

# Lemma Reusing for SAT based Planning and Scheduling \*

**Hidetomo Nabeshima**

University of Yamanashi  
4-3-11 Takeda, Kofu-shi 400-8511, Japan

**Katsumi Inoue**

National Institute of Informatics  
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

**Takehide Soh**

Kobe University  
1-1 Rokkodai, Nada, Kobe 657-8501, Japan

**Koji Iwanuma**

University of Yamanashi  
4-3-11 Takeda, Kofu-shi 400-8511, Japan

## Abstract

In this paper, we propose a new approach, called *lemma-reusing*, for accelerating SAT based planning and scheduling. Generally, SAT based approaches generate a sequence of SAT problems which become larger and larger. A SAT solver needs to solve the problems until it encounters a satisfiable SAT problem. Many state-of-the-art SAT solvers learn *lemmas* called conflict clauses to prune redundant search space, but lemmas deduced from a certain SAT problem can not apply to solve other SAT problems. However, in certain SAT encodings of planning and scheduling, we prove that lemmas generated from a SAT problem are *reusable* for solving larger SAT problems. We implemented the lemma-reusing planner (LRP) and the lemma-reusing job shop scheduling problem solver (LRS). The experimental results show that LRP and LRS are faster than lemma-no-reusing ones. Our approach makes it possible to use the latest SAT solvers more efficiently for the SAT based planning and scheduling.

## Introduction

In recent years, a *propositional satisfiability* (SAT) problem has been studied actively. The state-of-the-art SAT solvers can solve a SAT problem which consists of millions of clauses in a few minutes. One of the common features of such SAT solvers is learning mechanism of *lemmas* called *conflict clauses* to prune redundant search space. The idea of learning conflict clauses had been introduced by the SAT solver GRASP (Marques-Silva & Sakallah 1999) and efficiently implemented in Chaff (Moskewicz *et al.* 2001) which was the fastest solver in industrial benchmark category of SAT 2004 competition. In SAT 2005 competition, SATELITE (Eén & Biere 2005), MINISAT (Eén & Sörensson 2003a) and HAIFASAT (Gershman 2005) took the first to third places in industrial benchmark category respectively, and also learn conflict clauses.

On the other hand, SAT is the prototypical NP-complete problem, and every instance  $\pi$  of a problem in NP can be

translated into an instance of SAT. The birth of high-speed SAT solvers motivates the translation approach which encodes a certain problem  $\pi$  into a SAT problem and solves it by a fast SAT solver. In this paper, we call such problem solving technique as *SAT encoding approach*.

Many kind of SAT encoding approaches are proposed (Cadoli & Schaerf 2005). In this study, we focus on SAT based planning and scheduling. The SAT planning is the best application of SAT encoding approach. In fact, the SAT based planner SATPLAN04 (Kautz 2004) took the first place for optimal deterministic planning at the International Planning Competition 2004. This shows the potentiality of SAT encoding approach. In this paper, for the SAT planning, we introduce Graphplan-based encoding and action-based encoding which are adopted as default encodings in Blackbox (Kautz & Selman 1998) and SATPLAN04, respectively. Blackbox is the most fundamental SAT based planner. For the SAT scheduling, we present Crawford encoding (Crawford & Baker 1994) which deals with a typical scheduling problem, that is, a *job shop scheduling problem* (JSSP).

Generally, SAT encoding approaches generate a sequence of SAT problems which become larger and larger. A SAT solver needs to solve the problems until it encounters a satisfiable SAT problem. Although generated SAT problems have many common clauses, these are solved independently by the SAT solver. The bottleneck of this approach is satisfiability checking of SAT problems.

For conquering this problem, we focus on lemmas which are learned by various state-of-the-art SAT solvers. A lemma means precept deduced from failure in search, and is used to avoid making the same mistake in the future search. Generally, lemmas deduced from a certain SAT problem are not applicable for solving other SAT problems. However, in Graphplan-based, action-based and Crawford encoding, we prove that lemmas generated from a SAT problem are *reusable* for solving larger SAT problems.

*Bounded model checking* (BMC) for verification and optimization in electronic design automation is also one of the SAT encoding approaches. Lemma-reusing for BMC has already been proposed (Whittemore, Kim, & Sakallah 2001; Shtrichman 2001; Eén & Sörensson 2003b; Jin & Somenzi 2005). Our lemma-reusing approach is basically the same as one proposed by Eén and Sörensson, but it has been de-

\*This research is supported partly by Grant-in-Aid from The Ministry of Education, Science and Culture of Japan (No.16700136).

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

veloped independently for the SAT planning (Nabeshima, Nozawa, & Iwanuma 2005). In this paper, we clarify the condition of lemma-reusability, and give the formal correctness of lemma-reusing which is not described in (Eén & Sörensson 2003b).

We implemented the lemma-reusing planner (LRP) based on Blackbox and the lemma-reusing JSSP solver (LRS) based on Crawford encoding. LRP can encode a SAT problem by Graphplan-based encoding or action-based encoding. The only difference between LRP (LRS) and Blackbox (Crawford encoding) is whether lemmas are reused or not. The experimental results show that LRP and LRS are faster than lemma-no-reusing ones.

Our lemma-reusing approach only extracts lemmas from a general-purpose SAT solver and then simply adds it to the next SAT problem. The community working on SAT is very active, and more effective and faster general-purpose SAT solvers will be developed year by year. Our approach makes it possible to use the latest SAT solvers more efficiently for the SAT based planning and scheduling. In this paper, we show Chaff and MINISAT are applicable to lemma-reusing for the SAT planning and scheduling.

The remaining part of this paper organized as follows: Section 2 introduces Chaff and its learning mechanism of conflict clauses. In Section 3, we define *lemma-reusability condition* and prove the correctness of it. Section 4 and 5 describe SAT planning and scheduling, respectively. Section 6 shows the experimental results. Section 7 and 8 are related works and the conclusion of this work, respectively.

## Chaff

The idea of learning conflict clauses had been introduced by (Marques-Silva & Sakallah 1999) and efficiently implemented in Chaff (Moskewicz *et al.* 2001). In this section, we describe the learning mechanism of Chaff.

We represent a SAT problem by conjunctive normal form (CNF). This form consists of the logical AND of one or more *clauses*, which consist of the logical OR of one or more *literals*. The *literal* is a propositional variable  $v$  (denoted as  $+v$ ) or its complement (denoted as  $-v$ ). Specially, a clause which consists of only one literal is called a *unit clause*. We represent a clause as a set of literals. The answer of a SAT problem is the truth assignment satisfying all clauses in the problem.

