Solving Constraint Satisfaction Problems by a SAT Solver

Naoyuki Tamura, Tomoya Tanjo, and Mutsunori Banbara

Kobe University, JAPAN

CICLOPS-WLPE-2010
A SAT-based Constraint Solver *Sugar*
Contents

1. SAT problems and SAT solvers
2. SAT encodings of Constraint Satisfaction Problems (CSP)
3. A SAT-based Constraint Solver **Sugar**
Contents

1. SAT problems and SAT solvers
2. SAT encodings of Constraint Satisfaction Problems (CSP)
3. A SAT-based Constraint Solver Sugar
4. Solving CSP by Examples
   - Open-Shop Scheduling (OSS) Problems
   - Latin Square Problems
5. Demonstration
   - Huge Sudoku Puzzles
   - Scala Interface
SAT problems and SAT solvers
SAT problems

**SAT** (Boolean satisfiability testing) is a problem to decide whether a given Boolean formula has any satisfying truth assignment.

- SAT is a central problem in Computer Science both theoretically and practically.
- SAT was the first NP-complete problem [Cook 1971].
- SAT has very efficient implementation (MiniSat, etc.)
- SAT-based approach is becoming popular in many areas.
SAT instances

SAT instances are given in the conjunctive normal form (CNF).

**CNF formula**

- A **CNF formula** is a conjunction of clauses.
- A **clause** is a disjunction of literals.
- A **literal** is either a Boolean variable or its negation.

**DIMACS CNF** is used as the standard format for CNF files.

```
P cnf 9 7 ; Number of variables and clauses
1 2 0 ; a ∨ b
9 3 0 ; c ∨ d
1 8 4 0 ; a ∨ e ∨ f
-2 -4 5 0 ; ¬b ∨ ¬f ∨ g
-4 6 0 ; ¬f ∨ h
-2 -6 7 0 ; ¬b ∨ ¬h ∨ i
-5 -7 0 ; ¬g ∨ ¬i
```

Naoyuki Tamura, Tomoya Tanjo, and Mutsunori Banbara
Solving Constraint Satisfaction Problems by a SAT Solver
SAT solvers

**SAT Solver**

SAT solver is a program to decide whether a given SAT instance is satisfiable (SAT) or unsatisfiable (UNSAT).

Usually, it also returns a truth assignment as a solution when the instance is SAT.

- Systematic (complete) SAT solver answers SAT or UNSAT.
  - Most of them are based on the DPLL algorithm.
- Stochastic (incomplete) SAT solver only answers SAT (no answers for UNSAT).
  - Local search algorithms are used.
DPLL Algorithm

[Davis & Putnam 1960], [Davis, Logemann & Loveland 1962]

(1) function DPLL(S: a CNF formula, σ: a variable assignment)
(2) \( σ := \text{UP}(S, σ); \) /* unit propagation */
(3) if \( S \) is satisfied by \( σ \) then return true;
(4) if \( S \) is falsified by \( σ \) then return false;
(5) choose an unassigned variable \( x \) from \( Sσ; \)
(6) return DPLL(S, \( σ \cup \{x \mapsto 0\} \)) or DPLL(S, \( σ \cup \{x \mapsto 1\} \));

(1) function UP(S: a CNF formula, σ: a variable assignment)
(2) while \( Sσ \) contains a unit clause \( \{l\} \) do
(3) if \( l \) is positive then \( σ := σ \cup \{l \mapsto 1\} \);
(4) else \( σ := σ \cup \{\overline{l} \mapsto 0\} \);
(5) return \( σ \);

- \( Sσ \) represents a CNF formula obtained by applying \( σ \) to \( S \).
1. Choose $a$ and decide $a \mapsto 0$

$C_1 : a \lor b$
$C_2 : c \lor d$
$C_3 : a \lor e \lor f$
$C_4 : \neg b \lor \neg f \lor g$
$C_5 : \neg f \lor h$
$C_6 : \neg b \lor \neg h \lor i$
$C_7 : \neg g \lor \neg i$
Choose $a$ and decide $a \mapsto 0$

- Propagate $b \mapsto 1$ from $C_1$

\[\begin{align*}
C_1 & : a \lor b \\
C_2 & : c \lor d \\
C_3 & : a \lor e \lor f \\
C_4 & : \neg b \lor \neg f \lor g \\
C_5 & : \neg f \lor h \\
C_6 & : \neg b \lor \neg h \lor i \\
C_7 & : \neg g \lor \neg i
\end{align*}\]
DPLL

Choose \( a \) and decide \( a \mapsto 0 \)
- Propagate \( b \mapsto 1 \) from \( C_1 \)

Choose \( c \) and decide \( c \mapsto 0 \)

\[
\begin{align*}
C_1 : & a \lor b \\
C_2 : & c \lor d \\
C_3 : & a \lor e \lor f \\
C_4 : & \neg b \lor \neg f \lor g \\
C_5 : & \neg f \lor h \\
C_6 : & \neg b \lor \neg h \lor i \\
C_7 : & \neg g \lor \neg i
\end{align*}
\]
Naoyuki Tamura, Tomoya Tanjo, and Mutsunori Banbara
Solving Constraint Satisfaction Problems by a SAT Solver

**DPLL**

\[
C_1 : a \lor b \\
C_2 : c \lor d \\
C_3 : a \lor e \lor f \\
C_4 : \neg b \lor \neg f \lor g \\
C_5 : \neg f \lor h \\
C_6 : \neg b \lor \neg h \lor i \\
C_7 : \neg g \lor \neg i
\]

1. Choose \( a \) and decide \( a \mapsto 0 \)
   - Propagate \( b \mapsto 1 \) from \( C_1 \)
2. Choose \( c \) and decide \( c \mapsto 0 \)
   - Propagate \( d \mapsto 1 \) from \( C_2 \)

Conflict occurred at \( C_6 \)

Backtrack and decide \( e \mapsto 1 \)
Naoyuki Tamura, Tomoya Tanjo, and Mutsunori Banbara  
Solving Constraint Satisfaction Problems by a SAT Solver

DPLL

\[
C_1 : a \lor b \\
C_2 : c \lor d \\
C_3 : a \lor e \lor f \\
C_4 : \neg b \lor \neg f \lor g \\
C_5 : \neg f \lor h \\
C_6 : \neg b \lor \neg h \lor i \\
C_7 : \neg g \lor \neg i
\]

1. Choose \(a\) and decide \(a \leftarrow 0\)
   - Propagate \(b \leftarrow 1\) from \(C_1\)
2. Choose \(c\) and decide \(c \leftarrow 0\)
   - Propagate \(d \leftarrow 1\) from \(C_2\)
3. Choose \(e\) and decide \(e \leftarrow 0\)

Conflict occurred at \(C_6\).

