exclusive use of a machine for an uninterrupted duration, i.e. its processing time. A schedule is a set of start times for each operation. The time required to complete all the jobs is called the makespan. The objective of the JSSP is to determine the schedule which minimizes the makespan.

In this study, we follow a variant of the SAT encoding proposed by Crawford and Baker [4]. In the SAT encoding, we assume there is a schedule whose makespan is at most i and generate a SAT instance S_i. If S_i is satisfiable, then the JSSP can complete all the jobs by the makespan i. Therefore, if we find a positive integer k such that S_k is satisfiable and S_{k-1} is unsatisfiable, then the minimum makespan is k.

For minimizing the makespan, Nabeshima et al. [13] applied two kinds of methods, *incremental search and binary search.* One can easily estimate the upper bound L_{up} of the minimum makespan by serialising all the operations of all the jobs [4]. The lower bound L_{low} is also easily estimated by taking the maximum length of each job in which we assume every job is performed independently. In the incremental search, we start from L_{low} and increase the makespan by 1 until we encounter the satisfiable instance S_t. If such S_t is found, then the minimum makespan is t. We explain the binary search by an example of $L_{up} = 393$ and $L_{low} = 49$. Firstly, we try to solve S_{221} because 221 is the midpoint between 48 and 393. If S_{221} is satisfiable, then try S_{135}. If S_{135} is unsatisfiable, then try S_{178}. We continue this binary search until we encounter the satisfiable instance S_t and unsatisfiable instance S_{t-1}.

[3] The JSSP is such a problem.

[4] In this study, we use a modified estimation a bit cleverer than this obvious estimation.

In order to solve the JSSP in the minimal model generation framework, we introduce a set $P_u = \{p_1, p_2, \dots, p_u\}$ of new atoms when $L_{up} = u$. The intended meaning of $p_i = \mathit{true}$ is that we found a schedule whose makespan is i or longer than i. To realize the intention, the formulas $F_i(i = 1, \dots, u)$, which represent "if all the operations complete at i, then p_i becomes true," are introduced. Besides, we introduce a formula $T_u = (\neg p_u \vee p_{u-1}) \wedge (\neg p_{u-1} \vee p_{u-2}) \wedge \cdots \wedge (\neg p_2 \vee p_1)$ which implies that $\forall l(1 \leq l < k)(p_l = \mathit{true})$ must hold if $p_k = \mathit{true}$ holds.

In this setting, if we obtain a model M of $G_u(= S_u \wedge F_1 \wedge \cdots \wedge F_u \wedge T_u)$ and k is the maximum integer such that $p_k \in M$, that is, $\forall j(k < j \leq u)(p_j \notin M)$, then we must have $\forall l(1 \leq l \leq k)(p_l \in M)$, namely, $M \cap P_u = \{p_1, \dots, p_k\}$. The existence of such k is guaranteed by F_k and T_u, and indicates that there is a schedule whose makespan is k. If k is the minimum makespan, there is no model of G_u smaller than M with respect to P_u. Thus, a minimal model of G_u with respect to P_u represents a schedule which minimizes the makespan.

Example 4. Given a JSSP with $L_{up} = 10$. Then, we make S_{10} according to Crawford encoding, $P_{10} = \{p_1, \dots, p_{10}\}$, and $T_{10} = (\neg p_{10} \vee p_9) \wedge \cdots \wedge (\neg p_2 \vee p_1)$. Let M be a minimal model of $G_{10}(= S_{10} \wedge F_1 \wedge \cdots \wedge F_{10} \wedge T_{10})$ with respect to P_{10} and $M \cap P_{10} = \{p_1, p_2, p_3\}$. Then, the minimum makespan of the JSSP is 3.

This SAT encoding technique, in which a minimal model represents an optimum solution, is applicable to several problems such as graph coloring problem, open-shop scheduling problem, two dimensional strip packing problem, and so on. Thus, the technique gives a framework to solve these problems.

The encoding is easily adapted for a partial Max-SAT encoding by adding some unit clauses. Max-SAT is the optimization version of SAT where the goal is to find a model satisfying the maximum number of clauses. In order to solve the JSSP in the partial Max-SAT framework[5], we introduce u unit clauses $\neg p_i(i = 1, \dots, u)$. Then, we solve $MAX_u(= G_u \wedge \neg p_1 \wedge \ldots \wedge \neg p_u)$ with a partial Max-SAT solver where all clauses in G_u are treated as hard clauses and $\neg p_i(i = 1, \dots, u)$ are as soft clauses. A Max-SAT model of MAX_u represents a optimum schedule.

