Artificial Intelligence 243 (2017) 26–44

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

CCEHC: An efficient local search algorithm for weighted partial maximum satisfiability

Chuan Luo[a,b], Shaowei Cai[c,*], Kaile Su[d,e], Wenxuan Huang[f]

[a] *Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*
[b] *State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214125, China*
[c] *State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China*
[d] *College of Information Science and Technology, Jinan University, Guangzhou 510632, China*
[e] *Institute for Integrated and Intelligent Systems, Griffith University, Brisbane 4111, Australia*
[f] *Department of Material Science and Engineering, Massachusetts Institute of Technology, MA 02139, USA*

ARTICLE INFO

Article history:
Received 13 May 2015
Received in revised form 19 October 2016
Accepted 14 November 2016
Available online 17 November 2016

Keywords:
Local search
Weighted partial maximum satisfiability
Emphasis on hard clauses

ABSTRACT

Weighted maximum satisfiability and (unweighted) partial maximum satisfiability (PMS) are two significant generalizations of maximum satisfiability (MAX-SAT), and weighted partial maximum satisfiability (WPMS) is the combination of the two, with more important applications in practice. Recently, great breakthroughs have been made on stochastic local search (SLS) for weighted MAX-SAT and PMS, resulting in several state-of-the-art SLS algorithms *CCLS*, *Dist* and *DistUP*. However, compared to the great progress of SLS on weighted MAX-SAT and PMS, the performance of SLS on WPMS lags far behind. In this paper, we present a new SLS algorithm named *CCEHC* for WPMS. *CCEHC* employs an extended framework of *CCLS* with a heuristic emphasizing hard clauses, called EHC. With strong accents on hard clauses, EHC has three components: a variable selection mechanism focusing on configuration checking based only on hard clauses, a weighting scheme for hard clauses, and a biased random walk component. Extensive experiments demonstrate that *CCEHC* significantly outperforms its state-of-the-art SLS competitors. Further experimental results on comparing *CCEHC* with a state-of-the-art complete solver show the effectiveness of *CCEHC* on a number of application WPMS instances, and indicate that *CCEHC* might be beneficial in practice. Also, empirical analyses confirm the effectiveness of each component underlying the EHC heuristic.

1. Introduction

The maximum satisfiability (MAX-SAT) problem is the optimization version of the Boolean satisfiability (SAT) problem, which is a prototypical NP-complete problem and is of great importance in a variety of fields of computer science, mathematical logic and artificial intelligence. In the context of the SAT and MAX-SAT problems, a propositional formula F is usually expressed in conjunctive normal form (CNF) [1], i.e., $F = \bigwedge_i \bigvee_j l_{ij}$, where each l_{ij} is a literal, which is either a Boolean variable or its negation. A CNF formula can be expressed as a set of clauses, where a clause is a disjunction of literals, and each CNF formula is a conjunction of clauses.

* Corresponding author.
E-mail addresses: chuanluosaber@gmail.com (C. Luo), shaoweicai.cs@gmail.com (S. Cai), k.su@griffith.edu.au (K. Su), key01027@mit.edu (W. Huang).

http://dx.doi.org/10.1016/j.artint.2016.11.001
0004-3702/© 2016 Elsevier B.V. All rights reserved.

Given a propositional formula in conjunctive normal form (CNF), the SAT problem is to decide whether an assignment exists such that all clauses in this CNF formula are satisfied; the MAX-SAT problem is to seek out an assignment that maximizes the number of satisfied clauses in the CNF formula; the weighted MAX-SAT problem, where each clause is associated with a positive integer as its weight, is to find an assignment that maximizes the total weight of satisfied clauses, and is an important generalization of MAX-SAT; the (unweighted) partial maximum satisfiability (PMS) problem, where clauses are divided into hard ones and soft ones, is to find an assignment that satisfies all hard clauses and maximizes the number of satisfied soft clauses, and is also an important generalization of MAX-SAT. The weighted partial maximum satisfiability (WPMS) problem is the combination of both weighted MAX-SAT and PMS, and is a significant generalization of MAX-SAT: Given a CNF formula, the WPMS problem, where clauses are divided into hard ones and soft ones, and each soft clause is associated with a positive integer as its weight, is to seek out an assignment that satisfies all hard clauses and maximizes the total weight of satisfied soft clauses. In theory, MAX-SAT and its generalizations (i.e., weighted MAX-SAT, PMS and WPMS), are typically NP-hard problems, and it is well known that optimal solutions to these problems are hard to approximate [2]. Thus, it is very interesting to explore high-performance heuristic procedures to solve these hard problems. In this paper, our focus is on the WPMS problem.

