Jin JW, Ma FF, Zhang J. Integrating standard dependency schemes in QCSP solvers. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 27(1): 37–41 Jan. 2012. DOI 10.1007/s11390-012-1204-5

Integrating Standard Dependency Schemes in QCSP Solvers

Ji-Wei Jin[1] (金继伟), Fei-Fei Ma[2] (马菲菲), *Member*, *ACM*
and Jian Zhang[3] (张健), *Senior Member*, *CCF*, *ACM*, *IEEE*

State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China

E-mail: {jinjw, maff, zj}@ios.ac.cn

Received July 1, 2011; revised October 17, 2011.

Abstract Quantified constraint satisfaction problems (QCSPs) are an extension to constraint satisfaction problems (CSPs) with both universal quantifiers and existential quantifiers. In this paper we apply variable ordering heuristics and integrate standard dependency schemes in QCSP solvers. The technique can help to decide the next variable to be assigned in QCSP solving. We also introduce a new factor into the variable ordering heuristics: a variable's *dep* is the number of variables depending on it. This factor represents the probability of getting more candidates for the next variable to be assigned. Experimental results show that variable ordering heuristics with standard dependency schemes and the new factor *dep* can improve the performance of QCSP solvers.

Keywords quantified constraint satisfaction problem, standard dependency scheme, variable ordering heuristics

1 Introduction

Quantified constraint satisfaction problems (QCSPs) are an extension to constraint satisfaction problems (CSPs) that allow variables to be universally quantified. It can be used to model various PSPACE-complete problems from areas such as planing under uncertainty, adversary game playing, model checking and so on. For example, when planning in a safety critical environment we require that an action is possible for every eventuality; in game playing we want to find a winning strategy for all possible moves of the opponent. In [1-2] the authors gave an application of QCSPs to factory scheduling, in [3] the authors used QCSPs to solve online bin packing problem. QCSPs have received increasing attention from the Artificial Intelligence (AI) community and some effective QCSP solvers have been developed recently, such as QCSP-solve[4-5], Blocksolve[6], QeCode[7], and Queso[1]. Some techniques are developed to improve the performance of QCSP solvers. Most of them are borrowed from CSP and quantified Boolean formula (QBF). However, few QCSP solvers use variable ordering heuristics during search, which have been successful in CSP. This motivates us to investigate the effect of variable ordering heuristics in QCSP solvers.

Variable and value ordering heuristics are useful for CSP solvers[8]. Value ordering heuristics have been investigated recently[9], but variable ordering heuristics have not received enough attention in QCSP community until now. Most QCSP solvers use *top-down* search approach, which assigns a variable each time and backjumps when necessary. During search the order of assignments should follow the quantification order of variables. So variable ordering heuristics can only be used within blocks. This limits the effect of variable ordering heuristics. In [10] the authors gave some results of variable ordering heuristics within blocks. In some cases they can improve the performance of QCSP solvers, while in other cases they do not work well. However, in some instances of QCSPs, some later variables are independent of former variables and it is not necessary to assign former variables first.

In this paper we integrate standard dependency schemes (D^{std}), which are borrowed from QBF, in QCSP solvers and implement variable ordering heuristics across blocks. A dependency scheme is a relation on variables indicating the dependence among them. In QBF standard dependency schemes and related techniques grant QBF solvers more freedom in the solution process[11-13]. These techniques have been introduced and integrated in QBF solvers successfully, but in QCSP standard dependency schemes have only been investigated in theory[14]. To our knowledge, the technique has not been integrated in QCSP solvers so far. In this paper we introduce and implement a QCSP algorithm with standard dependency schemes. We use the technique to help to decide the next variable to

Short Paper

This work is supported in part by the National Natural Science Foundation of China under Grant No. 61070039.

be assigned. Experiments demonstrate that variable ordering heuristics with standard dependency schemes can improve the performance of QCSP solvers. In this paper we also introduce a new factor for variable ordering heuristics, namely *dep*. For a variable x, the *dep* of x is the number of variables which depend on x. The larger the *dep* is, the more probable that we have more candidates to choose as the next variable when x is assigned. Experiments demonstrate that heuristics with *dep* perform better than those without *dep*.

This paper is organized as follows. Section 2 gives the necessary definitions about QCSPs. Section 3 describes the method to integrate standard dependency schemes in QCSP solvers and some other techniques used in our solver. Experiments and analysis are presented in Section 4. Section 5 concludes our contributions and discusses the future work.