Backtrack and decide \(e \leftarrow 1\)
Choose a and decide $a \mapsto 0$
- Propagate $b \mapsto 1$ from $C_1$

Choose c and decide $c \mapsto 0$
- Propagate $d \mapsto 1$ from $C_2$

Choose e and decide $e \mapsto 0$
- Propagate $f \mapsto 1$ from $C_3$
DPLL

1. Choose $a$ and decide $a \leftarrow 0$
   - Propagate $b \leftarrow 1$ from $C_1$
2. Choose $c$ and decide $c \leftarrow 0$
   - Propagate $d \leftarrow 1$ from $C_2$
3. Choose $e$ and decide $e \leftarrow 0$
   - Propagate $f \leftarrow 1$ from $C_3$
   - Propagate $g \leftarrow 1$ from $C_4$

Conflict occurred at $C_6$.

\[ C_1 : a \lor b \]
\[ C_2 : c \lor d \]
\[ C_3 : a \lor e \lor f \]
\[ C_4 : \neg b \lor \neg f \lor g \]
\[ C_5 : \neg f \lor h \]
\[ C_6 : \neg b \lor \neg h \lor i \]
\[ C_7 : \neg g \lor \neg i \]
Choose a and decide $a \leftarrow 0$
- Propagate $b \leftarrow 1$ from $C_1$

Choose c and decide $c \leftarrow 0$
- Propagate $d \leftarrow 1$ from $C_2$

Choose e and decide $e \leftarrow 0$
- Propagate $f \leftarrow 1$ from $C_3$
- Propagate $g \leftarrow 1$ from $C_4$
- Propagate $i \leftarrow 0$ from $C_7$
C_1 : a \lor b  
C_2 : c \lor d  
C_3 : a \lor e \lor f  
C_4 : \neg b \lor \neg f \lor g  
C_5 : \neg f \lor h  
C_6 : \neg b \lor \neg h \lor i  
C_7 : \neg g \lor \neg i  

1. Choose a and decide $a \leftarrow 0$
   - Propagate $b \leftarrow 1$ from $C_1$
2. Choose c and decide $c \leftarrow 0$
   - Propagate $d \leftarrow 1$ from $C_2$
3. Choose e and decide $e \leftarrow 0$
   - Propagate $f \leftarrow 1$ from $C_3$
   - Propagate $g \leftarrow 1$ from $C_4$
   - Propagate $i \leftarrow 0$ from $C_7$
   - Propagate $h \leftarrow 1$ from $C_5$

Conflict occurred at $C_6$. 

Backtrack and decide $e \leftarrow 1$
DPLL

\[ C_1 : a \lor b \]
\[ C_2 : c \lor d \]
\[ C_3 : a \lor e \lor f \]
\[ C_4 : \neg b \lor \neg f \lor g \]
\[ C_5 : \neg f \lor h \]
\[ C_6 : \neg b \lor \neg h \lor i \]
\[ C_7 : \neg g \lor \neg i \]

1. Choose \( a \) and decide \( a \leftrightarrow 0 \)
   - Propagate \( b \leftrightarrow 1 \) from \( C_1 \)

2. Choose \( c \) and decide \( c \leftrightarrow 0 \)
   - Propagate \( d \leftrightarrow 1 \) from \( C_2 \)

3. Choose \( e \) and decide \( e \leftrightarrow 0 \)
   - Propagate \( f \leftrightarrow 1 \) from \( C_3 \)
   - Propagate \( g \leftrightarrow 1 \) from \( C_4 \)
   - Propagate \( i \leftrightarrow 0 \) from \( C_7 \)
   - Propagate \( h \leftrightarrow 1 \) from \( C_5 \)
   - Conflict occurred at \( C_6 \)
**DPLL**

\[
\begin{align*}
C_1 & : a \lor b \\
C_2 & : c \lor d \\
C_3 & : a \lor e \lor f \\
C_4 & : \neg b \lor \neg f \lor g \\
C_5 & : \neg f \lor h \\
C_6 & : \neg b \lor \neg h \lor i \\
C_7 & : \neg g \lor \neg i
\end{align*}
\]

1. Choose \( a \) and decide \( a \mapsto 0 \)
   - Propagate \( b \mapsto 1 \) from \( C_1 \)

2. Choose \( c \) and decide \( c \mapsto 0 \)
   - Propagate \( d \mapsto 1 \) from \( C_2 \)

3. Choose \( e \) and decide \( e \mapsto 0 \)
   - Propagate \( f \mapsto 1 \) from \( C_3 \)
   - Propagate \( g \mapsto 1 \) from \( C_4 \)
   - Propagate \( i \mapsto 0 \) from \( C_7 \)
   - Propagate \( h \mapsto 1 \) from \( C_5 \)
   - Conflict occurred at \( C_6 \)

4. Backtrack and decide \( e \mapsto 1 \)
Modern SAT solvers

- The following techniques have been introduced to DPLL and they drastically improved the performance of modern SAT solvers.
  - **CDCL** (Conflict Driven Clause Learning) [Silva 1996]
  - Non-chronological Backtracking [Silva 1996]
  - Random Restarts [Gomes 1998]
  - Watched Literals [Moskewicz & Zhang 2001]
  - Variable Selection Heuristics [Moskewicz & Zhang 2001]
- **Chaff** and **zChaff** solvers made one to two orders magnitude improvement [2001].
- SAT competitions and SAT races since 2002 contribute to the progress of SAT solver implementation techniques.
- **MiniSat** solver showed its good performance in the 2005 SAT competition with about 2000 lines of code in C++.
- Modern SAT solvers can handle instances with more than $10^6$ variables and $10^7$ clauses.
CDCL (Conflict Driven Clause Learning)

- At conflict, a reason of the conflict is extracted as a clause and it is remembered as a learnt clause.
- Learnt clauses significantly prunes the search space in the further search.
- Learnt clause is generated by resolution in backward direction.
- The resolution is stopped at First UIP (Unique Implication Point) [Moskewicz & Zhang 2001].