In practice, as many combinatorial problems in real-world applications usually contain hard and soft constraints [3] and also soft constraints often have different priorities, encoding such real-world problems into the WPMS problem is more natural and direct than encoding them into SAT, MAX-SAT, weighted MAX-SAT or PMS. In fact, many important realistic problems in a wide range of real-world applications, such as computational protein design [4,5], set covering [6], coalition structure generation [7] and so on, can be encoded and solved as WPMS instances.

There are two popular categories of practical algorithms for solving MAX-SAT: complete algorithms and stochastic local search (SLS) algorithms. Complete algorithms are able to prove the optimality of the solution, but they may not return a good-quality solution for large-sized instances within reasonable time [8]. Complete algorithms can be classified into two main classes: branch and bound MAX-SAT algorithms [9–11] which are based on *DPLL* procedures [12,13], and SAT based algorithms [14–16] which successively call an efficient *CDCL* (*Conflict-Driven Clause Learning*) SAT solver [17,18]. Although SLS algorithms are typically incomplete, i.e., they do not guarantee the optimality of the solutions they find, SLS algorithms are often able to find good-quality solutions within a reasonable time frame [19,3]. SLS algorithms are usually evolving out of *GSAT* [20] and *WalkSAT* [21]. However, there is little work on SLS algorithms for solving WPMS, and almost all of the existing solvers for solving WPMS are complete ones.

Recently, significant breakthroughs have been achieved on SLS algorithms for solving weighted MAX-SAT and PMS, resulting in state-of-the-art SLS algorithms namely *CCLS* [22] and *Dist* [3] as well as *Dist*'s improvement *DistUP* [23]. The *CCLS* algorithm makes great progress in solving weighted MAX-SAT. *CCLS* won several categories in the incomplete solver track of the MAX-SAT Evaluations 2013 and 2014, thanks to the configuration checking strategy [24], which has been successfully applied to SAT [25] and minimum vertex cover [26]. The *CCLS* algorithm can be used to solve WPMS by translating WPMS into weighted MAX-SAT, as the translation can be very straightforward via setting the weight of each hard clause to the total weight of all soft clauses plus 1. However, when it comes to the WPMS problem, *CCLS* loses its power and shows ineffectiveness, as can be seen from the competition results of the incomplete solver track of the MAX-SAT Evaluation 2014.[1] The *Dist* algorithm shows great success on solving PMS, and won several categories in the incomplete solver track of the MAX-SAT Evaluation 2014, and also competes well with state-of-the-art complete algorithms on some classes of PMS application instances, such as advanced encryption standard and protein [3]. *Dist* can also be adapted to solve WPMS, and is indeed the current best SLS algorithm for solving WPMS, winning the random WPMS category and the crafted WPMS category in the incomplete solver track of the MAX-SAT Evaluation 2014. Particularly, the competition results of the MAX-SAT Evaluation 2014 show that *Dist* performs better than several state-of-the-art complete algorithms on the crafted WPMS benchmark. The *DistUP* algorithm, an improvement of *Dist* by using unit propagation as its initialization procedure, shows improvement over *Dist* on industrial instances. However, *CCLS*, *Dist* and *DistUP* are not dedicated to solving WPMS specifically, and their performance for solving WPMS could be further improved. Compared to the great progress of SLS algorithms on solving weighted MAX-SAT and PMS, the performance of SLS algorithms on solving WPMS lags far behind. This motivates us to design a more efficient SLS algorithm for solving WPMS. Inspired by the success of *Dist* as well as *DistUP*, our ambition is to solve more classes of structured problems and real-world ones.

In this work, we present a new SLS algorithm named *CCEHC* (*Configuration Checking with Emphasis on Hard Clauses*) for solving WPMS. Our *CCEHC* algorithm employs an extended framework of *CCLS* with a heuristic emphasizing hard clauses, called EHC. With strong focus on hard clauses, the EHC heuristic has three components: a variable selection mechanism focusing on a forbidding mechanism of configuration checking based only on hard clauses, a weighting scheme for hard clauses, and an approach of biased random walk. Our main contributions in this paper are summarized as follows.