2 Preliminaries

In this section we give some necessary definitions and terminologies about QCSPs. QCSPs are an extension to CSPs. A CSP consists of a set of variables $V = \{x_1, \ldots, x_n\}$ where each variable x_i is associated with a finite domain Dom_{x_i} of possible values, and a set of constraints $C = \{c_1, \ldots, c_m\}$ where each constraint involves some variables in V. We use $var(c)$ to denote the set of variables in c. A constraint c with $var(c) = \{x_1, \ldots, x_k\}$ is a set of tuples in $Dom_{x_1} \times \cdots \times Dom_{x_k}$, that is, $c \subseteq Dom_{x_1} \times \cdots \times Dom_{x_k}$. In semantics a constraint c restricts the combinations of values which can be taken simultaneously by the variables in c. A variable can only take values in its domain. Assignments are mappings from the set of variables to their domains. An assignment $\langle x_1 \mapsto a_1, \ldots, x_k \mapsto a_k \rangle$ satisfies a constraint c with $var(c) = \{x_1, \ldots, x_k\}$ if and only if the tuple $(a_1, \ldots, a_k) \in c$. For an assignment $x \mapsto a$, $c|_{x \mapsto a}$ denotes the subset of c which includes only tuples in which x takes the value a. If $x \notin var(c)$, then $c|_{x \mapsto a} = c$ for any value $a \in Dom_x$. For $C = \{c_1, \ldots, c_m\}$, $C|_{x \mapsto a}$ is the shorthand for $\{c_1|_{x \mapsto a}, \ldots, c_m|_{x \mapsto a}\}$. QCSPs are an extension to CSPs that allow variables to be universally quantified.

Definition 1. *A quantified constraint satisfaction problem (QCSP) is a formula of the form $Q_1 x_1 \cdots Q_n x_n C$. $Q_1 x_1 \cdots Q_n x_n$ is a sequence of quantifiers Q_i and variables x_i, $Q_i \in \{\forall, \exists\}$, $1 \leqslant i \leqslant n$. Each variable occurs exactly once in the sequence. C is a set of constraints, $\{c_1, \ldots, c_m\}$, where each c_i involves some variables among $x_1, \ldots, x_n$.*

We call $Q_1 x_1 \cdots Q_n x_n$ (Q for short) the prefix and C the matrix. A quantifier *block* of the prefix Q is a maximal contiguous subsequence of Q such that every variable in it has the same quantifier.

The semantics of a QCSP formula $\varphi = Q_1 x_1 \cdots Q_n x_n C$ can be defined recursively. If the prefix Q is empty or the matrix C is empty, then the formula is true. If Q_1 is $\exists$, then φ is true if and only if there exists a value a in Dom_{x_1} such that the formula $Q_2 x_2 \cdots Q_n x_n C|_{x_1 \mapsto a}$ is true. And if Q_1 is $\forall$, then φ is true if and only if for all values a in Dom_{x_1}, the formula $Q_2 x_2 \cdots Q_n x_n C|_{v_1 \mapsto a}$ is true.

For simplicity we restrict our attention to binary QCSPs, that is, the QCSPs where each constraint involves only two variables.

3 Algorithm

For a QCSP QC with m blocks, Q is written as $S_1 \cdots S_m$, where all quantifiers in S_i are the same. Also we associate Q with a linear order that $S_1 < \cdots < S_m$. A block S_i is an existential block ($\exists$ block) if all quantifiers in it are existential, otherwise it is a universal block ($\forall$ block). The set of all existential variables (resp. universal variables) is denoted by $V_\exists$ (resp. $V_\forall$). We use $q(S_i)$ to denote the quantifiers of S_i, and $q(x)$ the quantifier of the variable x. For each variable x in S_i we introduce a number $\delta(x) = i$ to indicate in which block it is. For an existential variable (resp. universal) x, $\overline{q(x)}$ is the quantifier $\forall$ (resp. $\exists$). For a quantifier $quant \in \{\exists, \forall\}$, $V_{quant,i}$ is the set of variables x with $q(x) = quant$ and $\delta(x) \geqslant i$, that is, the set of variables with *quant* quantifier and in blocks whose index is equal to or larger than i. For two variables x and y and a set of variables X, an *X-path* between x and y is a sequence $c_1, \ldots, c_k$ of constraints such that $x \in var(c_1)$ and $y \in var(c_k)$ and $var(c_i) \cap var(c_{i+1}) \cap X \neq \varnothing$, $1 \leqslant i < k$. The definition of standard dependency schemes is as follows[14-15].