In the previous example, $\neg b \lor \neg f$ is generated as a learnt clause.

\[
\begin{align*}
C_6 & : \neg b \lor \neg h \lor i & C_5 & : \neg f \lor h \\
\therefore \neg b \lor \neg f \lor i & & \text{C7 : } \neg g \lor \neg i \\
\quad \neg b \lor \neg f \lor \neg g & & \text{C4 : } \neg b \lor \neg f \lor g \\
\therefore \neg b \lor \neg f \quad \text{[Moskewicz & Zhang 2001]} & & \text{Naoyuki Tamura, Tomoya Tanjo, and Mutsunori Banbara}
\end{align*}
\]
SAT-based Approach

SAT-based approach is becoming popular for solving hard combinatorial problems.

- Planning (SATPLAN, Blackbox) [Kautz & Selman 1992]
- Automatic Test Pattern Generation [Larrabee 1992]
- Job-shop Scheduling [Crawford & Baker 1994]
- Software Specification (Alloy) [1998]
- Bounded Model Checking [Biere 1999]
- Software Package Dependency Analysis (SATURN)
  - SAT4J is used in Eclipse 3.4.
- Rewriting Systems (AProVE, Jambox)
- Answer Set Programming (clasp, Cmodels-2)
- FOL Theorem Prover (iProver, Darwin)
- First Order Model Finder (Paradox)
- Constraint Satisfaction Problems (Sugar) [Tamura et al. 2006]
Why SAT-based? (personal opinions)

SAT solvers are very fast.

- Clever implementation techniques, such as two literal watching.
  - It minimizes house-keeping informations for backtracking.
- Cache-aware implementation [Zhang & Malik 2003]
  - For example, a SAT-encoded Open-shop Scheduling problem instance gp10-10 is solved within 4 seconds with more than 99% cache hit rate by MiniSat.

```
$ valgrind --tool=cachegrind minisat gp10-10-1091.cnf
L2 refs: 42,842,531 ( 31,633,380 rd +11,209,151 wr)
L2 misses: 25,674,308 ( 19,729,255 rd + 5,945,053 wr)
L2 miss rate: 0.4% ( 0.4% + 1.0% )
```
Why SAT-based? (personal opinions)

SAT-based approach is similar to RISC approach in ’80s by Patterson.

- **RISC**: Reduced Instruction Set Computer
- Patterson claimed a computer of a “reduced” and fast instruction set with an efficient optimizing compiler can be faster than a “complex” computer (CISC).

In that sense, study of both SAT solvers and SAT encodings are important and interesting topics.
SAT encodings
of
Constraint Satisfaction Problems
**Finite linear CSP**

- **Integer variables** with finite domains
  - \( \ell(x) \): the lower bound of \( x \)
  - \( u(x) \): the upper bound of \( x \)

- **Boolean variables**

- **Arithmetic operators**: +, −, constant multiplication, etc.

- **Comparison operators**: =, ≠, ≥, >, ≤, <

- **Logical operators**: ¬, ∧, ∨, ⇒

- We can restrict the comparison to \( \sum a_i x_i \leq c \) without loss of generality where \( x_i \)'s are integer variables and \( a_i \)'s and \( c \) are integer constants.

- We also use the followings in further descriptions.
  - \( n \): number of integer variables
  - \( d \): maximum domain size of integer expressions
There have been several methods proposed to encode CSP into SAT.

- **Direct encoding** is the most widely used one [de Kleer 1989].
- **Order encoding** is a new encoding showing a good performance for a wide variety of problems [Tamura et al. 2006].
  - It is first used to encode job-shop scheduling problems by [Crawford & Baker 1994].
  - It succeeded to solve previously undecided problems in open-shop scheduling, job-shop scheduling, and two-dimensional strip packing.

Other encodings:

- **Multivalued encoding** [Selman 1992]
- **Support encoding** [Kasif 1990]
- **Log encoding** [Iwama 1994]
- **Log-support encoding** [Gavanelli 2007]
In direct encoding [de Kleer 1989], a Boolean variable $p(x = i)$ is defined as true iff the integer variable $x$ has the domain value $i$, that is, $x = i$.

For example, the following five Boolean variables are used to encode an integer variable $x \in \{2, 3, 4, 5, 6\}$,

\[
p(x = 2) \quad p(x = 3) \quad p(x = 4) \quad p(x = 5) \quad p(x = 6)
\]
The following at-least-one and at-most-one clauses are required to make \( p(x = i) \) be true iff \( x = i \).

**Clauses for each integer variable \( x \)**

\[
p(x = \ell(x)) \lor \cdots \lor p(x = u(x)) \\
\neg p(x = i) \lor \neg p(x = j) \\
\quad (\ell(x) \leq i < j \leq u(x))
\]

For example, 11 clauses are required for \( x \in \{2, 3, 4, 5, 6\} \).

**11 clauses for \( x \in \{2, 3, 4, 5, 6\} \)**

\[
\begin{align*}
p(x = 2) & \lor p(x = 3) \lor p(x = 4) \lor p(x = 5) \lor p(x = 6) \\
\neg p(x = 2) & \lor \neg p(x = 3) \\
\neg p(x = 2) & \lor \neg p(x = 6) \\
\neg p(x = 3) & \lor \neg p(x = 6) \\
\neg p(x = 4) & \lor \neg p(x = 6) \\
\neg p(x = 4) & \lor \neg p(x = 5) \\
\neg p(x = 5) & \lor \neg p(x = 6)
\end{align*}
\]
A constraint is encoded by enumerating its **conflict points**.