Firstly, we identify an efficient algorithm framework for solving WPMS. It is surprising that our algorithm framework is based on *CCLS* instead of *Dist*, though *CCLS* shows worse performance on solving WPMS compared to *Dist*.

Secondly, we propose a new variable selection mechanism focusing on a new forbidding mechanism of configuration checking. This forbidding mechanism is similar to clause state based configuration checking in the context of SAT [27–29]. However, there is a major distinction that this configuration checking mechanism emphasizes hard clause, and a configu-

[1] http://www.maxsat.udl.cat/14/results-incomplete/index.html.

ration consists of the states of hard clause rather than all clauses. Also, to the best of our knowledge, this is the first time this kind of clause states based configuration checking is applied to MAX-SAT solving, while previous configuration checking techniques in the context of MAX-SAT are all based on neighboring variables [22,30]. We remark that, although this variable selection mechanism does not improve the performance of *CCEHC* on random WPMS instances, it makes contributions to the performance of *CCEHC* on a number of structured and real-world application WPMS instances. The related experiments, which confirms the effectiveness of this variable selection mechanism, and the detailed discussions can be found in Section 7.1.

Finally, by adopting a weighting scheme for hard clauses (which was first exploited in *Dist*) and adjusting the strategy of biased random walk for partial MAX-SAT, we integrate the two with our new configuration checking mechanism in a subtle way and obtain our new EHC heuristic. The three components underlying the heuristic EHC accentuates hard clauses and, as a whole, fits into our algorithm framework very well.

To evaluate the efficiency and the robustness of our *CCEHC* algorithm, we compare *CCEHC* against *CCLS*, *Dist* and *DistUP* on a broad range of WPMS benchmarks including the random, crafted and industrial benchmarks from the MAX-SAT Evaluation 2014 as well as four real-world application benchmarks. Experimental results clearly show that *CCEHC* generally performs much better than *CCLS*, *Dist* and *DistUP*, and thus establishes a new state of the art on SLS algorithms for solving WPMS. We also conduct more experiments to study the performance variability of SLS algorithms on selected instances.

Further, we conduct more empirical evaluations to analyze each underlying component in the EHC heuristic, and the related results confirm the effectiveness of these components in the EHC heuristic. Also, we compare the *CCEHC* algorithm with a state-of-the-art complete algorithm, i.e., *WPM-2014* [15,14] on these WPMS benchmarks. According to the experimental results, although *CCEHC* performs worse than *WPM-2014* on the industrial benchmark from the MAX-SAT Evaluation 2014 and an application benchmark, *CCEHC* is able to return much better-quality solutions than *WPM-2014* on other three important real-world application benchmarks as well as the random and the crafted benchmarks from the MAX-SAT Evaluation 2014. We also include the related experiments with a state-of-the-art lower-bound based complete solver *Eva* [16]. The experiments indicate that on a number of application instances, where *Eva* could not prove optimality, our *CCEHC* algorithm is able to return good-quality solutions fast, indicating that our *CCEHC* algorithm might be beneficial in practice. Additionally, we conduct empirical evaluations to analyze the efficiency of the combination of *CCEHC* and the unit propagation initialization [23] utilized by *DistUP*, and the related experiments demonstrate that the resulting solver *CCEHC+UP* gives a performance improvement over *CCEHC* on a large number of WPMS instances, which indicates that *CCEHC* could cooperate well with the unit propagation initialization.

The remainder of our paper is structured as follows. In Section 2, we provide some preliminary definitions and notations. Section 3 presents a brief review of the *CCLS* algorithm. In Section 4, we propose the EHC heuristic and introduce those components underlying the EHC heuristic. Section 5 presents the CCEHC algorithm and describes it in detail. In Section 6, extensive experiments on a wide range of WPMS benchmarks are conducted to present the efficiency of *CCEHC*. In Section 7, we empirically analyze the effectiveness of each component underlying the EHC heuristic, then list the main differences between *CCHEC* and *CCLS* as well as the major differences between *CCEHC* and *Dist*, and evaluate the efficiency of the combination of *CCEHC* and the unit propagation initialization. Section 8 concludes this paper and lists future work.