Example 5. Let P_{10}, G_{10}, and M be the same as in *Example 4.* Then $MAX_{10} = G_{10} \wedge \neg p_1 \wedge \ldots \wedge \neg p_{10}$ has a (Max-SAT) model M which falsifies only three soft clauses $\neg p_1$, $\neg p_2$, and $\neg p_3$. Note that every model of G_{10} falsifies at least these three clauses.

5 Experiments

We executed the three versions(naive/normal/single-solution), a Max-SAT solver MiniMaxSat[6], and a SAT-based JSSP solver SATSHOP which is a successor of

[5] A partial Max-SAT solver can handle hard clauses and soft clauses. The hard clauses must be satisfied while the soft clauses need not be necessarily satisfied. The goal is to find a model satisfying the all hard clauses and the maximum number of soft clauses.

the JSSP solver proposed in [13]. The MiniMaxSat took the third place in the partial Max-SAT category (industrial) of Max-SAT Evaluation 2008.

The SATSHOP tries to solve a JSSP in the following way. First, making a relaxed problem to improve the upper bound L_{up}. The relaxed problem is an approximation of the original problem. It is obtained by rounding up every operation time. Its optimum solution gives a new upper bound L_{up}^{new} which satisfies $L_{up}^{new} \leq L_{up}$. The relaxed problem is solved with the SAT encoding technique using binary search.

Next, solving the problem with L_{up}^{new} by *decremental search*. Basically, decremental search is a dual of incremental search. We start from L_{up}^{new} and decrease the makespan until we encounter the unsatisfiable instance.

We try to solve 82 JSSPs in OR-Library [15]. The problems are abz5–abz9, ft06, ft10, ft20, la01–la40, orb01–orb10, swv01–swv20, and yn1–yn4. We limited the execution time of each problem to 2 CPU hours. The single-solution version and SATSHOP succeed to solve 33 problems out of 82 problems. The naive version, normal version, and MiniMaxSAT succeed to solve 32, 31, and 14 problems, respectively. Table 1 shows the experimental results of 33 problems solved.

All experiments were conducted on a Pentium M 753(1.20GHz) machine with 1GB memory running Linux 2.6.16. Each problem is encoded to a CNF (conjunctive normal form)[6]. The second and third columns show statistics of CNFs. The fourth column "$|P|$" shows the size of an atom set P with respect to which we minimize a model. The fifth column "Optimum" shows the minimum makespan. "Single" is the single-solution version. The "Total" row shows the total CPU time for the single-solution version or SATSHOP. The "Ratio" row shows (total time of SATSHOP)/(total time of Single).

The single-solution version usually beats other two versions as expected. On average it solves problems 1.6 times faster than the naive version and 1.3 times faster than the normal version for the 31 problems solved by these three versions.

On the other hand, the SATSHOP beats these three versions on almost all problems. On average it solves problems about 1.7 times faster than the single-solution version. The main reason for the domination of the SATSHOP is that it tries to solve a relaxed problem first. The relaxed one is easy to solve by orders of magnitude. In order to eliminate the effect of the relaxation, we also run the SATSHOP in a non-relaxation mode where it try to solve JSSPs without relaxation. This causes an increase of the runtime of the SATSHOP. The single-solution version, then, is almost comparable with the SATSHOP: the former sometimes beats the latter, and vice versa. On average, the latter solves problems about 1.2 times faster than the former.

The MiniMaxSAT is the worst solver in our experience. It can solve only half of problems solved by others within 2 CPU hours. It seems to be several hundred times slower than others. We may need to develop a SAT encoding tailored for MaxSAT solvers.

[6] We also use the SATSHOP as an encoder. Thus, the core part of a SAT instance solved by the three versions is the same one solved by the SATSHOP.