**Constraint clauses**

- When $x_1 = i_1, \ldots, x_k = i_k$ violates the constraint, the following clause is added.

\[
\neg p(x_1 = i_1) \lor \cdots \lor \neg p(x_k = i_k)
\]
A constraint $x + y \leq 7$ is encoded into the following 15 clauses by enumerating conflict points (crossed points).

### 15 clauses for $x + y \leq 7$

- $\neg p(x = 2) \lor \neg p(y = 6)$
- $\neg p(x = 3) \lor \neg p(y = 5)$
- $\neg p(x = 3) \lor \neg p(y = 6)$
- $\neg p(x = 4) \lor \neg p(y = 4)$
- $\neg p(x = 4) \lor \neg p(y = 5)$
- $\neg p(x = 4) \lor \neg p(y = 6)$
- $\neg p(x = 5) \lor \neg p(y = 3)$
- $\neg p(x = 5) \lor \neg p(y = 4)$
- $\neg p(x = 5) \lor \neg p(y = 5)$
- $\neg p(x = 5) \lor \neg p(y = 6)$
- $\neg p(x = 6) \lor \neg p(y = 2)$
- $\neg p(x = 6) \lor \neg p(y = 3)$
- $\neg p(x = 6) \lor \neg p(y = 4)$
- $\neg p(x = 6) \lor \neg p(y = 5)$
- $\neg p(x = 6) \lor \neg p(y = 6)$
Order encoding

In order encoding [Tamura et al. 2006], a Boolean variable $p(x \leq i)$ is defined as true iff the integer variable $x$ is less than or equal to the domain value $i$, that is, $x \leq i$.

<table>
<thead>
<tr>
<th>Boolean variables for each integer variable $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p(x \leq i)$</td>
</tr>
</tbody>
</table>

For example, the following four Boolean variables are used to encode an integer variable $x \in \{2, 3, 4, 5, 6\}$,

<table>
<thead>
<tr>
<th>4 Boolean variables for $x \in {2, 3, 4, 5, 6}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p(x \leq 2)$</td>
</tr>
</tbody>
</table>

Boolean variable $p(x \leq 6)$ is unnecessary since $x \leq 6$ is always true.
The following clauses are required to make $p(x \leq i)$ be true iff $x \leq i$.

### Clauses for each integer variable $x$

\[-p(x \leq i - 1) \lor p(x \leq i) \quad (\ell(x) < i < u(x))\]

For example, 3 clauses are required for $x \in \{2, 3, 4, 5, 6\}$.

#### 3 clauses for $x \in \{2, 3, 4, 5, 6\}$

- $\neg p(x \leq 2) \lor p(x \leq 3)$
- $\neg p(x \leq 3) \lor p(x \leq 4)$
- $\neg p(x \leq 4) \lor p(x \leq 5)$
Order encoding (cont.)

The following table shows possible satisfiable assignments for the given clauses.

\[ \neg p(x \leq 2) \lor p(x \leq 3) \]
\[ \neg p(x \leq 3) \lor p(x \leq 4) \]
\[ \neg p(x \leq 4) \lor p(x \leq 5) \]

<table>
<thead>
<tr>
<th>( p(x \leq 2) )</th>
<th>( p(x \leq 3) )</th>
<th>( p(x \leq 4) )</th>
<th>( p(x \leq 5) )</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( x = 2 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( x = 3 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( x = 4 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( x = 5 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( x = 6 )</td>
</tr>
</tbody>
</table>
### Satisfiable partial assignments

<table>
<thead>
<tr>
<th>$p(x \leq 2)$</th>
<th>$p(x \leq 3)$</th>
<th>$p(x \leq 4)$</th>
<th>$p(x \leq 5)$</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>$x = 2 .. 6$</td>
</tr>
<tr>
<td>−</td>
<td>−</td>
<td>−</td>
<td>1</td>
<td>$x = 2 .. 5$</td>
</tr>
<tr>
<td>−</td>
<td>−</td>
<td>1</td>
<td>1</td>
<td>$x = 2 .. 4$</td>
</tr>
<tr>
<td>−</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$x = 2 .. 3$</td>
</tr>
<tr>
<td>0</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>$x = 3 .. 6$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>−</td>
<td>−</td>
<td>$x = 4 .. 6$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>−</td>
<td>$x = 5 .. 6$</td>
</tr>
<tr>
<td>0</td>
<td>−</td>
<td>−</td>
<td>1</td>
<td>$x = 3 .. 5$</td>
</tr>
<tr>
<td>0</td>
<td>−</td>
<td>1</td>
<td>1</td>
<td>$x = 3 .. 4$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>−</td>
<td>1</td>
<td>$x = 4 .. 5$</td>
</tr>
</tbody>
</table>

“−” means undefined.

- Partial assignments on Boolean variables represent bounds of integer variables.
A constraint is encoded by enumerating its conflict regions instead of conflict points.

**Constraint clauses**

- When all points \((x_1, \ldots, x_k)\) in the region \(i_1 < x_1 \leq j_1, \ldots, i_k < x_k \leq j_k\) violate the constraint, the following clause is added.

\[
p(x_1 \leq i_1) \lor \neg p(x_1 \leq j_1) \lor \cdots \lor p(x_k \leq i_k) \lor \neg p(x_k \leq j_k)
\]
Order encoding (cont.)

Encoding a constraint \( x + y \leq 7 \)
Encoding a constraint $x + y \leq 7$

$\neg (y \geq 6)$
Encoding a constraint $x + y \leq 7$

$p(y \leq 5)$
Order encoding (cont.)