2. Preliminaries

Given a set of n Boolean *variables* $V = \{x_1, x_2, \cdots, x_n\}$ and the set of *literals* corresponding to these variables $L = \{x_1, \neg x_1, x_2, \neg x_2, \cdots, x_n, \neg x_n\}$, a *clause* is a disjunction of literals. A *formula* in conjunctive normal form (CNF) is a conjunction of clauses, and can be described as a set of clauses. Given a CNF formula F, we use $V(F)$ to denote the set of all variables in F. Two different variables are neighbors when they appear in at least one clause simultaneously, and we use the notation $N(x)$ to denote the set of all x's neighboring variables. A complete *assignment* is a mapping that assigns a Boolean value (either 0 or 1) to each variable in the formula. For SLS algorithms for solving WPMS (as well as MAX-SAT, weighted MAX-SAT and PMS), a candidate solution is a complete assignment. Given a CNF formula F and a complete assignment α corresponding to F, each clause in F under assignment α has two possible *states*: *satisfied* and *unsatisfied*; a clause c in F is satisfied if at least one literal in c is true under α; otherwise c is unsatisfied.

A weighted partial CNF formula is such a CNF formula, where all clauses are divided into hard ones and soft ones, and each soft clause c is associated with a positive integer $w(c)$ as its weight. Given a weighted partial CNF formula F, the **weighted partial maximum satisfiability (WPMS)** problem is to find such a complete assignment which satisfies all hard clauses in F and maximizes the total weight of all satisfied soft clauses in F.

Given a weighted partial CNF formula F (i.e., a WPMS instance), a complete assignment is *feasible* if it satisfies all hard clauses in F, and the cost of a feasible complete assignment α, denoted as $cost(\alpha)$, is the total weight of all unsatisfied soft clauses under α. The optimal assignment is the feasible complete assignment with the minimum cost value.

The basic framework of SLS algorithms for solving WPMS can be described as follows. In the initialization, the SLS algorithm randomly generates an assignment mapping to all Boolean variables; then the SLS algorithm repetitively selects and flips a Boolean variable until timeout or the number of search steps exceeds a given limit; finally, the SLS algorithm reports the feasible assignment with the lowest cost value encountered during the search process as the solution. In the search process, most SLS algorithms work between two modes: the greedy (intensification) mode and the diversification mode. In greedy mode, SLS algorithms prefer to flip those variables whose flips would decrease the number of unsatisfied hard clauses and the total weight of unsatisfied soft clauses. In diversification mode, SLS algorithms tend to diversify the search and thus explore the search space by using randomized strategies.

In SLS algorithms for solving WPMS, for a variable x, the *hard make score* of x, denoted by $hmake(x)$, is the number (or total weight if using clause weighting scheme) of unsatisfied hard clauses that would become satisfied if x is flipped; the *hard break score* of x, denoted by $hbreak(x)$, is the number (or total weight if using clause weighting scheme) of satisfied hard clauses that would become unsatisfied if x is flipped; the *hard score* of x, denoted by $hscore(x)$, is the increment in the number (or total weight if using clause weighting scheme) of satisfied hard clauses if x is flipped, and can be understood as $hscore(x) = hmake(x) - hbreak(x)$ [3]; the *soft make score* of x, denoted by $smake(x)$, is the total weight of unsatisfied soft clauses that would become satisfied if x is flipped; the *soft break score* of x, denoted by $sbreak(x)$, is the total weight of satisfied soft clauses that would become unsatisfied if x is flipped; the *soft score* of x, denoted by $sscore(x)$, is the increment in the total weight of satisfied soft clauses if x is flipped, and can be understood as $sscore(x) = smake(x) - sbreak(x)$ [3]. Most SLS algorithms do not separate hard clauses and soft clauses, and use the concepts of general *make score*, *break score* and *score* properties to select variables to be flipped: for a variable x, the *make score* of x, denoted by $make(x)$, can be calculated as $make(x) = A \times hmake(x) + smake(x)$; the *break score* of x, denoted by $break(x)$, can be calculated as $break(x) = A \times hbreak(x) + sbreak(x)$; the *score* of x, denoted by $score(x)$, can be calculated as $score(x) = A \times hscore(x) + sscore(x)$, and can also be seen as $score(x) = make(x) - break(x)$, where A in $make(x)$, $break(x)$ and $score(x)$ is a positive integer whose value is usually set to the total weight of all soft clauses plus 1. In our algorithm, when we use $score(x)$, we also adopt this definition and setting.