## Davis-Putnam Backtrack Search

Chaff solves a SAT problem by Davis-Putnam (DP) backtrack search. Figure 1 shows the pseudo code of DP backtrack search (Moskewicz *et al.* 2001). We explain Figure 1 briefly. Initially, values of all variables are unassigned. The operation of `bcp()`, which carries out *boolean constraint propagation*, is as following: (i) finds a clause that consists of one unassigned literal  $L$  and all other assigned literals whose values are false, and then (ii) assigns true to  $L$  for satisfying the clause. This truth value assignment by `bcp()` is called a *implication*. Clauses in this state are said to be *unit*. Strictly, these clauses are not unit clauses structurally, but it can be considered as unit clauses semantically. `bcp()` repeats implications transitively until either there are no more

---

```

while (true) {
  while (!bcp())
    if (!resolveConflict())
      return(not satisfiable);
  if (!decide()) // if no unassigned vars
    return(satisfiable);
}

bool resolveConflict() {
  d = most recent decision not 'tried both ways';
  if(d == NULL) return false; // no such d was found
  flip the value of d;
  mark d as tried both ways;
  undo any invalidated implications;
  return true;
}

```

---

Figure 1: Davis-Putnam backtrack search

implications (in which case it returns true) or a *conflict* is produced (in which case it returns false). The operation of `decide()` is to select a variable that is not currently assigned, and give it a value. This truth value assignment is called a *decision*, and the assigned variable is called a *decision variable*. When a new decision is made, a record of the decision is pushed onto the *decision stack*. We define the height of the decision stack as *decision level* (DL). `decide()` will return false if no unassigned variables remain and true otherwise. A *conflict* occurs when implications for setting the same variable to both 1 and 0 are produced. If a conflict occurs, `resolveConflict()` cancels all implications caused by the most recent decision, that is, invalidates assignments of all variables assigned by the implications, and flips the value of the decision assignment. If both values have already been tried for this decision, `resolveConflict()` finds a decision that has not been tried both ways, and proceeds from there in the manner described above.

## Conflict Clause

When a conflict is produced, Chaff generates a lemma, called a *conflict clause*. The purpose of lemma generation is to prune redundant search space. We explain the lemma generation algorithm in Chaff using an example.

Figure 2 shows the *implication graph* (Marques-Silva & Sakallah 1999) generated by Chaff for solving the SAT problem which consists of 8 clauses.  $X@Y$  in the figure shows that the literal  $X$  is assigned as true at DL  $Y$ . At first, Chaff assigns true to the literal  $-5$  because it is contained in the unit clause No.8 ( $-5@0$ ). As a result,  $+5$  in the first clause becomes false, and hence  $-6$  is assigned as true for satisfying this clause ( $-6@0$ ). Now there are no more unsatisfied unit clauses, `decide()` selects the literal  $+9$  by VSIDS decision heuristic (Zhang *et al.* 2001), and assigns true to  $+9$  ( $+9@1$ ). By this decision, the clauses No.3 and No.4 become unit clauses, and then the literals  $-7$  and  $+8$  are assigned as true ( $-7@1, +8@1$ ). Since  $+6$  and  $+7$  are assigned as false, for satisfying the second clause, Chaff tries to assign true to  $-8$ . However, a conflict is produced because  $+8$  is already assigned as true.

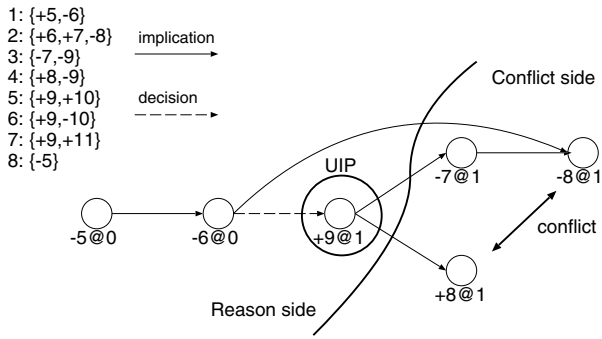


Figure 2: Implication graph

When a conflict is produced, Chaff selects a special node, called a *unique implication point* (UIP) (Marques-Silva & Sakallah 1999), from the conflicting implication graph. A node  $x$  at DL  $l$  is a *UIP* iff any path from a decision variable at DL  $l$  to the conflicting variable needs to go through  $x$ <sup>1</sup>. In this example,  $+9$  is a UIP. The set of nodes whose values are assigned after the UIP is called *conflict side*, and the set of the remaining nodes (including the UIP) is called *reason side*. The conflict clause consists of the complements of all nodes directly connected to the reason side. In this example,  $-6$  and  $+9$  are directly connected to the reason side. Hence, the conflict clause  $\{+6, -9\}$  is generated, and is added to the original SAT problem. The generated conflict clause is used to avoid making the same conflict in the future search. For example, if  $-6$  is assigned as true, then immediately  $\text{bcp}()$  assigns true to  $-9$  by that conflict clause. Therefore, the same conflict is avoided.

Note that if a conflict occurs at DL 0, there is a unit clause which is the trigger of the conflict. In such case, one can immediately conclude that the problem is unsatisfiable and stop the solver. However, Chaff generates a conflict clause even if a conflict occurs at DL 0. Since this conflict clause is not used and is not necessary, in the following discussion, we assume that a conflict clause is not generated at DL 0.

### Lemma Reusing

Generally, SAT encoding approaches generate a sequence of SAT problems  $\langle P_1, P_2, \dots \rangle$  which become larger and larger. These SAT problems are similar with each other, that is,  $P_{i+1}$  includes  $P_i$  except for some clauses. This kind of problem is called an *incremental SAT problem*.

For efficiently solving an incremental SAT problem, we define the lemma-reusability condition.

**Definition 1** Suppose that  $P$  and  $Q$  are SAT problems. The *lemma-reusability condition between  $P$  and  $Q$*  is as follows:

If  $P$  includes a non-unit clause  $x$ , then  $Q$  contains  $x$ .

This condition is basically the same as the formalism proposed by (Eén & Sörensson 2003b), but we give the formal correctness of the reusability condition.

<sup>1</sup>Note that there may be one or more UIPs for a certain conflict. In such a case, we choose the nearest UIP from the conflict.

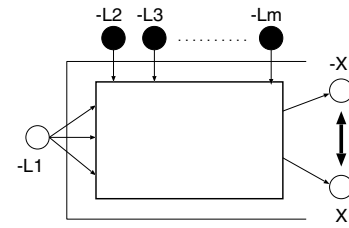


Figure 3: Abstracted implication graph of  $c$

**Theorem 1** If  $P$  is a SAT problem and  $c$  is any conflict clause generated by solving  $P$ , then  $c$  is a logical consequence of a set of some non-unit clauses in  $P$ .

**Proof:** Suppose that  $c = L_1 \vee L_2 \vee \dots \vee L_m$  where each  $L_l$  is a literal. The conflict clause  $c$  was generated by solving  $P$ . We show an abstracted implication graph when  $c$  was generated at Figure 3. UIP is  $-L_1$ . Black circles represent literals assigned before  $-L_1$ , and the white box contains nodes assigned after  $-L_1$ .  $X$  and  $-X$  are contradicting literals.