Definition 2. *For $x \in V, i = \delta(x) + 1$, the standard dependency schemes of x is $D^{std}(x) = \{y \in V_{\overline{q(x)},i} |$ there exists an X-path between x and y for $X = V_{\exists,i}\}$*

For a variable x, $D^{std}(x)$ is the set of variables depending on x. In QCSP solving, these variables should be assigned after x. We use $\overline{D_u^{std}}(y)$ to denote the set of unassigned variables x such that $y \in D^{std}(x)$, that is, $\overline{D_u^{std}}(y)$ is the set of variables which should be assigned before y in QCSP solving and is unassigned at that time. $\overline{D_u^{std}}$ changes dynamically during search. It has been proven that in backtracking search algorithms, only a variable x with $\overline{D_u^{std}}(x) = \varnothing$ can be assigned[14].

Let us see an example. Let φ be a QCSP formula:

$$\varphi = \exists x_1 \forall y_1 \exists x_2 \exists x_3 \forall y_2 \forall y_3 \exists x_4 \{c_1, c_2, c_3, c_4, c_5, c_6\},$$

where $var(c_1) = \{x_1, x_4\}$, $var(c_2) = \{y_1, x_4\}$, $var(c_3) = \{x_1, y_3\}$, $var(c_4) = \{x_3, y_3\}$, $var(c_5) = \{y_2, x_4\}$, $var(c_6) = \{x_2, y_2\}$. Fig.1 shows the dependence

relation from Definition 2. We have: $D^{std}(x_1) = \{y_1, y_2, y_3\}$, $D^{std}(x_2) = \{y_2\}$, $D^{std}(x_3) = \{y_3\}$, $D^{std}(y_1) = D^{std}(y_2) = \{x_4\}$, $\overline{D_u^{std}}(x_1) = \overline{D_u^{std}}(x_2) = \overline{D_u^{std}}(x_3) = \varnothing$, $\overline{D_u^{std}}(x_4) = \{y_1, y_2\}$, $\overline{D_u^{std}}(y_1) = \{x_1\}$, $\overline{D_u^{std}}(y_2) = \{x_1, x_2\}$, $\overline{D_u^{std}}(y_3) = \{x_1, x_3\}$.

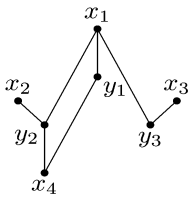


Fig.1. Example of dependence relation.

Variables y_1 and y_2 are not directly connected to x_1, where a variable is *directly connected to* another variable means that these two variables are in the same constraint. But they are directly connected to x_4 and x_4 is directly connected to x_1. By Definition 2, y_1 and y_2 depend on x_1. When solving φ, although x_2 and x_3 are after y_1 in the prefix, they can still be assigned before y_1 since they do not depend on any variables.

To compute standard dependency schemes we use methods similar to those in [12-13, 15]. For a universal variable x with $\delta(x) = i$, we have: $D_0(x) = \{y \in V_{\exists,i+1} |$ y is directly connected to $x\}$, $D_{k+1}(x) = \{z \in V_{\exists,i+1} |$ z is directly connected to $y \in D_k(x)\}$, $D^{std}(x) = \bigcup_k D_k(x)$. For an existential variable x with $\delta(x) = i$, we have: $D_0(x) = \{y' \in V_{\exists,i+1} | y'$ is directly connected to $x\}$, $D_{k+1}(x) = \{z' \in V_{\exists,i+1} | z'$ is directly connected to $y' \in D_k(x)\}$, $D(x) = \bigcup_k D_k(x)$, $D^{std}(x) = \{y \in V_{\forall,i} |$ y is directly connected to $z' \in D(x)\}$.

Our algorithm is described in detail as Algorithm 1. It is similar to QCSP-solve, except that we apply variable ordering heuristics and integrate standard dependency schemes in it. We compute standard dependency schemes at the beginning of search and update them during search. We compute D^{std} and $\overline{D_u^{std}}$ for all variables in *set_up_d_std*() by the method mentioned above. Variable ordering heuristics are implemented in *next_variable*() which chooses a variable to assign. During search *var* is the current variable being handled in the loop.