3. Reviewing the *CCLS* algorithm

In this section, we briefly review the *CCLS* algorithm [22], which serves as the basis of our proposed algorithm. The *CCLS* algorithm is a recent breakthrough in local search for solving MAX-SAT, and shows state-of-the-art performance on solving weighted MAX-SAT.

We would like to note that the *CCLS* algorithm is designed for solving weighted MAX-SAT. Thus, when *CCLS* is applied to solving WPMS, it first translates the WPMS instance into a weighted MAX-SAT instance by setting the weight of each hard clause to the total weight of all soft clauses plus 1 and then solves the translated weighted MAX-SAT instance. The pseudo-code of *CCLS* is outlined in Algorithm 1 and can also be found in the literature [22].

Algorithm 1: The *CCLS* algorithm.

```
   Input: Weighted CNF-formula $F$, $maxSteps$
1  generate a random assignment $\alpha$, $\alpha^* \leftarrow \alpha$;
2  for $step \leftarrow 1$ to $maxSteps$ do
3      if time limit is exceeded or all clauses are satisfied then break;
4      if with fixed probability $p$ then
5          $c \leftarrow$ a random unsatisfied clause;
6          $v \leftarrow$ a random variable in $c$;
7      else
8          if CCMPvars is not empty then
9              $v \leftarrow x$ with the greatest score in CCMPvars, breaking ties randomly;
10         else
11             $c \leftarrow$ a random unsatisfied clause;
12             $v \leftarrow$ a random variable in $c$;
13     $\alpha \leftarrow \alpha$ with $v$ flipped;
14     if $cost(\alpha) < cost(\alpha^*)$ then $\alpha^* \leftarrow \alpha$;
15 return $\alpha^*$;
```

In the initializing stage, the *CCLS* algorithm first generates a complete assignment α to all Boolean variables as the initial solution (line 1 in Algorithm 1). Then the *CCLS* algorithm starts the search process (i.e., executing a loop), where in each search step *CCLS* selects and flips a variable. During the search process, whenever a better solution is found by the *CCLS* algorithm, the best solution α^* is updated accordingly (line 14 in Algorithm 1).

In each search step, *CCLS* first checks whether the number of search steps exceeds the step limit *maxSteps*, time limit is exceeded or all clauses are satisfied: if one of these three terminating criterions is met, the search iteration terminates; otherwise, *CCLS* tries to select a flipping variable in this search step. With fixed probability p, *CCLS* works in diversification mode by employing the standard random walk component (lines 5–6 in Algorithm 1), i.e., it selects an unsatisfied clause c randomly and then picks a variable in c randomly; otherwise, *CCLS* switches to the greedy mode by activating the CCM (*Configuration Checking and Make*) heuristic (lines 8–12 in Algorithm 1) to select variables to be flipped. The CCM heuristic employs neighboring variables based configuration checking [24] to avoid local optima, and prefers to select the CCMP (*configuration changed and make positive*) variable with highest score, breaking ties randomly during the search [22]. A variable x is defined as CCMP if $make(x) > 0$ and, since x's last flip, at least one of x's neighboring variables has been flipped [22]. The notation *CCMPvars* is used to denote the set of all CCMP variables during the search. After the flipping variable is selected, the *CCLS* algorithm flips the selected variable and then starts the next search step (line 13 in Algorithm 1).

Finally, once the search process terminates, the *CCLS* algorithm reports the feasible solution with the least cost value.

4. The heuristic with emphasis on hard clauses

As can be clearly seen from Algorithm 1, the *CCLS* algorithm does not distinguish hard clauses and soft clauses. In our opinion, this is a disadvantage of *CCLS* when it is applied to solving WPMS, because hard clause is a very important feature of the WPMS problem. By treating hard clauses and soft clauses differently via different heuristics, we can make better use of this feature.

To improve the performance of *CCLS* on solving WPMS, we follow this direction and design a more sophisticated heuristic called EHC (*Emphasis on Hard Clauses*), which separates hard clauses and soft clauses and emphasizes hard clauses. The EHC heuristic is composed of three components: a variable selection heuristic using a configuration checking mechanism based only on hard clauses, a weighting scheme for hard clauses, and a strategy of biased random walk.