We represent clauses which were used for constructing the implication graph as follows:

$$\begin{aligned} D_1 &= \{s_{11}, \dots, s_{1s_1}, t_{11}, \dots, t_{1t_1}, u_1\}, \\ D_2 &= \{s_{21}, \dots, s_{2s_2}, t_{21}, \dots, t_{2t_2}, u_2\}, \\ &\vdots \\ D_v &= \{s_{v1}, \dots, s_{vs_v}, t_{v1}, \dots, t_{vt_v}, u_v\}, \end{aligned} \quad (1)$$

where  $s_j > 0$  and  $t_j \geq 0$  ( $1 \leq j \leq v$ ). Suppose that  $S(D_j) = \{s_{j1}, \dots, s_{js_j}\}$ ,  $T(D_j) = \{t_{j1}, \dots, t_{jt_j}\}$  and  $U(D_j) = u_j$ . Then, each clause  $D_j$  ( $1 \leq j \leq v$ ) satisfies the following two conditions:

$$\forall L \in S(D_j) \quad (-L \text{ is UIP} \vee (\exists k < j (U(D_k) = -L))) \quad (2)$$

$$T(D_j) \subseteq \{L_2, \dots, L_m\} \quad (3)$$

Additionally, there are  $D_i$  and  $D_j$  ( $1 \leq i \neq j \leq v$ ) which satisfy

$$\exists i, j (U(D_i) = -U(D_j)). \quad (4)$$

We explain meanings of (2), (3) and (4) intuitively.  $D_j$  consists of  $S(D_j)$ ,  $T(D_j)$ , and  $U(D_j)$ .  $T(D_j)$  indicates a set of literals already assigned as false before  $-L_1$ , and each literal in  $S(D_j)$  is assigned false by implications after  $-L_1$  or by the complement of UIP. Therefore, in order to satisfy  $D_j$ ,  $U(D_j)$  must be assigned as true, and this serves as a trigger of new implications. (4) shows that the chain of implications generated inconsistency.

Each clause  $D_k$  ( $1 \leq k \leq v$ ) is an element of  $P$  and the length of  $D_k$  is two or more by its definition (1). We show that any model of  $D_1 \wedge \dots \wedge D_v$  satisfies  $c$ :

$$D_1 \wedge \dots \wedge D_v \models c \quad (5)$$

We prove (5) by *reductio ad absurdum*. When we assume

$$D_1 \wedge \dots \wedge D_v \not\models c \quad (6)$$

and lead a contradiction. (6) says that any model of  $D_1 \wedge \dots \wedge D_v$  satisfies  $\neg L_1 \wedge \dots \wedge \neg L_m$ . If  $\neg L_1 \wedge \dots \wedge \neg L_m$  is true, then it is obvious that the conflict  $X \wedge \neg X$  is implied by implications in Figure 3. Hence, the assumption is contradictory, and then (5) is proved. This means that  $c$  is a logical consequence of a set of the non-unit clauses  $\{D_1, \dots, D_v\}$  ( $\subseteq P$ ).  $\square$

Theorem 1 had been described in our previous work for lemma-reusing planning (Nabeshima, Nozawa, & Iwanuma 2005), but in this paper, we redefined Theorem 1 and reconstructed the proof more clearly.

If SAT problems  $P$  and  $Q$  generated by a certain SAT encoding approach satisfy the reusability condition, Theorem 1 guarantees that all the clauses concerned with generation of any conflict clause in  $P$  are contained in  $Q$ . Therefore, we can reuse conflict clauses for pruning redundant search space in  $Q$  regardless of the satisfiability of  $P$  and  $Q$ .

### SAT Planning

In this section, we introduce two kind of encodings, *Graphplan-based encoding* and *action-based encoding*, for SAT planning. Both encodings are proposed by (Kautz, McAllester, & Selman 1996), and adopted as default encodings in Blackbox (Kautz & Selman 1998) and SATPLAN04 (Kautz 2004), respectively. Note that Blackbox can use the action-based encoding by specifying at a command line argument. SATPLAN04 took the first place for optimal deterministic planning at the International Planning Competition 2004. The single most important difference between Blackbox and SATPLAN04 is the SAT solvers (Kautz 2004). Blackbox uses Chaff and SATPLAN04 uses Siege<sup>2</sup> as a default SAT solver. We describe the algorithm of Blackbox as an example of SAT planning.

Blackbox accepts a planning problem specified in STRIPS notation, and outputs a shortest partial ordered plan. Given a planning problem, Blackbox performs the following: (i) assumes that a plan length is  $i$  (initially,  $i = 1$ ), (ii) converts the planning problem into a *planning graph* (Blum & Furst 1997), (iii) translates the planning graph into a SAT problem  $P_i$ , and (iv) solves  $P_i$  by Chaff. If  $P_i$  is found to be satisfiable, the planner extracts actions which are assigned as true, sorts the actions according to their levels, removes redundant actions, and then outputs the plan of length  $i$ . When  $P_i$  is unsatisfiable, the planner increases the plan length to be  $i + 1$ , and returns to (i). We define the notation  $P_i$  as the SAT problem which is generated under the assumption that the plan length is  $i$ .

A *planning graph* is a directed leveled graph (Figure 4). Nodes in even numbered levels correspond to propositions, and nodes in odd numbered levels correspond to actions. An edge connects a proposition to an action at the next level whose preconditions contain the proposition. Similarly, an action is connected to a proposition which is contained in the effect of the action. In Figure 4, a horizontal line between proposition levels indicates a *no-operation action*  $nop(f)$  which propagates the proposition  $f$  at level  $k$  to level  $k + 1$ .

<sup>2</sup>Siege was developed by Lawrence Ryan at Simon Fraser University. See <http://www.cs.sfu.ca/~loryan/personal/>

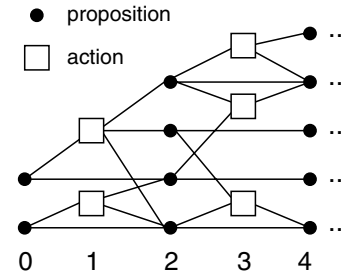


Figure 4: Planning graph

In a planning graph, the *mutual exclusion relation* (*mutex*) between two actions (propositions) at level  $i$  is defined as follows:

- *Two actions at level  $i$  are mutex* if either
  1. the effect of one action denies another action's effect,
  2. the effect of one action denies another action's precondition, or
  3. the actions have preconditions that are mutually exclusive at level  $i - 1$ .
- *Two propositions at level  $i$  are mutex* if
  1. one is the negation of the other, or
  2. all ways of achieving the propositions (that is, actions at level  $i - 1$ ) are pairwise mutex.

The fact that any two actions (propositions) at a certain level  $i$  are mutex implies that the two actions (propositions) are not simultaneously satisfied at level  $i$ .