Encoding a constraint $x + y \leq 7$

$p(y \leq 5)$
$\neg(x \geq 3 \land y \geq 5)$
Encoding a constraint $x + y \leq 7$

\[
p(y \leq 5) \\
p(x \leq 2) \lor p(y \leq 4)
\]
Encoding a constraint \( x + y \leq 7 \)

\[
p(y \leq 5) \land p(x \leq 2) \lor p(y \leq 4) \land \neg (x \geq 4 \land y \geq 4)
\]
Order encoding (cont.)

Encoding a constraint $x + y \leq 7$

$p(y \leq 5)$
$p(x \leq 2) \lor p(y \leq 4)$
$p(x \leq 3) \lor p(y \leq 3)$
Order encoding (cont.)

Encoding a constraint \( x + y \leq 7 \)

\[
p(y \leq 5) \\
p(x \leq 2) \lor p(y \leq 4) \\
p(x \leq 3) \lor p(y \leq 3) \\
\neg (x \geq 5 \land y \geq 3)
\]
Order encoding (cont.)

Encoding a constraint $x + y \leq 7$

$p(y \leq 5)
\quad p(x \leq 2) \lor p(y \leq 4)
\quad p(x \leq 3) \lor p(y \leq 3)
\quad p(x \leq 4) \lor p(y \leq 2)$
Encoding a constraint \( x + y \leq 7 \)

\[
\begin{align*}
p(y \leq 5) \\
p(x \leq 2) \lor p(y \leq 4) \\
p(x \leq 3) \lor p(y \leq 3) \\
p(x \leq 4) \lor p(y \leq 2) \\
\neg(x \geq 6)
\end{align*}
\]
Order encoding (cont.)

Encoding a constraint \( x + y \leq 7 \)

- \( p(y \leq 5) \)
- \( p(x \leq 2) \lor p(y \leq 4) \)
- \( p(x \leq 3) \lor p(y \leq 3) \)
- \( p(x \leq 4) \lor p(y \leq 2) \)
- \( p(x \leq 5) \)

\[ \begin{array}{cccccccc}
    & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
0 & x & x & x & x & x & x & x & x & x \\
1 & x & x & x & x & x & x & x & x & x \\
2 & x & x & x & x & x & x & x & x & x \\
3 & x & x & x & x & x & x & x & x & x \\
4 & x & x & x & x & x & x & x & x & x \\
5 & x & x & x & x & x & x & x & x & x \\
6 & x & x & x & x & x & x & x & x & x \\
7 & x & x & x & x & x & x & x & x & x \\
\end{array} \]
Bound propagation in order encoding

Encoding a constraint $x + y \leq 7$

$C_1 : \; p(y \leq 5)$
$C_2 : \; p(x \leq 2) \lor p(y \leq 4)$
$C_3 : \; p(x \leq 3) \lor p(y \leq 3)$
$C_4 : \; p(x \leq 4) \lor p(y \leq 2)$
$C_5 : \; p(x \leq 5)$

- When $p(x \leq 3)$ becomes false (i.e. $x \geq 4$), $p(y \leq 3)$ becomes true (i.e. $y \leq 3$) by unit propagation on $C_3$.
- This corresponds to the bound propagation in CSP solvers.
A SAT-based Constraint Solver
Sugar
**Sugar: a SAT-based Constraint Solver**

- **Sugar** is a constraint solver based on the **order encoding**.
- In the **2008 CSP solver competition**, Sugar became the **winner in GLOBAL category**.
- In the **2008 Max-CSP solver competition**, Sugar became the **winner in three categories** of INTENSIONAL and GLOBAL constraints.
- In the **2009 CSP solver competition**, Sugar became the **winner in three categories** of GLOBAL constraints.
Components of Sugar

- **Java program**
  - Parser
  - Linearizer
  - Simplifier of eliminating variable domains by General Arc Consistency algorithm
  - Encoder based on the order encoding
  - Decoder

- **External SAT solver**
  - MiniSat (default), PicoSAT, and any other SAT solvers

- **Perl script**
  - Command line script
Translation of constraints

- Linear constraints are translated by the order encoding.
- Non-linear constraints are translated in linear forms as follows:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E = F$</td>
<td>$(E \leq F) \land (E \geq F)$</td>
</tr>
<tr>
<td>$E \neq F$</td>
<td>$(E &lt; F) \lor (E &gt; F)$</td>
</tr>
<tr>
<td>$\max(E, F)$</td>
<td>$x$ with $(x \geq E) \land (x \geq F) \land ((x \leq E) \lor (x \leq F))$</td>
</tr>
<tr>
<td>$\min(E, F)$</td>
<td>$x$ with $(x \leq E) \land (x \leq F) \land ((x \geq E) \lor (x \geq F))$</td>
</tr>
<tr>
<td>$\text{abs}(E)$</td>
<td>$\max(E, -E)$</td>
</tr>
<tr>
<td>$E \text{ div } c$</td>
<td>$q$ with $(E = c \cdot q + r) \land (0 \leq r) \land (r &lt; c)$</td>
</tr>
<tr>
<td>$E \text{ mod } c$</td>
<td>$r$ with $(E = c \cdot q + r) \land (0 \leq r) \land (r &lt; c)$</td>
</tr>
</tbody>
</table>
Translation of global constraints

- \( \text{alldifferent}(x_1, x_2, \ldots, x_n) \) constraint is translated as follows:

\[
\bigwedge_{i < j} (x_i \neq x_j) \\
\bigvee_i (x_i \geq lb + n - 1) \\
\bigvee_i (x_i \leq ub - n + 1)
\]

where the last two are extra pigeon hole clauses, and \( lb \) and \( ub \) are the lower and upper bounds of \( \{x_1, x_2, \ldots, x_n\} \).

- Other global constraints (element, weightedsum, cumulative, etc.) are translated in a straightforward way.
Solving CSP by Examples

- Open-Shop Scheduling (OSS) Problems
- Latin Square Problems
Open-Shop Scheduling (OSS) Problems

- An OSS problem consists of $n$ jobs and $n$ machines.
  - $J_0, J_1, \ldots, J_{n-1}$
  - $M_0, M_1, \ldots, M_{n-1}$
- Each job $J_i$ consists of $n$ operations.
  - $O_{i0}, O_{i1}, \ldots, O_{i(n-1)}$
- An operation $O_{ij}$ of job $J_i$ is processed at machine $M_j$, and has a positive processing time $p_{ij}$.
- Operations of the same job $J_i$ must be processed sequentially but can be processed in any order.
- Each machine $M_j$ can handle one operation at a time.