Table 1. Experimental results of OR-Library

Prob-	No. of	No. of	$\|P\|$	Opti-	runtime in seconds				
lem	Variables	Clauses		mum	Naive	Normal	Single	MaxSAT	SATSHOP
abz5	103,440	1,111,236	1374	1234	55.1	47.1	25.5	time-out	27.5
abz6	81,995	879,864	1075	943	9.2	6.6	6.8	2640.8	6.7
ft06	1,847	11,744	63	55	0.0	0.0	0.0	0.1	0.0
ft10	98,278	1,057,688	1211	930	180.9	263.9	106.5	time-out	63.1
la01	41,288	435,549	921	666	13.6	9.4	6.4	630.1	2.2
la02	36,328	382,368	815	655	50.9	35.7	10.3	1414.1	4.8
la03	37,038	390,635	825	597	21.1	7.0	8.3	1414.6	4.3
la04	37,473	395,009	838	590	5.5	3.3	3.4	1031.9	2.4
la05	28,360	297,253	651	593	7.4	10.2	8.7	1402.5	6.4
la16	94,286	1,014,336	1171	945	32.0	22.7	18.8	5448.9	8.1
la17	71,549	767,296	926	784	12.3	5.8	5.7	1537.5	3.8
la18	73,387	786,292	963	848	14.1	6.7	6.7	1341.9	3.2
la19	92,303	992,462	1162	842	28.4	30.4	14.0	5189.8	10.1
la20	91,828	986,657	1162	902	12.0	6.5	8.7	1473.4	5.9
la22	158,923	2,493,700	1275	927	1815.0	1625.1	1246.2	time-out	878.7
la23	157,152	2,461,589	1279	1032	2239.0	1810.0	1408.4	time-out	827.9
la24	162,299	2,545,886	1318	935	1657.4	1401.4	1280.9	time-out	1238.6
la25	146,928	2,300,907	1196	977	2271.5	2167.6	1343.0	time-out	1180.4
la36	265,072	4,178,965	1546	1268	996.2	770.0	641.5	time-out	210.1
la37	334,107	5,277,206	1868	1397	4975.2	4129.2	3284.7	time-out	1749.7
la38	289,006	4,560,370	1624	1196	time-out	time-out	4797.2	time-out	2511.2
la39	277,638	4,379,775	1584	1233	1738.9	1458.6	892.4	time-out	457.2
la40	279,616	4,411,308	1591	1222	6856.2	time-out	5532.0	time-out	2498.5
orb01	106,638	1,148,842	1303	1059	1822.4	1658.1	1351.08	time-out	1302.4
orb02	107,190	1,154,808	1295	888	25.5	14.5	13.0	2673.5	6.5
orb03	123,706	1,334,614	1461	1005	1530.1	749.0	552.1	time-out	554.2
orb04	113,489	1,223,253	1369	1005	80.6	85.6	63.1	time-out	45.5
orb05	94,014	1,010,346	1152	887	50.1	46.9	31.2	time-out	27.9
orb06	126,502	1,364,933	1500	1010	431.3	294.5	193.4	time-out	105.2
orb07	45,996	492,810	563	397	22.5	17.3	13.6	1414.0	9.8
orb08	101,159	1,089,539	1209	899	98.9	62.2	49.6	time-out	40.0
orb09	96,905	1,043,343	1189	934	106.3	83.9	68.9	time-out	40.2
orb10	125,675	1,356,366	1503	944	49.7	27.9	33.2	time-out	14.9
Total [seconds]					-	-	23025.3	-	13847.4
Ratio					-	-	(1.00)	-	0.60

Turning now to the 49 problems unsolved within 2 CPU hours, even their 48 relaxed problems can not be solved by SATSHOP. Furthermore, some SAT instances are huge [7] for our experimental environment. Ten of the 49 instances require more than 1GB memory, and five of the ten require more than 4GB memory which a 32-bits CPU can not manipulate any more.

6 Conclusions and Future Work

In this paper we presented a characterization of a minimal model with respect to an atom set. Based on this characterization, we gave minimal model generation procedures using a SAT solver as a subroutine. The only function we require from the SAT solver is to compute a model of a satisfiable SAT instance. Thus, our implementation benefits from efficiency of state-of-the-art SAT solvers.