In Graphplan-based encoding, a planning graph is translated into the SAT problem according to the following rules:

1. The initial condition of a planning problem holds at level 0. Let  $f_1 \wedge \dots \wedge f_s$  be the initial condition, the following unit clauses are added:

$$f_1^0 \wedge \dots \wedge f_s^0$$

2. The goal condition holds at the maximum level  $m$ . Let  $g_1 \wedge \dots \wedge g_t$  be the goal condition, the following unit clauses are added:

$$g_1^m \wedge \dots \wedge g_t^m$$

3. Two actions  $a, b$  which are mutex at level  $i$  are not satisfied simultaneously.

$$\neg a^i \vee \neg b^i$$

4. An action implies its precondition. That is, for every action  $a$  at level  $i$  and the precondition  $p_1, \dots, p_k$ , the following formula is added:

$$a^i \supset p_1^{i-1} \wedge \dots \wedge p_k^{i-1}.$$

5. A proposition at level  $i$  implies the disjunction of the actions at level  $i - 1$  which have it as an effect. That is, for every proposition  $f$  at level  $i$  and the actions  $a_1, \dots, a_k$  which are connected to  $f$ , the following formula is added:

$$f^i \supset a_1^{i-1} \vee \dots \vee a_k^{i-1}$$

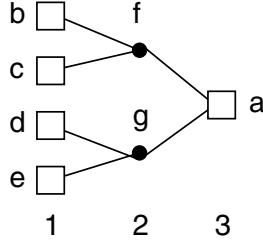


Figure 5: A part of planning graph

Let  $P_i$  and  $P_{i+1}$  be SAT problems generated by Graphplan-based encoding. Since the nodes of a planning graph increase monotonically, all the clauses in  $P_i$  except for the unit clauses added by the rule 2 are included in  $P_{i+1}$ . Hence, the following Lemma holds.

**Lemma 1** Graphplan-based encoding satisfies the lemma-reusability condition.

The action-based encoding uses rule 2, 3 and the following rule 6, and uses 4 for only propositions at the maximum level. Note that only the actions whose preconditions are satisfied by the initial condition are added to level 1. Hence the rule 1 is unnecessary because an action at level 1 can be performed.

6. An action at level  $i$  implies each of the disjunctions of the actions at level  $i - 2$  that add each of its preconditions. For example, the action  $a$  in Figure 5 is translated into the following formula:

$$a^3 \supset (b^1 \vee c^1) \wedge (d^1 \vee e^1)$$

Let  $P_i$  and  $P_{i+1}$  be SAT problems generated by the action-based encoding. The following Lemma holds as well as Lemma 1. This shows that we can reuse lemmas generated from  $P_i$  for solving  $P_{i+1}$ .

**Lemma 2** The action-based encoding satisfies the lemma-reusability condition.

## SAT Scheduling

In this section, we introduce a *job shop scheduling problem* (JSSP) which is a typical scheduling problem, and show the SAT encoding approach proposed by Crawford and Baker (Crawford & Baker 1994). We call this encoding as *Crawford encoding*.

A JSSP consists of a set of  $n$  jobs  $\{J_1, \dots, J_n\}$  and a set of  $m$  machines  $\{M_1, \dots, M_m\}$ . Each job  $J_i$  is a sequence of operations  $\langle O_1^i, \dots, O_{q_i}^i \rangle$ . Each operation  $O_i^l$  requires the exclusive use of a machine  $M_{O_i^l}$  ( $1 \leq O_i^l \leq m$ ) for an uninterrupted duration  $p_i^l$ , its processing time. A *schedule* is a set of start times for each operation  $O_i^l$ . The time required to complete all the jobs is called the *makespan*  $L$ . The objective of the JSSP is to determine the schedule which minimizes  $L$ .

We introduce three kinds of boolean variables:

- $pr_{i,j}^{l,k}$  means that  $O_i^l$  precedes  $O_j^k$ .
- $sa_{i,t}^l$  means that  $O_i^l$  starts at time  $t$  or later.

- $eb_{i,t}^l$  means that  $O_i^l$  ends by time  $t$ .

We translate the JSSP into a SAT problem by the following rules<sup>3</sup>:

1.  $O_i^l$  precedes  $O_{i+1}^l$ :

$$pr_{i,i+1}^{l,l} \quad (1 \leq l \leq n, 1 \leq i < q_l)$$

2. If  $O_i^l$  and  $O_j^k$  require the same machine, then  $O_i^l$  precedes  $O_j^k$  or  $O_j^k$  precedes  $O_i^l$ . That is, if  $O_i^l = O_j^k$ , then we add the clause:

$$pr_{i,j}^{l,k} \vee pr_{j,i}^{k,l} \quad (1 \leq l < k \leq n) \\ (1 \leq i \leq q_l, 1 \leq j \leq q_k)$$

3.  $O_i^l$  starts at time  $t$  or later:

$$sa_{i,t}^l \quad (1 \leq l \leq n, 1 \leq i \leq q_l, t = \sum_{u=1}^{i-1} p_u^l)$$

4.  $O_i^l$  ends by time  $t$ :

$$eb_{i,t}^l \quad (1 \leq l \leq n, 1 \leq i \leq q_l, t = L - \sum_{u=i+1}^{q_l} p_u^l)$$

5. If  $O_i^l$  starts at or after time  $t$ , then it starts at or after time  $t - 1$ :

$$sa_{i,t}^l \rightarrow sa_{i,t-1}^l \quad (1 \leq l \leq n, 1 \leq i \leq q_l, 1 \leq t \leq L)$$

6. If  $O_i^l$  ends by  $t$ , then it ends by  $t + 1$ :

$$eb_{i,t}^l \rightarrow eb_{i,t+1}^l \quad (1 \leq l \leq n, 1 \leq i \leq q_l, 0 \leq t < L)$$

7. If  $O_i^l$  starts at or after time  $t$ , then it cannot end before time  $t + p_i^l - 1$ :

$$sa_{i,t}^l \rightarrow \neg eb_{i,t+p_i^l-1}^l \quad (1 \leq l \leq n, 1 \leq i \leq q_l) \\ (0 \leq t < L - p_i^l + 1)$$

8. If  $O_i^l$  starts at or after  $t$  and  $O_j^k$  follows  $O_i^l$ , then  $O_j^k$  can not start until  $O_i^l$  is finished. That is, for  $pr_{i,j}^{l,k}$  added by rule 1 and 2, we add the clause:

$$sa_{i,t}^l \wedge pr_{i,j}^{l,k} \rightarrow sa_{j,t+p_i^l}^k \quad (0 \leq t \leq L - p_i^l)$$

Let  $P$  be the SAT problem translated by the above Crawford encoding. If  $P$  is satisfiable, then the JSSP can complete all the jobs by the makespan  $L$ . Let  $S$  be the truth assignment satisfying  $P$ . The start time  $t_i^l$  of each operation  $O_i^l$  is given by extracting the last  $sa_{i,t}^l$  which is assigned as true in  $S$ . More precisely,  $t_i^l$  is given by  $t$  which satisfies the following expression:

$$(sa_{i,t}^l \in S) \wedge (\neg \exists u > t (sa_{i,u}^l \in S))$$

For minimizing the makespan, we apply two kinds of methods, *incremental search* and *binary search*. One can

<sup>3</sup>We redefine Crawford encoding somewhat precisely.