Objective of OSS

- Minimize the completion time (makespan) of finishing all jobs.
- OSS is highly non-deterministic. OSS with $n$ jobs and $n$ machines has $(n!)^{2n}$ feasible scheduling.
OSS instance gp03-01

\[
(p_{ij}) = \begin{pmatrix}
661 & 6 & 333 \\
168 & 489 & 343 \\
171 & 505 & 324
\end{pmatrix}
\]

An optimal solution of gp03-01 (makespan=1168)
Defining integer variables

- $m$: makespan
- $s_{ij}$: start time of the operation $O_{ij}$

Defining constraints

- For each $s_{ij}$,
  \[ s_{ij} + p_{ij} \leq m \]
- For each pair of operations $O_{ij}$ and $O_{il}$ of the same job $J_i$,
  \[ (s_{ij} + p_{ij} \leq s_{il}) \lor (s_{il} + p_{il} \leq s_{ij}) \]
- For each pair of operations $O_{ij}$ and $O_{kj}$ of the same machine $M_j$,
  \[ (s_{ij} + p_{ij} \leq s_{kj}) \lor (s_{kj} + p_{kj} \leq s_{ij}) \]
Solving Constraint Satisfaction Problems by a SAT Solver

Naoyuki Tamura, Tomoya Tanjo, and Mutsunori Banbara
Solving gp03-01 by Sugar

Satisfiable case \( (m \leq 1168) \)

MiniSat finds a solution by performing 12 decisions and 1 conflict (backtrack).

Unsatisfiable case \( (m \cdot 1167) \)

MiniSat proves the unsatisfiability by performing 6 decisions and 5 conflicts (backtracks).

Efficient bound propagations were realized by unit propagations of MiniSat solver.
Satisfiable case \((m \leq 1168)\)

- MiniSat finds a solution by performing
  - 12 decisions and
  - 1 conflict (backtrack).
Solving gp03-01 by Sugar

**Satisfiable case** \((m \leq 1168)\)

- MiniSat finds a solution by performing
  - 12 decisions and
  - 1 conflict (backtrack).

**Unsatisfiable case** \((m \leq 1167)\)

- Efficient bound propagations were realized by unit propagations of MiniSat solver.
Solving gp03-01 by Sugar

**Satisfiable case** \((m \leq 1168)\)
- MiniSat finds a solution by performing
  - 12 decisions and
  - 1 conflict (backtrack).

**Unsatisfiable case** \((m \leq 1167)\)
- MiniSat proves the unsatisfiability by performing
  - 6 decisions and
  - 5 conflicts (backtracks).

- Efficient bound propagations were realized by unit propagations of MiniSat solver.
## Latin Square Problems

### Latin Square Problem of size 5

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>X11</td>
<td>X12</td>
<td>X13</td>
<td>X14</td>
<td>X15</td>
<td></td>
</tr>
<tr>
<td>X21</td>
<td>X22</td>
<td>X23</td>
<td>X24</td>
<td>X25</td>
<td></td>
</tr>
<tr>
<td>X31</td>
<td>X32</td>
<td>X33</td>
<td>X34</td>
<td>X35</td>
<td></td>
</tr>
<tr>
<td>X41</td>
<td>X42</td>
<td>X43</td>
<td>X44</td>
<td>X45</td>
<td></td>
</tr>
<tr>
<td>X51</td>
<td>X52</td>
<td>X53</td>
<td>X54</td>
<td>X55</td>
<td></td>
</tr>
</tbody>
</table>

- $x_{ij} \in \{1, 2, 3, 4, 5\}$

Latin Square of size 5 is satisfiable.
**Latin Square Problems**

### Latin Square Problem of size 5

<table>
<thead>
<tr>
<th>$x_{11}$</th>
<th>$x_{12}$</th>
<th>$x_{13}$</th>
<th>$x_{14}$</th>
<th>$x_{15}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{21}$</td>
<td>$x_{22}$</td>
<td>$x_{23}$</td>
<td>$x_{24}$</td>
<td>$x_{25}$</td>
</tr>
<tr>
<td>$x_{31}$</td>
<td>$x_{32}$</td>
<td>$x_{33}$</td>
<td>$x_{34}$</td>
<td>$x_{35}$</td>
</tr>
<tr>
<td>$x_{41}$</td>
<td>$x_{42}$</td>
<td>$x_{43}$</td>
<td>$x_{44}$</td>
<td>$x_{45}$</td>
</tr>
<tr>
<td>$x_{51}$</td>
<td>$x_{52}$</td>
<td>$x_{53}$</td>
<td>$x_{54}$</td>
<td>$x_{55}$</td>
</tr>
</tbody>
</table>

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldifferent in each row (5 rows)
### Latin Square Problem of size 5

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{11}$</td>
<td>$x_{12}$</td>
<td>$x_{13}$</td>
<td>$x_{14}$</td>
<td>$x_{15}$</td>
</tr>
<tr>
<td>$x_{21}$</td>
<td>$x_{22}$</td>
<td>$x_{23}$</td>
<td>$x_{24}$</td>
<td>$x_{25}$</td>
</tr>
<tr>
<td>$x_{31}$</td>
<td>$x_{32}$</td>
<td>$x_{33}$</td>
<td>$x_{34}$</td>
<td>$x_{35}$</td>
</tr>
<tr>
<td>$x_{41}$</td>
<td>$x_{42}$</td>
<td>$x_{43}$</td>
<td>$x_{44}$</td>
<td>$x_{45}$</td>
</tr>
<tr>
<td>$x_{51}$</td>
<td>$x_{52}$</td>
<td>$x_{53}$</td>
<td>$x_{54}$</td>
<td>$x_{55}$</td>
</tr>
</tbody>
</table>