4.1. Variable selection via hard clauses' states based configuration checking

Inspired by the success of the clause states based configuration checking (CSCC) strategy [27–29] in SLS algorithms for SAT, we adapt this CSCC strategy to solving WPMS. In the WPMS problem, hard clauses are much more important than soft clauses, as hard clauses are forced to be satisfied in feasible solutions, so we propose a new forbidding strategy of configuration checking, which is based only on the states of hard clauses, named HCSCC (*Hard Clauses' States based Configuration Checking*). HCSCC concerns only hard clauses and is different from the original CSCC strategy [29] which concerns all clauses. Because the definition of configuration is the most important concept in the configuration checking strategy [24], we give the formal definition of configuration in our HCSCC strategy as follows.

Definition 1. Given a weighted partial CNF formula F and a complete assignment α to $V(F)$, the configuration of a variable $x \in V(F)$ for HCSCC is a vector *configuration*(x) consisting of the states of all hard clauses where x appears under assignment α.

For a variable x, a change on any element of *configuration*(x) is considered as a change on the whole *configuration*(x) vector. The HCSCC strategy is designed to prevent flipping the variable x whose *configuration*(x) has not been changed since x's last flip.

Similar to the approximate implementation of CSCC [29], in order to implement HCSCC more efficiently, we employ a Boolean array *hardConf* whose size is equal to the number of variables. The array *hardConf* is maintained according to the following rules.

- In the initializing stage, for each variable x, *hardConf*(x) is set to 1.
- Whenever a variable x is flipped, *hardConf*(x) is set to 0. Then each hard clause c, where x appears, is checked whether c's state is changed (from satisfied to unsatisfied or vice versa). If c's state is indeed changed, for each variable y $(y \neq x)$ in c, *hardConf*(y) is set to 1.

Thus, in the implementation of our HCSCC strategy, a variable x's configuration has been changed since x's last flip if *hardConf*$(x) = 1$.

Similar to the notion of CCD (*Configuration Changed and Decreasing*) variables [24] in SLS algorithms for solving SAT, we define the notion of HCSCCD (*Hard Clauses' States based Configuration Changed and Decreasing*) variables, which combines the HCSCC strategy with the *hscore* property. The formal definition of an HCSCCD variable is given as follows.

Definition 2. Given a partial weighted CNF formula F and a complete assignment α to $V(F)$, a variable $x \in V(F)$ is HCSCCD if *hardConf*$(x) = 1$ and *hscore*$(x) > 0$.

In this work, we use the notation *HCSCCDvars* to denote the set of all HCSCCD variables during the search. In the search process, our *CCEHC* algorithm prefers to select variables to be flipped from the *HCSCCDvars* set.

Here we prove an important lemma stating that, if a variable x is a HCSCCD variable, then x is a CCMP variable.

Lemma 1. *For a given variable x, if x is a HCSCCD variable, then x is a CCMP variable.*

Proof. For a variable x, x is a HCSCCD variable, meaning that *hscore*$(x) > 0$ and, since x's last flip, there exists a hard clause c where x appears such that c's state has been changed (from satisfied to unsatisfied or vice versa). On the other hand, to change c's state, there exists one variable y $(y \neq x)$ that has been flipped since x's last flip. As x and y appear in the same hard clause c, we derive $y \in N(x)$, meaning that since x's last flip, at least one of x's neighboring variables has been flipped.

Since $hscore(x) > 0$ and $hscore$ can be calculated as $hmake(x) - hbreak(x)$, we have $hmake(x) > hbreak(x)$. Since $hbreak(x)$ is a nonnegative integer according to its definition in Section 2, we have $hbreak(x) \geq 0$. Therefore, we can easily obtain $hmake(x) > 0$. Since $smake(x)$ is a nonnegative integer according to its definition in Section 2, we have $smake(x) \geq 0$. As $make(x) = A \times hmake(x) + smake(x)$, we can easily derive $make(x) > 0$.