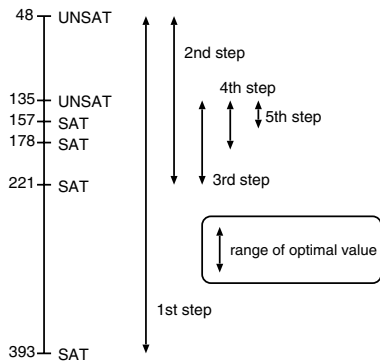


Figure 6: Binary search for JSSP

easily estimate the upper bound  $L_{up}$  and lower bound  $L_{low}$  of the minimum makespan:

$$L_{up} = \sum_{l=1}^n \sum_{i=1}^{q_l} p_i^l, \quad L_{low} = \max_{l=1}^n \left( \sum_{i=1}^{q_l} p_i^l \right)$$

Obviously, the JSSP can complete all the jobs by  $L_{up}$  and can not complete by  $L_{low} - 1$ . Let  $P_x$  be a SAT problem generated by Crawford encoding under the assumption that the makespan is  $x$ . In the incremental search, we start from  $L_{low}$  and increase the makespan by 1 until we encounter the satisfiable problem  $P_t$ . If  $P_t$  is found, then the minimum makespan is  $t$ . Figure 6 shows the process of the binary search. For example, let  $L_{up} = 393$  and  $L_{low} = 49$ . Firstly, we try to solve  $P_{221}$  because 221 is the midpoint between 48 and 393. If  $P_{221}$  is satisfiable, then try  $P_{135}$ . If  $P_{135}$  is unsatisfiable, then try  $P_{178}$ . We continue this binary search until we encounter the satisfiable problem  $P_t$  and unsatisfiable problem  $P_{t-1}$ . Then, the minimum makespan is  $t$ .

Let  $P_i$  and  $P_j$  be SAT problems generated by Crawford encoding ( $i < j$ ). The following lemma is valid since all clauses in  $P_i$  except for the unit clauses added by rule 4 are contained in  $P_j$ . This shows that we can reuse lemmas generated from  $P_i$  for solving  $P_j$ .

**Lemma 3** The Crawford encoding satisfies the lemma-reusability condition.

## Experimental Results

### Lemma-Reusing Planner

We implemented the lemma-reusing planner (LRP) based on Blackbox ver.4.3. LRP was originally developed in (Nabeshima, Nozawa, & Iwanuma 2005). In this study, we modified LRP and Blackbox, in order to use not only Chaff but also MINISAT as a SAT solver. MINISAT has learning mechanism of conflict clauses like Chaff and took the second place in industrial benchmark category of SAT 2005 competition. LRP can use not only Graphplan-based encoding but also the action-based encoding as well as Blackbox. The only difference between LRP and Blackbox is whether lemmas are reused or not. We compared LRP with Blackbox and SATPLAN04 (Kautz 2004)

for benchmark problems used in the International Planning Competition (IPC) 2004. SATPLAN04 was the fastest planner in optimal deterministic planning at the IPC 2004.

Chaff and MINISAT generate a large numbers of conflict clauses for solving a hard SAT problem. Sometimes the number of conflict clauses exceeds hundreds of thousands. Reusing a lot of lemmas often causes the serious fall in speed. Hence, we also implemented the special version of LRP, called LRP10 which reuses only the conflict clauses whose length are 10 or less.

Table 1 shows the experimental results of IPC 2004 benchmark problems. All experiments were conducted on a Xeon 2.8GHz machine with 1GB memory on Linux. We executed 13 kinds of planners, SATPLAN04 with Siege and 12 combinations of three planners (Blackbox, LRP and LRP10), two SAT solvers (Chaff and MINISAT), and two encodings (Graphplan-based and the action-based encoding).

IPC 2004 benchmark problems for optimal deterministic planning consists of 279 problems. We limited the execution time of each planner to 600 CPU seconds. We omit the result of 95 problems which all the planners solved within less than 10 CPU seconds and omit 116 problems which any planner could not solve within 600 CPU seconds or by cause of memory overflow. 21 problems were solved by only SATPLAN04, while all the other solvers (including Blackbox) could not solve the problems because of memory overflow. Therefore, this does not indicate the weak point of the lemma-reusing approach. Consequently, we omitted the 21 problems due to limitations of space. As a result, Table 1 contains the results of 47 problems. ‘‘Len’’ is length of the shortest plan. ‘‘SP04’’ and ‘‘BBox’’ are SATPLAN04 and Blackbox, respectively. ‘‘AIRPORT/NONTEMPORAL/STRIPS’’ represent the directory in IPC 2004 benchmark problems, and ‘‘P09, P16, P17, . . .’’ are planning problems contained in the directory. ‘‘t.o.’’ means that the problem could not be solved within 600 CPU seconds. ‘‘Total’’ is the total of CPU time of a planner. For calculating appropriate total time, we regarded ‘‘t.o.’’ as 600 seconds and added it to the total time. ‘‘Speedup’’ is (total time of Blackbox) / (total time of LRP (LRP10)).

Table 1 shows that LRS10 has stable improvement of efficiency. On the other hand, LRP has almost no good effect. This shows that naive reuse of lemmas has harmful effect. It is interesting that while Chaff is good at Graphplan-based encoding with lemma-reusing, MINISAT is good at the action-based encoding with lemma-reusing. Especially LRP10 with MINISAT using the action-based encoding is the fastest, and has achieved two times improvement by reusing lemmas. Although SATPLAN04 is the new implementation of Blackbox, it is slower than Blackbox with Chaff in our experiment. The same tendency is shown in the experimental results in (Vidal & Geffner 2006). However, as mentioned above, SATPLAN04 is robust than Blackbox because 21 problems were solved by only SATPLAN04.

### Lemma-Reusing JSSP Solver

We implemented the lemma-reusing JSSP solver (LRS) and the normal JSSP solver (denoted as CRW) based on Craw-