The look-ahead method used in our algorithm is forward checking (FC), the look-back method is conflict-based jumping (CBJ) and solution-directed pruning (SDP). We implement these techniques as those in [4]. We check if an assignment $\langle x \mapsto a \rangle$ is consistent with the subsequent constraints by FC. When back jumping, *back*(x) returns a variable x which is assigned before by CBJ or SDP. If no such a variable exists, that means the formula is false, *back*(x) returns FALSE and the search terminates. Also *back*(x) restores all domain changes caused by assignment for variables after x. For details of CBJ, SDP and FC we refer the reader to [16].

Algorithm 1. QCSP Solver

```
1   Boolean QCSP solver(QC)
2   set_up_d_std(); var ← next_variable();
3   while TRUE do
4       if var is existential then
5         if Dom_var is empty then
6             var ← back();
7         else
8             val ← next_value(var);
9             remove val from Dom_var;
10            if ⟨var ↦ val⟩ is consistent then
11              if all variables are assigned then
12                if no universal variables in Q then
13                  return TRUE;
14                else
15                  var ← back();
16              else
17                var ← next_variable();
18      else
19        if no more values in Dom_var then
20            if var is the first universal variable then
21              return TRUE;
22            else
23              var ← back();
24        else
25          if each value in Dom_var is consistent then
26            val ← next_value(var);
27            remove val from Dom_var;
28            var ← next_variable();
29          else
30            var ← back();
31  end
```

Our variable ordering heuristics are implemented mostly in *set_up_d_std*(), *next_variable*() and *back*(). *next_variable*() selects a variable to be assigned via standard dependency schemes. It collects all variables x with empty $\overline{D_u^{std}(x)}$ first, which are called *candidates*, then chooses one by some heuristics and returns it. It also removes x from the $\overline{D_u^{std}}$ of all variables y with $x \in \overline{D_u^{std}(y)}$. When back jumping, *back*() puts back all unassigned variables x into $\overline{D_u^{std}(y)}$ with $y \in D^{std}(x)$. We do not consider value ordering heuristics in this paper, and *next_value*(x) returns a value in Dom_x arbitrarily.

In variable ordering heuristics *dom* and *deg* are important factors. For a variable x, *dom* is the size of the remaining domain of x and *deg* is the degree of x (the number of variables connected to x in constraint graph). First the heuristic *dom*, which selects the variable with minimal *dom*, was shown to be quite effective[17]. Then in [18] the authors improved on *dom* and proposed *dom+deg*, which selects variables as the heuristic *dom* does except that in case of a tie it chooses the one with

the highest *deg*. Later *dom/deg* was introduced in [8], which selects the variable with minimal ratio of *dom* to *deg*.

Inspired by [18], we introduce a new factor *dep* (from the word *dependence*) into variable ordering heuristics. For a variable x, *dep* is the size of $D^{std}(x)$. In our algorithm when x is assigned, for all variables y in $D^{std}(x)$, the size of $\overline{D_u^{std}(y)}$ is reduced. When $\overline{D_u^{std}(y)}$ is empty, y becomes a candidate. So the larger *dep* is, the higher probability that we get more candidates. In this paper we propose a heuristic *dom/deg+dep*, that is to select variables as the heuristic *dom/deg* does, in case of a tie choose the one with the highest *dep*. In this paper we only consider binary QCSPs. So for a variable x, *deg* is the number of constraints which involve x and another unassigned variable.

Algorithm 2. next_variable()

```
1   Function next_variable()
2   S ← ∅
3   for each x with D_u^std(x) = ∅ do
4       S ← S ∪ {x}
5   end
6   choose a variable var from S by heuristics
7   for each y with var ∈ D_u^std(y) do
8       remove var from D_u^std(y)
9   end
10  return var;
```

Algorithm 3. back()

```
1   Function back()
2   if no variable is returned by CBJ or SDP then
3       exit FALSE;
4   var ← the variable returned by CBJ or SDP;
5   for each variable x which is assigned after var do
6       restore all domains changed by x
7       unassign x
8       for each variable y ∈ D^std(x) do
9           D_u^std(y) ← D_u^std(y) ∪ {x}
10      end
11  end
12  return var;
```

4 Experimental Results

In this section we present experimental results on random QCSP instances generated by the generation model introduced in [4]. The generator takes seven parameters $\langle n, d, b, s, p, q_{\forall\exists}, q_{\exists\exists}\rangle$. n is the number of variables, d is the uniform domain size, b is the number of blocks, s is the uniform block size, p is the number of constraints as a fraction of all possible constraints, $q_{\exists\exists}$ is the number of tuples as a fraction of all possible tuples in constraints between existential variables, $q_{\forall\exists}$ is a parameter for constraints between a universal variable and an existential variable later, which is the number of tuples being rejected as a fraction of the number of tuples in a bijection. In the generator the last block is always existential. Our aim is to demonstrate the effect of variable ordering heuristics and standard dependency schemes in QCSP solvers. Therefore, we only give indicative results for the various heuristics and have not compared our work with other QCSP solvers.