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldifferent in each row (5 rows)
Latin Square Problems

Latin Square Problem of size 5

\[
\begin{array}{ccccc}
X_{11} & X_{12} & X_{13} & X_{14} & X_{15} \\
X_{21} & X_{22} & X_{23} & X_{24} & X_{25} \\
X_{31} & X_{32} & X_{33} & X_{34} & X_{35} \\
X_{41} & X_{42} & X_{43} & X_{44} & X_{45} \\
X_{51} & X_{52} & X_{53} & X_{54} & X_{55} \\
\end{array}
\]

- \(x_{ij} \in \{1, 2, 3, 4, 5\}\)
- alldifferent in each row (5 rows)
- alldifferent in each column (5 columns)
Latin Square Problems

Latin Square Problem of size 5

<table>
<thead>
<tr>
<th>x_{11}</th>
<th>x_{12}</th>
<th>x_{13}</th>
<th>x_{14}</th>
<th>x_{15}</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_{21}</td>
<td>x_{22}</td>
<td>x_{23}</td>
<td>x_{24}</td>
<td>x_{25}</td>
</tr>
<tr>
<td>x_{31}</td>
<td>x_{32}</td>
<td>x_{33}</td>
<td>x_{34}</td>
<td>x_{35}</td>
</tr>
<tr>
<td>x_{41}</td>
<td>x_{42}</td>
<td>x_{43}</td>
<td>x_{44}</td>
<td>x_{45}</td>
</tr>
<tr>
<td>x_{51}</td>
<td>x_{52}</td>
<td>x_{53}</td>
<td>x_{54}</td>
<td>x_{55}</td>
</tr>
</tbody>
</table>

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldifferent in each row (5 rows)
- alldifferent in each column (5 columns)
Latin Square Problems

### Latin Square Problem of size 5

<table>
<thead>
<tr>
<th></th>
<th>x_{11}</th>
<th>x_{12}</th>
<th>x_{13}</th>
<th>x_{14}</th>
<th>x_{15}</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_{21}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{22}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{23}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{24}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{25}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{31}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{32}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{33}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{34}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{35}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{41}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{42}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{43}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{44}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{45}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{51}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{52}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{53}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{54}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_{55}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \( x_{ij} \in \{1, 2, 3, 4, 5\} \)
- alldifferent in each row (5 rows)
- alldifferent in each column (5 columns)
- alldifferent in each diagonal (10 diagonals)
**Latin Square Problems**

**Latin Square Problem of size 5**

\[
\begin{array}{c|c|c|c|c}
  \text{x11} & \text{x12} & \text{x13} & \text{x14} & \text{x15} \\
  \hline
  \text{x21} & \text{x22} & \text{x23} & \text{x24} & \text{x25} \\
  \hline
  \text{x31} & \text{x32} & \text{x33} & \text{x34} & \text{x35} \\
  \hline
  \text{x41} & \text{x42} & \text{x43} & \text{x44} & \text{x45} \\
  \hline
  \text{x51} & \text{x52} & \text{x53} & \text{x54} & \text{x55} \\
\end{array}
\]

- \( x_{ij} \in \{1, 2, 3, 4, 5\} \)
- alldifferent in each row (5 rows)
- alldifferent in each column (5 columns)
- alldifferent in each diagonal (10 diagonals)
# Latin Square Problems

## Latin Square Problem of size 5

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{11}$</td>
<td>$x_{12}$</td>
<td>$x_{13}$</td>
<td>$x_{14}$</td>
<td>$x_{15}$</td>
</tr>
<tr>
<td>$x_{21}$</td>
<td>$x_{22}$</td>
<td>$x_{23}$</td>
<td>$x_{24}$</td>
<td>$x_{25}$</td>
</tr>
<tr>
<td>$x_{31}$</td>
<td>$x_{32}$</td>
<td>$x_{33}$</td>
<td>$x_{34}$</td>
<td>$x_{35}$</td>
</tr>
<tr>
<td>$x_{41}$</td>
<td>$x_{42}$</td>
<td>$x_{43}$</td>
<td>$x_{44}$</td>
<td>$x_{45}$</td>
</tr>
<tr>
<td>$x_{51}$</td>
<td>$x_{52}$</td>
<td>$x_{53}$</td>
<td>$x_{54}$</td>
<td>$x_{55}$</td>
</tr>
</tbody>
</table>

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldifferent in each row (5 rows)
- alldifferent in each column (5 columns)
- alldifferent in each diagonal (10 diagonals)
### Latin Square Problem of size 5

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{11}$</td>
<td>$x_{12}$</td>
<td>$x_{13}$</td>
<td>$x_{14}$</td>
<td>$x_{15}$</td>
</tr>
<tr>
<td>$x_{21}$</td>
<td>$x_{22}$</td>
<td>$x_{23}$</td>
<td>$x_{24}$</td>
<td>$x_{25}$</td>
</tr>
<tr>
<td>$x_{31}$</td>
<td>$x_{32}$</td>
<td>$x_{33}$</td>
<td>$x_{34}$</td>
<td>$x_{35}$</td>
</tr>
<tr>
<td>$x_{41}$</td>
<td>$x_{42}$</td>
<td>$x_{43}$</td>
<td>$x_{44}$</td>
<td>$x_{45}$</td>
</tr>
<tr>
<td>$x_{51}$</td>
<td>$x_{52}$</td>
<td>$x_{53}$</td>
<td>$x_{54}$</td>
<td>$x_{55}$</td>
</tr>
</tbody>
</table>

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldifferent in each row (5 rows)
- alldifferent in each column (5 columns)
- alldifferent in each diagonal (10 diagonals)

---

Naoyuki Tamura, Tomoya Tanjo, and Mutsunori Banbara
Solving Constraint Satisfaction Problems by a SAT Solver
Latin Square Problems

Latin Square Problem of size 5

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldifferent in each row (5 rows)
- alldifferent in each column (5 columns)
- alldifferent in each diagonal (10 diagonals)
- Latin Square of size 5 is satisfiable.