Table 1: Experimental results of IPC04 benchmark problems

Prob	Len	Siege	Chaff						MINISAT					
		SP04 [sec]	Graphplan-based			Action-based			Graphplan-based			Action-based		
			BBox [sec]	LRP [sec]	LRP10 [sec]	BBox [sec]	LRP [sec]	LRP10 [sec]	BBox [sec]	LRP [sec]	LRP10 [sec]	BBox [sec]	LRP [sec]	LRP10 [sec]
AIRPORT/NONTEMPORAL/STRIPS														
P09	27	3.7	10.8	11.0	10.9	2.3	2.3	2.3	9.8	9.8	9.8	1.3	1.3	1.3
P16	27	5.5	13.8	15.4	13.8	3.5	2.8	2.8	12.5	12.5	12.5	1.6	1.6	1.6
P17	28	8.4	31.8	32.4	32.3	5.4	5.2	5.7	28.6	29.0	28.5	2.5	2.5	2.6
P18	31	31.8	107.5	102.8	102.9	16.4	15.3	14.3	92.6	92.1	91.4	5.7	6.3	5.6
P19	30	9.0	57.4	56.4	56.2	8.4	7.2	8.3	50.4	50.5	50.7	3.3	3.4	3.5
P20	32	50.8	181.6	175.8	181.4	25.7	24.6	25.2	159.8	163.0	162.6	10.2	8.7	9.6
PIPESWORLD/NOTANKAGE.NONTEMPORAL/STRIPS														
P04	11	131.3	3.3	5.2	2.5	7.7	12.1	3.7	3.0	3.1	3.7	2.9	1.5	1.7
P09	9	103.5	10.5	19.3	7.6	20.6	17.1	7.8	4.1	4.1	4.1	2.4	2.7	2.9
P10	10	261.4	79.3	178.5	98.6	283.1	195.9	352.8	26.1	18.3	12.2	43.1	56.4	7.8
P13	12	59.1	16.1	31.3	19.0	33.2	35.2	28.5	7.8	6.4	6.3	9.6	3.4	6.2
P14	16	228.8	401.2	t.o.	290.7	t.o.	t.o.	t.o.	t.o.	539.7	202.0	t.o.	t.o.	336.2
P17	12	231.1	109.5	155.6	132.8	187.8	184.8	152.0	45.8	62.3	56.4	78.2	93.2	55.4
P21	14	166.2	97.9	119.8	60.2	178.9	122.6	106.0	48.9	37.0	28.4	368.4	108.1	44.0
P23	13	38.7	36.5	37.1	36.0	51.1	32.9	44.3	19.3	22.7	20.7	16.8	13.6	7.7
P24	14	204.5	184.5	162.0	140.5	215.5	147.9	242.9	169.5	116.0	99.4	170.9	334.8	73.8
PIPESWORLD/TANKAGE.NONTEMPORAL/STRIPS														
P02	11	264.7	2.2	2.8	2.2	2.4	2.9	1.7	1.2	1.2	1.2	0.6	0.6	0.6
P04	11	459.7	59.9	56.7	33.9	244.6	329.7	426.8	58.2	161.2	182.5	360.1	t.o.	283.0
PROMELA/OPTICAL TELEGRAPH/STRIPS														
P06	13	12.7	6.0	6.0	6.0	2.5	2.5	2.5	4.9	5.0	5.0	1.4	1.4	1.4
P07	13	17.4	8.2	8.2	8.3	3.1	3.3	3.2	6.9	6.9	6.8	1.8	1.8	1.8
P08	13	22.2	11.1	11.0	11.1	3.9	3.9	4.0	9.3	9.3	9.3	2.2	2.3	2.3
P09	13	27.7	14.5	12.7	14.4	4.8	4.8	4.8	12.3	12.4	12.3	2.9	2.8	2.9
P10	13	34.2	17.9	18.3	18.2	5.9	5.9	5.9	15.7	16.0	15.8	3.5	3.5	3.5
P11	13	41.6	22.9	23.1	22.8	7.1	7.0	7.0	19.7	20.2	19.4	4.2	4.3	4.2
P12	13	51.0	28.1	28.2	28.1	8.4	8.4	8.5	24.9	24.9	24.6	5.1	5.1	5.1
P13	13	58.7	34.3	34.3	34.3	9.7	9.9	9.8	30.2	29.6	30.3	6.0	6.1	6.2
PROMELA/PHILOSOPHERS/STRIPS														
P25	11	6.3	9.0	9.9	10.0	3.7	3.6	3.6	8.4	8.1	8.3	1.9	1.8	1.9
P26	11	7.1	10.9	10.9	10.9	3.9	3.9	3.9	9.2	9.0	9.2	2.0	1.9	2.0
P27	11	7.6	11.8	12.0	11.9	4.2	4.2	4.2	10.1	9.9	10.1	2.2	2.1	2.1
P28	11	8.1	12.9	13.0	13.0	4.5	4.5	4.5	11.1	10.9	11.1	2.3	2.3	2.3
P29	11	8.7	14.2	14.1	14.2	4.9	4.8	4.9	12.2	11.7	12.1	2.5	2.4	2.5
PSR/SMALL/STRIPS														
P22	25	136.9	540.9	182.2	67.4	336.6	358.3	288.0	52.8	48.7	11.8	121.7	132.4	29.1
P25	9	155.6	3.6	3.7	3.6	3.6	3.6	3.7	3.6	3.6	3.7	3.7	3.4	3.7
P29	18	24.1	13.4	19.4	9.6	21.7	26.9	13.7	4.7	4.5	4.2	3.8	8.3	4.4
P31	16	13.3	53.1	35.8	23.9	63.9	49.0	30.9	10.8	9.3	8.6	10.5	11.3	10.6
P36	16	5.0	17.0	16.3	14.6	16.2	14.8	12.0	6.7	6.7	6.5	3.4	5.7	4.2
P40	15	14.3	43.9	42.5	41.6	49.1	59.8	37.6	17.8	15.2	14.1	28.5	48.2	33.0
P47	23	31.3	51.5	49.6	19.3	56.7	94.6	33.6	22.1	14.3	7.6	34.0	72.8	16.1
P48	26	267.3	t.o.	t.o.	157.0	t.o.	t.o.	493.0	117.5	67.3	39.5	261.9	t.o.	82.9
SATELLITE/STRIPS/STRIPS														
P01	8	11.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.0	0.0	0.0
P02	12	281.6	1.2	1.0	1.0	1.2	1.4	0.9	0.3	0.3	0.3	0.2	0.2	0.2
P04	10	139.7	2.1	2.3	2.1	1.6	1.8	1.6	0.8	0.8	0.8	0.4	0.4	0.4
P05	7	56.7	1.6	2.1	1.8	1.4	2.2	1.5	0.9	0.9	0.9	0.6	0.5	0.5
P06	8	28.3	1.9	2.0	2.0	1.6	1.6	1.6	0.9	0.9	1.0	0.4	0.4	0.4
P07	6	26.9	2.1	2.3	2.2	1.8	1.8	1.8	1.0	1.0	1.1	0.5	0.6	0.6
P08	8	305.8	394.8	180.6	44.0	256.6	433.9	41.4	195.5	135.9	55.6	23.8	27.2	16.9
P09	6	39.7	6.9	7.2	6.9	5.0	5.1	5.1	3.6	3.6	3.6	1.6	1.6	1.6
P10	8	196.1	126.0	219.7	43.1	39.9	48.7	26.8	40.6	51.9	47.0	17.5	34.6	26.1
Total [sec]		4295	3466	3330	1864	3410	3509	3085	1992	1868	1353	2228	2823	1112
Speedup				1.04	1.86		0.97	1.11		1.07	1.47		0.79	2.00