We implemented a prototype QCSP solver, which has four variants. The first one is *baseSearcher* which uses no variable ordering heuristics and assigns variables by their order. The second variant is *blockSearcher* which uses the heuristic *dom/deg* within blocks. The third variant is *dstdSearcher* which uses the heuristic *dom/deg* and implements standard dependency schemes to help to make decision. The last variant is *dstdSearcher+* which also uses standard dependency schemes but the heuristic is *dom/deg+dep*. We compare these heuristics using random instances with a variety of parameter settings. All the results presented later are averages over 100 instances at each data point, and the value of $q_{\exists\exists}$ is varied in steps of 0.05. We compare all the four implementations and the results are given in Figs. 2∼4. The results confirm the better performance brought by variable ordering heuristics, standard dependency schemes and the new factor *dep*.

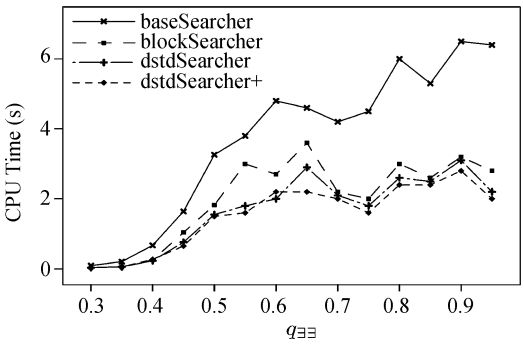


Fig.2. $n = 24$, $d = 9$, $b = 3$, $s = 8$, $p = 0.2$, $q_{\forall\exists} = 0.5$.

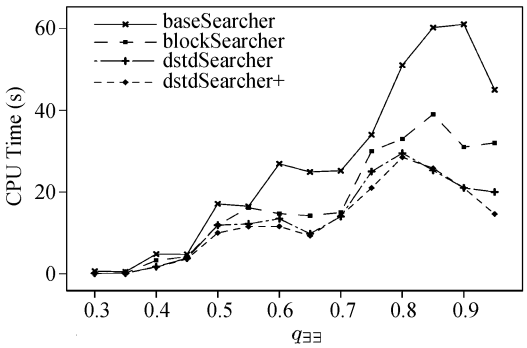


Fig.3. $n = 25$, $d = 8$, $b = 5$, $s = 5$, $p = 0.2$, $q_{\forall\exists} = 0.5$.

Our experimental results show that both true instances and false instances benefit from variable ordering heuristics. It is easy to understand that they are

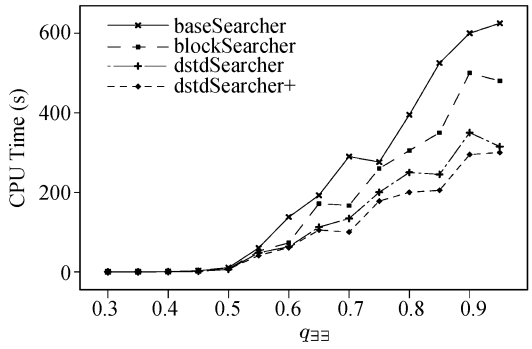


Fig.4. $n = 28$, $d = 8$, $b = 7$, $s = 4$, $p = 0.2$, $q_{\forall\exists} = 0.5$.

effective on false instances, and it needs a little explanation about their effect on true instances. When an instance is true, there are still a large number of conflicts during search, and finding these conflicts early, which is the aim of variable ordering heuristics, is also important.

5 Conclusions

In this paper we apply variable ordering heuristics and integrate standard dependency schemes in QCSP solvers, and enhance their efficiency. Standard dependency schemes are techniques borrowed from QBF. We show that they are useful in QCSP solving. We also introduce a new factor *dep* to variable ordering heuristics and propose a heuristic *dom/deg+dep*. The new heuristic is better than *dom/deg*. In this paper we are only interested in variable ordering heuristics and do not consider other techniques. One of our future work is to combine variable and value ordering heuristics. Also we will investigate more effective heuristics involving *dom, deg* and *dep*. Our solver in this paper is only a prototype. We will equip it with more advanced techniques and extend it to handle non-binary constraints in the future.