Naoyuki Tamura, Tomoya Tanjo, and Mutsunori Banbara
Solving Constraint Satisfaction Problems by a SAT Solver
## Performance comparison of Latin Square Problems

The table shows the CPU times (in seconds) of Latin Square Problems at 2009 CSP Solver Competition (“-” means timeout).

<table>
<thead>
<tr>
<th>Size</th>
<th>SAT/UNSAT</th>
<th>Sugar+m</th>
<th>Abscon</th>
<th>Mistral</th>
<th>bpsolver</th>
<th>Choco</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>UNSAT</td>
<td>0.57</td>
<td>0.66</td>
<td>0.01</td>
<td>0.03</td>
<td>0.41</td>
</tr>
<tr>
<td>4</td>
<td>UNSAT</td>
<td>0.59</td>
<td>0.63</td>
<td>0.01</td>
<td>0.03</td>
<td>0.41</td>
</tr>
<tr>
<td>5</td>
<td>SAT</td>
<td>0.73</td>
<td>0.68</td>
<td>0.01</td>
<td>0.03</td>
<td>0.58</td>
</tr>
<tr>
<td>6</td>
<td>UNSAT</td>
<td>0.79</td>
<td>0.82</td>
<td>0.03</td>
<td>0.17</td>
<td>0.73</td>
</tr>
<tr>
<td>7</td>
<td>SAT</td>
<td>0.94</td>
<td>0.78</td>
<td>0.01</td>
<td>0.04</td>
<td>0.77</td>
</tr>
<tr>
<td>8</td>
<td>UNSAT</td>
<td>1.01</td>
<td>676.75</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>UNSAT</td>
<td>1.08</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>UNSAT</td>
<td>1.16</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>SAT</td>
<td>1.35</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>UNSAT</td>
<td>1.66</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
### Effect of pigeon hole clauses

<table>
<thead>
<tr>
<th>Size</th>
<th>SAT/UNSAT</th>
<th>Sugar+m with p.h. clauses</th>
<th>Sugar+m w/o p.h. clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>UNSAT</td>
<td>0.44</td>
<td>0.35</td>
</tr>
<tr>
<td>4</td>
<td>UNSAT</td>
<td>0.47</td>
<td>0.41</td>
</tr>
<tr>
<td>5</td>
<td>SAT</td>
<td>0.44</td>
<td>0.43</td>
</tr>
<tr>
<td>6</td>
<td>UNSAT</td>
<td>0.52</td>
<td>0.40</td>
</tr>
<tr>
<td>7</td>
<td>SAT</td>
<td>0.80</td>
<td>0.69</td>
</tr>
<tr>
<td>8</td>
<td>UNSAT</td>
<td>1.08</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>UNSAT</td>
<td>0.98</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>UNSAT</td>
<td>3.12</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>SAT</td>
<td>1.59</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>UNSAT</td>
<td>3.23</td>
<td>-</td>
</tr>
</tbody>
</table>

- Pigeon hole clauses drastically improve the performance in both cases of SAT and UNSAT.
- They are well suited to the order encoding, and only two extra SAT clauses are required for each alldifferent constraint.
Demonstrations

- Huge Sudoku Puzzles
- Scala Interface
Huge Sudoku Puzzle

- It consists of 105 Sudoku puzzles of $9 \times 9$ overlapping at corners (created by Hirofumi Fujiwara).
- It contains 6885 cells, 1808 hints, and 2655 alldifferent constraints.
Huge Sudoku Puzzle

- It consists of 105 Sudoku puzzles of $9 \times 9$ overlapping at corners (created by Hirofumi Fujiwara).
- It contains 6885 cells, 1808 hints, and 2655 alldifferent constraints.
- *Sugar* can solve it in 30 seconds.
Scala Interface of Sugar

- Functional OOPL on JVM (Java Virtual Machine)
- Java class libraries can be used.
- Both compiler and interpreter (REPL) are available.
- It is useful to define a DSL (Domain Specific Language).
Example of Scala Interface

**Importing methods of Sugar**

```scala
import Sugar._
```

**Declaring variables**

```scala
v('x, 0, 7)
v('y, 0, 7)
```

**Adding constraints**

```scala
c('x + 'y === 7)
c('x * 2 + 'y * 4 === 20)
```

**Search a solution**

```scala
if (search)
  println(solution)
```
import Sugar._

def queens(n: Int) = {
  val qs = for (i <- 0 to n-1) yield 'q(i)
  qs.foreach(v(_, 0, n-1))
  c(Alldifferent(qs: _*))
  c(Alldifferent((0 to n-1).map(i => 'q(i) + i): _*))
  c(Alldifferent((0 to n-1).map(i => 'q(i) - i): _*))
  if (search)
    do {
      println(solution)
    } while (searchNext)
}
Summary

- We presented
  - Order encoding and
  - Sugar constraint solver.
- Sugar showed a good performance for a wide variety of problems.
- The source package can be downloaded from the following web page.
  - http://bach.istc.kobe-u.ac.jp/sugar/
- Sugar is developed as a software of the following project.
CSPSAT project (2008–2011)

Objective and Research Topics

R&D of efficient and practical SAT-based CSP solvers

- SAT encodings
  - CSP, Dynamic CSP, Temporal Logic, Distributed CSP
- Parallel SAT solvers
  - Multi-core, PC Cluster

Teams and Professors

- Kobe University (3)
- National Institute of Informatics (1)
- University of Yamanashi (3)
- Kyushu University (4)
- Waseda University (1)
Pointers

- SAT and SAT solvers
  - International Conference on Theory and Applications of Satisfiability Testing (SAT)
  - Journal on Satisfiability, Boolean Modeling and Computation
- SAT and LP
  - “Logic programming with satisfiability”, TPLP [Codish & Lagoon & Stuckey 2008]
  - “A Pearl on SAT Solving in Prolog”, FLOPS 2010 [Howe & King 2010]
- Papers on Sugar