Table 2: Experimental results of OR-library

Prob	Opt	Chaff						MINISAT					
		Incremental search			Binary search			Incremental search			Binary search		
		CRW [sec]	LRS [sec]	LRS10 [sec]	CRW [sec]	LRS [sec]	LRS10 [sec]	CRW [sec]	LRS [sec]	LRS10 [sec]	CRW [sec]	LRS [sec]	LRS10 [sec]
abz5	1234	t.o.	2888	t.o.	958	641	784	1699	1626	1370	153	126	154
abz6	943	1127	719	1028	295	251	281	487	493	474	54	56	54
ft06	55	0	0	0	0	0	0	0	0	0	0	0	0
ft10	930	t.o.	3474	t.o.	1560	1000	1820	1270	1109	989	210	186	162
la01	666	639	271	411	80	42	71	195	212	178	17	18	17
la02	655	1555	484	720	317	92	119	277	320	252	33	29	32
la03	597	496	229	308	89	63	78	166	203	156	16	16	15
la04	590	335	192	277	60	39	59	131	124	124	14	15	14
la05	593	823	395	462	147	88	101	313	400	229	48	44	44
la16	945	1798	817	1331	304	226	336	602	585	560	55	58	55
la17	784	742	369	554	158	138	184	288	290	273	36	36	37
la18	848	854	451	657	223	173	209	365	351	364	40	40	41
la19	842	1347	725	1235	289	241	302	485	477	466	59	63	61
la20	902	761	448	779	236	190	242	327	330	328	46	47	46
la22	927	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	1915	1190	1466
la23	1032	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	2431	2380	2279
la24	935	t.o.	t.o.	t.o.	m.o.	m.o.	m.o.	t.o.	t.o.	t.o.	1591	1124	1224
la25	977	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	1447	1572	1802
la36	1268	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	1373	958	1156
la39	1233	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	908	932	930
orb01	1059	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	3595	2278	2482
orb02	888	1427	831	1437	217	207	280	597	607	568	46	50	49
orb03	1005	t.o.	t.o.	t.o.	t.o.	3045	t.o.	3515	3182	2730	892	712	589
orb04	1005	2736	1155	2280	482	347	481	831	856	770	82	100	114
orb05	887	t.o.	1475	2618	851	342	768	942	1285	843	103	104	102
orb06	1010	t.o.	t.o.	t.o.	t.o.	1795	2893	2547	2072	1948	503	485	472
orb07	397	356	223	345	126	110	144	125	127	120	30	28	29
orb08	899	3020	1476	1934	941	545	633	980	1316	876	103	101	112
orb09	934	t.o.	1755	2376	802	460	571	978	812	787	96	108	131
orb10	944	3232	1425	2162	647	439	450	749	852	694	73	79	85
Total [sec]		35649	19800	28115	15981	10472	14406	17871	17628	15103	15968	12934	13751
Speedup			1.80	1.27		1.53	1.11		1.01	1.18		1.23	1.16

ford encoding. LRS (CRW) adopts Chaff and MINISAT as a SAT solver, and can use two kinds of search strategy, incremental search and binary search. The only difference between LRS and CRW is whether lemmas are reused or not. LRS10 is the special version of LRS which reuses only the conflict clauses whose length are 10 or less.

In the incremental search for JSSP, the size of generated SAT problems increases gradually. This means that we can reuse lemmas for solving the succeeding SAT problems. However, in the binary search for JSSP, there are some cases that we can not reuse lemmas. For example, Figure 6 shows that we solve SAT problems in the following order:  $P_{221}, P_{135}, P_{178}, P_{157}, \dots$ . We can reuse lemmas generated from  $P_{135}$  for solving the next problem  $P_{178}$ , but can not reuse lemmas generated from  $P_{221}$  for  $P_{135}$ . Because the lemma-reusability condition between  $P_{221}$  and  $P_{135}$  is not satisfied. Hence, in the binary search, although the optimal value can be quickly found compared with the incremental search, the effect of lemma-reusing is restrained.

We executed 12 combinations of three kinds of JSSP solvers (CRW, LRS and LRS10), two kinds of SAT solvers

(Chaff and MINISAT), and two kinds of search strategy (incremental and binary search). We limited the execution time of each JSSP solver to 1 CPU hour. Benchmark problems were taken from OR-library (Beasley 1990) and Sadeh's problems in (Sadeh 1991). OR-library contains 82 JSSPs. We show the results of 30 solved problems out of 82 in Table 2. Sadeh's benchmark problems consists of 46 JSSPs. We solved 20 problems in which due dates and release dates of all jobs are the same. Table 3 shows the results of the 20 problems. "Opt" is the minimum makespan. "m.o." means memory overflow. In Table 3, a problem name "eX-Y" is abbreviation of the original problem name e0ddrX-10-by-5-Y. "Total" is the total of CPU time of a JSSP solver. But, the calculation is more complicated than Table 1. We regarded "t.o." as 3600 seconds if any JSSP solvers of CRW, LRS and LRS10 solved the problem, and added it to the total time. Hence, note that comparison of the total time in the combination of different SAT solvers or different search strategy is impossible. For example, LRS with Chaff using binary search is not the fastest. LRS with MINISAT using binary search is the fastest in Table 2.

Table 3: Experimental results of Sadeh’s benchmark problems

Prob	Opt	Chaff						MINISAT					
		Incremental search			Binary search			Incremental search			Binary search		
		CRW [sec]	LRS [sec]	LRS10 [sec]	CRW [sec]	LRS [sec]	LRS10 [sec]	CRW [sec]	LRS [sec]	LRS10 [sec]	CRW [sec]	LRS [sec]	LRS10 [sec]
e1-1	140	2858	t.o.	2051	1361	708	774	901	1929	626	233	235	219
e1-2	146	976	1281	633	416	353	318	364	698	282	137	169	126
e1-3	137	t.o.	t.o.	3217	1900	t.o.	1774	1304	3317	892	443	806	497
e1-4	136	578	760	383	210	93	95	224	423	176	56	59	72
e1-5	127	t.o.	t.o.	t.o.	t.o.	t.o.	2585	1233	3121	1017	600	982	678
e1-6	139	961	1893	620	446	351	236	388	814	297	133	203	128
e1-7	144	t.o.	t.o.	2606	2767	t.o.	1953	1507	3015	960	467	843	536
e1-8	129	1437	2099	791	544	432	245	504	1251	345	183	256	177
e1-9	143	1181	1764	808	579	870	502	554	1163	422	204	240	200
e1-10	135	2790	t.o.	1590	1063	1434	763	846	1618	536	305	293	294
e2-1	157	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	1442	t.o.	1132	626	801	755
e2-2	150	t.o.	t.o.	t.o.	3252	t.o.	2288	1132	t.o.	1054	589	1291	714
e2-3	141	t.o.	t.o.	2916	t.o.	t.o.	3246	1723	t.o.	1156	881	1116	634
e2-4	141	t.o.	t.o.	2901	1121	1052	1383	1331	2922	867	334	485	273
e2-5	145	t.o.	t.o.	3264	2320	2004	1214	980	3578	986	361	811	411
e2-6	151	2976	t.o.	1226	983	1303	667	923	2168	727	223	298	239
e2-7	145	t.o.	t.o.	t.o.	2805	1536	1235	1518	t.o.	1131	364	420	456
e2-8	162	t.o.	t.o.	2183	1127	821	735	796	1823	582	356	515	319
e2-9	138	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	1239	t.o.	1204	613	1507	1052
e2-10	162	1214	2961	915	608	931	350	517	1313	470	336	624	247
Total [sec]		36570	39558	26103	28702	26288	20361	19426	47153	14862	7442	11953	8028
Speedup			0.92	1.40		1.09	1.41		0.41	1.31		0.62	0.93