We implemented the naive, normal, and single-solution versions with the SAT solver MiniSat 2. We have performed an experimental evaluation with 82 JSSPs. It shows that the single-solution version usually beats the other two versions. Unfortunately, it rarely beats the SAT-based JSSP solver SATSHOP which performs several optimizations concerning the problem domain. It is for this reason that the SATSHOP generally beats others. In spite of the domination of the SATSHOP, the minimal model generation approach still has an advantage over the SATSHOP in the sense that the former is more general than the latter: the latter solve only JSSP while the former can solve not only JSSP but also several problems such as graph coloring problem, two dimensional strip packing problem, and so on. Stochastic SAT solvers, such as WalkSAT [17], may be useful for increasing performance of the three versions.

We have also applied the Max-SAT solver MiniMaxSAT to the 82 JSSPs. The experimental results show that the MiniMaxSAT is definitely inefficient for solving the JSSP in our SAT encoding though it is a state-of-the-art Max-SAT solver. Implementing a Max-SAT solver based on our approach looks like interesting future work.

Some problems can not be solved because of memory capacity. In order to solve these problems in our framework, we have to purchase a 64-bits CPU and memory, or develop methods to manipulate the problem on the available memory. Encoding the problem into a first order formula seems to be a promising approach to save memory [16].

Answer set programming launched out into the new paradigm of logic programming in 1999, in which a logic program represents the constraints of a problem and its answer sets correspond to the solutions of the problem [11]. Computing answer sets is realized by generating minimal models and checking whether they satisfy some conditions for *negation as failure* [8]. We plan to extend this work to computing answer sets.

[7] SWV13 has the hugest instance in our experiment. It has 2.4 million variables and 121.6 million clauses. And its DIMACS file in gzip format occupies 596 MB.

References

1. L. Bordeaux, Y. Hamadi, and L. Zhang: Propositional Satisfiability and Constraint Programming: A Comparative Survey. *ACM Computing Surveys,* Vol.38, No.4, Article 12 (2006)
2. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu: Symbolic Model Checking without BDDs. In Proc. of TACAS'99, pp.193–207 (1999)
3. F. Bry and A. Yahya: Minimal Model Generation with Positive Unit Hyper-resolution Tableaux. In Proc. of TABLEAUX'96, pp.143–159 (1996)
4. J. M. Crawford, A. B. Baker: Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. In Proc. of AAAI-94, pp.1092–1097 (1994)
5. N. Eén and N. Sörensson: An Extensible SAT-solver. In Proc. of SAT-2003, pp.502–518 (2003)
6. F. Heras, J. Larrosa, and A. Oliveras: MiniMaxSAT: An Efficient Weighted Max-SAT Solver. *J. of Artificial Intelligence Research,* Vol.31, pp.1–32 (2008)
7. H. Kautz and B. Selman: Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In Proc. of AAAI-96, pp.1194–1201 (1996)
8. K. Inoue, M. Koshimura, and R. Hasegawa: Embedding Negation as Failure into a Model Generation Theorem Prover. In Proc. of CADE-11, pp.400–415 (1992)
9. R. Hasegawa, H. Fujita, and M. Koshimura: Efficient Minimal Model Generation Using Branching Lemmas. In Proc. of CADE-17, pp.184–199 (2000)
10. V. Lifschitz: Computing Circumscription. In Proc. of IJCAI-85, pp.121–127 (1985)
11. V. Lifschitz: Answer Set Planning. In Proc. of ICLP-99, pp.23–37 (1999)
12. M. V. Moskewicz, C. F. Madigan, Y. Zhao, and L. Zhang: Chaff: Engineering an Efficient SAT Solver. In Proc. of DAC'01, pp.530–535 (2001)
13. H. Nabeshima, T. Soh, K. Inoue, and K. Iwanuma: Lemma Reusing for SAT based Planning and Scheduling. In Proc. of ICAPS'06, pp.103–112 (2006)
14. I. Niemelä: A Tableau Calculus for Minimal Model Reasoning. In Proc. of TABLEAUX'96, pp.278–294 (1996)
15. OR-Library. `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`
16. J. A. Navarro-Pérez and A. Voronkov: Encodings of Bounded LTL Model Checking in Effectively Propositional Logic. In Proc. of CADE-21, pp.346–361 (2007)
17. B. Selman, H. Kautz, and B. Cohen: Local Search Strategies for Satisfiability Testing. Discrete Mathematics and Theoretical Computer Science, vol. 26, AMS, (1996)