As $make(x) > 0$ and since x's last flip, at least one of x's neighboring variables has been flipped, x is a CCMP variable. □

When to select HCSCCD variables: According to Lemma 1, it is easy to derive that the *HCSCCDvars* set is a subset of the *CCMPvars* set. Inspired by the success of the hierarchical combination of two different candidate variable sets in the context of SAT [31], this work adopts a hierarchical combination likewise: the *CCEHC* algorithm first tries to select a variable from the *HCSCCDvars* set with a higher priority if the *HCSCCDvars* set is not empty; otherwise, the algorithm selects a variable from the *CCMPvars* set.

Efficient implementation of searching HCSCCD variables: When using configuration checking in local search for SAT or MAX-SAT, the concept of CCD variables [24] is important, where a variable x is defined as CCD if $score(x) > 0$ and, since x's last flip, at least one of x's neighboring variables has been flipped [24]. The notation *CCDvars* denotes the set of all CCD variables during the search. In Lemma 2, we prove an important property – if a variable x is a HCSCCD variable, then x is a CCD variable. Thus, the *HCSCCDvars* set is a subset of the *CCDvars* set. The literature [22] proves that *CCDvars* is a subset of *CCMPvars*, so the key point in efficient implementation of searching HCSCCD variables is to search them in the *CCDvars* set rather than in the *CCMPvars* set.

Lemma 2. *For a given variable x, if x is a HCSCCD variable, then x is a CCD variable.*

Proof. For a variable x, x is a HCSCCD variable, meaning that $hscore(x) > 0$ and, since x's last flip, there exists a hard clause c where x appears such that c's state has been changed (from satisfied to unsatisfied or vice versa). On the other hand, to change c's state, there exists one variable y ($y \neq x$) that has been flipped since x's last flip. As x and y appear in the same hard clause c, we derive $y \in N(x)$, meaning that since x's last flip, at least one of x's neighboring variables has been flipped.

Since $hscore(x) > 0$ and $hscore(x)$ is a nonnegative integer according to its definition in Section 2, we can easily obtain $hscore(x) \geq 1$. Thus, we have $A \times hscore(x) \geq A$. As $sscore(x) = smake(x) - sbreak(x)$, since $smake(x)$ is a nonnegative integer according to its definition in Section 2, we can easily obtain $sscore(x) \geq -sbreak(x)$.

As $score(x) = A \times hscore(x) + sscore(x)$, thus $score(x) \geq A - sbreak(x)$. According to Section 2, A is a positive integer whose value equals the total weight of all soft clauses plus 1, and the value of $sbreak(x)$ is not larger than the total weight of all soft clauses. Thus, we can obtain $score(x) \geq 1$.

Because $score(x) > 0$ and since x's last flip, at least one of x's neighboring variables has been flipped, x is a CCD variable. □

4.2. Weighting scheme for hard clauses

In the context of SAT, clause weighting schemes serve as diversification mechanisms in SLS algorithms and have been used prominently and successfully in SLS algorithms for solving SAT [32–36]. This motivates us to further extend the *CCLS* algorithm framework with an effective clause weighting scheme.

In order to handle hard clauses with higher priority than soft clauses in the clause weighting scheme, it is natural to adopt a clause weighting scheme that only works for hard clauses. In this work, we utilize the one adopted by the *Dist* algorithm [3], which only adds weights to hard clauses. As stated in the literature [3], clause weighting for hard clauses helps to obtain feasible solutions and helps the algorithm to visit different satisfying assignments for hard clauses. The hard clause weighting scheme is similar to the PAWS scheme [35] and works as follows.

- In the beginning of the SLS algorithm, for each hard clause c, the weight of c (i.e., $w(c)$) is set to 1.
- When the hard clause weighting scheme is activated, with probability sp (sp is a real number and $0 \leq sp \leq 1$), for each satisfied hard clause c with $w(c) > 1$, $w(c)$ is decreased by 1; otherwise (with probability $1 - sp$), for each unsatisfied hard clause c, $w(c)$ is increased by 1.

Remark on the correctness of the hard clause weighting scheme: The correctness comes from two facts. (1) The final feasible solution reported by CCEHC must satisfy all hard clauses, as the algorithm verifies the solution before reporting it. (2) The hard clause weighting scheme only changes the weight of hard clauses and does not change the weight of soft clauses, so it would not influence the total weight of all unsatisfied soft clauses under the final feasible solution.