Table 4: The detail of LRS and LRS10 with MINISAT using binary search in e2-9

Step	Makespan	Clauses	LRS			LRS10			Result
			Lemmas	Ave Len	[sec]	Lemmas	Ave Len	[sec]	
1	206	127802	267	20.9	0.2	146	6.9	0.2	SAT
2	130	79162	746	17.6	0.7	392	6.6	0.6	UNSAT
3	168	103482	342	14.7	0.2	75	6.4	0.1	SAT
4	149	91322	14201	61.4	15.6	4315	7.1	13.6	UNSAT
5	158	97082	84158	84.2	246.3	18691	7.5	231.6	UNSAT
6	163	100282	129	42.2	1.8	11706	7.6	126.9	SAT
7	160	98362	121860	105.1	459.4	15554	7.7	294.0	UNSAT
8	161	99002	27801	29.4	770.2	13130	7.8	377.5	UNSAT
9	162	99642	680	57.3	7.5	699	7.8	5.8	SAT

In OR-library, LRS is good, while in Sadeh’s benchmark LRS10 is good. In Sadeh’s benchmark, there was a tendency to generate a lot of lemmas. Sometimes the number of conflict clauses exceeds that of the original problem. Hence, LRS10 had good performance compared with LRS. Table 4 shows the detail of e2-9 which is the worst case of LRS. Step 1 denotes that LRS first generated the SAT problem  $P_{206}$  which consists of 127802 clauses. Then MINISAT solved  $P_{206}$  in 0.2 CPU seconds and generated 267 lemmas whose average of length is 20.9 (but, LRS can not reuse these lemmas for solving other SAT problems because  $P_{206}$  is the biggest problem in e2-9). The result of  $P_{206}$  was satisfiable.

### Related Works

Some lemma-reusing approaches have been proposed in the study of bounded model checking (BMC). Whittemore et al. have proposed a general-purpose framework of lemma-

reusing (Whittemore, Kim, & Sakallah 2001). The approach records the set of clauses  $R$  used for constructing a conflict clause  $c$ . If a new SAT problem contains  $R$ , then  $c$  is reusable for solving the problem. However, keeping track of such dependencies may be expensive. The approach proposed by (Eén & Sörensson 2003b) is the same as ours. If two SAT problems  $P$  and  $Q$  satisfy the lemma-reusability condition, all conflict clauses generated from  $P$  are reusable for solving  $Q$ . There is no necessity to check the dependencies. Shtrichman showed that SAT problems translated by a SAT encoding in BMC have a common subset of clauses, and proposed an approach which fast distinguishes whether a conflict clause is generated from the subset or not (Shtrichman 2001). Jin et al. have extended the criterion of the common subset and have proposed an approach to distill conflict clauses (elimination of conflict clauses expected to be ineffective) (Jin & Somenzi 2005). As our experimental results

showed, naive reuse of lemmas may have harmful effect. Hence, the screening of good lemmas is one of the important future work.

In the study of distributed/parallel SAT solving, the technique of lemma exchange is used widely (Soh *et al.* 2005; Chrabakh & Wolski 2003; Sinz, Blochinger, & Küchlin 2001). The task of parallel SAT solving is to solve efficiently one SAT problem in distributed computing environment, and lemma exchange is performed between different SAT solvers which solve the same SAT problem basically. For example, Soh *et al.* have proposed an approach for solving JSSP using multiple and heterogeneous SAT solvers in parallel. These SAT solvers can be executed cooperatively by exchanging lemmas for solving the same SAT problem. The study of parallel SAT solving for an incremental SAT problem with lemma-reusing is also interesting future work.

## Conclusion

Generally, lemmas deduced from a certain SAT problem are not applicable for solving other SAT problems. In this paper, we have shown the lemma-reusability condition, proved that Graphplan-based encoding, action-based encoding and Crawford encoding satisfy the condition, and substantiated the usefulness of lemma-reusing approach for the SAT based planning and scheduling.

In the SAT based planning and scheduling, generated SAT problems have many common clauses, that is, many common parts of the search space. This indicates that it is reasonable to reuse lemmas. And also, our approach only extracts lemmas from a SAT solver, and simply adds it to the next SAT problem. Therefore, our approach makes it possible to use the latest SAT solvers more efficiently for the SAT based planning and scheduling.

**Acknowledgements.** The authors are grateful to reviewers for many useful comments.

## References

- Beasley, J. E. 1990. OR-library: distributing test problems by electronic mail. *Operational Research Society* 41:1069–1072.
- Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1–2):279–298.
- Cadoli, M., and Schaerf, A. 2005. Compiling problem specifications into sat. *Artificial Intelligence* 162:89–120.
- Chrabakh, W., and Wolski, R. 2003. GridSAT: A chaff-based distributed sat solver for the grid. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 37. IEEE Computer Society.
- Crawford, J. M., and Baker, A. B. 1994. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of AAAI-94*, 1092–1097.
- Eén, N., and Biere, A. 2005. Effective preprocessing in sat through variable and clause elimination. In *Proceedings of SAT-2005*, 61–75.
- Eén, N., and Sörensson, N. 2003a. An extensible sat-solver. In *Proceedings of SAT-2003*, 502–518.
- Eén, N., and Sörensson, N. 2003b. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science* 89(4).
- Gershman, R. 2005. HAIFASAT: A new robust sat solver. <http://www.cs.technion.ac.il/~gershman/>.
- Jin, H., and Somenzi, F. 2005. An incremental algorithm to check satisfiability for bounded model checking. *Electronic Notes in Theoretical Computer Science* 119(2):51–65.
- Kautz, H., and Selman, B. 1998. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *AIPS98 Workshop on Planning as Combinatorial Search*, 58–60.
- Kautz, H.; McAllester, D.; and Selman, B. 1996. Encoding plans in propositional logic. In *Proceedings of KR-96*, 374–384.
- Kautz, H. 2004. SATPLAN04: Planning as satisfiability. Informal note of the International Planning Competition 2004.
- Marques-Silva, J. P., and Sakallah, K. A. 1999. GRASP – A search algorithm for propositional satisfiability. *IEEE Transactions on Computers* 48:506–521.
- Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC-01*, 530–535.
- Nabeshima, H.; Nozawa, H.; and Iwanuma, K. 2005. Effective sat planning by lemma-reusing. In *Proceedings of AIA-2005*, 1–6.
- Sadeh, N. 1991. *Look-ahead Techniques for Micro-opportunistic Job Shop Scheduling*. Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, PA 15213.
- Shtreichman, O. 2001. Pruning techniques for the SAT-based bounded model checking problem. In *Proceedings of CHARME-2001*, volume 2144 of LNCS, 58–70. Springer.
- Sinz, C.; Blochinger, W.; and Küchlin, W. 2001. PaSAT - parallel sat-checking with lemma exchange: Implementation and applications. In *Proceedings of SAT-2001*, 212–217.
- Soh, T.; Inoue, K.; Banbara, M.; and Tamura, N. 2005. Experimental results for solving job-shop scheduling problems with multiple SAT solvers. In *Proceedings of DSCP-2005*, 25–38.
- Vidal, V., and Geffner, H. 2006. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence* 170(3):298–335.
- Whittemore, J.; Kim, J.; and Sakallah, K. A. 2001. SATIRE: A new incremental satisfiability engine. In *Proceedings of DAC-2001*, 542–545.
- Zhang, L.; Madigan, C. F.; Moskewicz, M. W.; and Malik, S. 2001. Efficient conflict driven learning in boolean satisfiability solver. In *Proceedings of ICCAD-01*, 279–285.