References

[1] Nightingale P. Consistency and the quantified constraint satisfaction problem [PhD thesis]. School of Computer Science, St Andrews University, 2007.

[2] Nightingale P. Non-binary quantified CSP: Algorithms and modelling. *Constraints*, 2009, 14(4): 539-581.

[3] Stynes D. Value ordering for offline and realtime-online solving of quantified constraint satisfaction problem [PhD thesis]. Department of Computer Science, National University of Ireland, 2009.

[4] Gent I P, Nightingale P, Stergiou K. QCSP-Solve: A solver for quantified constraint satisfaction problems. In *Proc. Int. Joint. Conf. Artificial Intelligence* (*IJCAI*), Edinburgh, UK, July 30-August 5, 2005, pp.138-143.

[5] Gent I P, Nightingale P, Rowley A, Stergiou K. Solving quantified constraint satisfaction problems. *Artif. Intell.*, 2008, 172(6-7): 738-771.

[6] Verger G, Bessiere C. Blocksolve: A bottom-up approach for solving quantified CSPs. In *Proc. CP*, Nantes, France, September 25-29, 2006, pp.635-649.

[7] Benedetti M, Lallouet A, Vautard J. Reusing CSP propagators for QCSPs. In *Proc. CSCLP*, Caparica, Portugal, June 26-28, 2006, pp.63-77.

[8] Bessière C, Régin J C. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Proc. CP*, Cambridge, USA, August 19-22, 1996, pp.61-75.

[9] Stynes D, Brown K N. Value ordering for quantified CSPs. *Constraints*, 2009, 14(1): 16-37.

[10] Bacchus F, Stergiou K. Solution directed backjumping for QCSP. In *Proc. CP*, Providence, USA, September 23-27, 2007, pp.148-163.

[11] Lonsing F, Biere A. Integrating dependency schemes in search-based QBF solvers. In *Proc. SAT*, Edinburgh, UK, July 11-14, 2010, pp.158-171.

[12] Bubeck U. Model-based transformations for quantified boolean formulas [Dissertations]. In *Artificial Intelligent*, Vol.329, Bibel W (eds.), IOS Press/Academische Verlags gesellschaft AKA, Amsterdam NL Heidelberg DE, 2010.

[13] Bubeck U, Kleine Büning H. Bounded universal expansion for preprocessing QBF. In *Proc. SAT*, Lisbon, Portugal, May 28-31, 2007, pp.244-257.

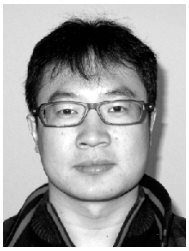
[14] Samer M. Variable dependencies of quantifed CSPs. In *Proc. Logic for Programming, Artificial Intelligence, and Reasoning*, Doha, Qatar, November 22-27, 2008, pp.512-527.

[15] Samer M, Szeider S. Backdoor sets of quantified boolean formulas. *J. Autom. Reasoning*, 2009, 42(1): 77-97.

[16] Rossi F, van Beek P, Walsh T. Handbook of Constraint Programming, Elsevier. 2006.

[17] Dechter R, Meiri I. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artif. Intell.*, 1994, 68(2): 211-241.

[18] Frost D, Dechter R. Look-ahead value ordering for constraint satisfaction problems. In *Proc. Int. Joint. Conf. Artificial Intelligence* (*IJCAI*), Montreal, Canada, August 20-25, 1995, pp.572-578.



Ji-Wei Jin is a postdoctoral researcher in the Institute of Software, Chinese Academy of Sciences (ISCAS). He obtained his Ph.D. degree from Sun Yat-sen University in 2010. His research interests include quantified Boolean formulae and quantified constraint satisfaction problems.

Fei-Fei Ma is an assistant research professor at the ISCAS. She obtained her Ph.D. degree from Chinese Academy of Sciences in 2010. Her research interests include automated reasoning and constraint solving.

Jian Zhang is a research professor at ISCAS. His research interests include automated reasoning, constraint solving, static program analysis and test data generation. He is a senior member of CCF, ACM, and IEEE.