When to activate the hard clause weighting scheme: In this work, our *CCEHC* algorithm activates the hard clause weighting scheme immediately when there exists no HCSCCD variable during the search. We regard the phenomenon that no HCSCCD variable exists, as evidence that *CCEHC* has stagnated. If an SLS algorithm has stagnated, it is important to use a diversification mechanism. As the clause weighting scheme serves as an effective diversification strategy, it is reasonable to activate the hard clause weighting scheme when no HCSCCD variable exists.

Efficient implementation of the hard clause weighting scheme: According to the description, the hard clause weighting scheme consists of two phases: the weight-increasing phase and the weight-decreasing phase. The weight-decreasing phase

Algorithm 2: The *CCEHC* algorithm.

```
   Input: Weighted partial CNF-formula $F$, $maxSteps$
1  generate a random assignment $\alpha$, $\alpha^* \leftarrow \alpha$;
2  for $step \leftarrow 1$ to $maxSteps$ do
3      if time limit is exceeded or all hard and soft clauses are satisfied then break;
4      if with fixed probability $p$ then
5          if there exists any unsatisfied hard clause then $c \leftarrow$ a random unsatisfied hard clause;
6          else $c \leftarrow$ a random unsatisfied soft clause;
7          $v \leftarrow$ a variable $x$ with greatest $sscore(x)$ in $c$, breaking ties randomly;
8      else
9          if HCSCCDvars is not empty then
10             $v \leftarrow$ a variable randomly selected from HCSCCDvars (with bias towards the ones with the greatest hscore);
11         else
12             update hard clause weights;
13             if CCMPvars is not empty then
14                 $v \leftarrow x$ with the greatest $score(x)$ in CCMPvars, breaking ties randomly;
15             else
16                 $c \leftarrow$ a random unsatisfied clause (with bias towards the unsatisfied hard clauses);
17                 $v \leftarrow$ a random variable in $c$;
18     $\alpha \leftarrow \alpha$ with $v$ flipped;
19     if $cost(\alpha) < cost(\alpha^*)$ then $\alpha^* \leftarrow \alpha$;
20 if $\alpha^*$ satisfies all hard clauses then return $\alpha^*$;
21 else return "No feasible assignment is found";
```

concentrates on those hard satisfied clauses whose weights are greater than 1. For efficient implementation of the hard clause weighting scheme, we maintain a set of hard clauses whose weights are greater than 1. When the weight-decreasing phase is activated, the algorithm only checks the clauses in that set rather than checking all clauses, and thus saves the computation time.

4.3. Biased random walk

An important component of the *CCLS* algorithm is the random walk component, which is designed for the diversification mode. The random walk component (lines 5–6 in Algorithm 1) used in *CCLS* is a standard mechanism designed for SAT and MAX-SAT. However, the standard random walk component may not be suitable for SLS algorithms for WPMS, because it does not distinguish between hard and soft clauses. Since hard clauses are forced to be satisfied in feasible solutions of the WPMS problem, it is reasonable for us to employ a biased random walk component that prefers selecting a hard clause with a higher priority to choosing a soft clause. The biased random walk strategy is suggested by the literature [37] and described as follows.

When the biased random walk component is called, if there exist any unsatisfied hard clauses, the SLS algorithm first tries to choose an unsatisfied hard clause randomly; otherwise, an unsatisfied soft clause is selected randomly. Then the SLS algorithm employs a strategy to pick a variable in the chosen hard or soft clause. In this work, this is accomplished by selecting the variable x with the greatest $sscore(x)$ in the chosen hard or soft clause, inspired by the literature [3].

How to apply biased random walk in our SLS algorithm: In our *CCEHC* algorithm, we replace the standard random walk component (lines 5–6 in Algorithm 1) with the described biased random walk component, which is a natural way to apply the biased random walk component.

Efficient implementation of biased random walk: As the biased random walk component would pick a clause randomly from either the set of unsatisfied hard clauses or the set of unsatisfied soft clauses in each search step, thus a key point to efficiently implement biased random walk is to maintain two clause sets during the search process: the set of current unsatisfied hard clauses and the set of current unsatisfied soft clauses.

5. The *CCEHC* algorithm

On the basis of the framework of the *CCLS* algorithm and three components underlying the EHC heuristic described in Section 4, here we present a new SLS algorithm called *CCEHC* for solving WPMS. We outline the pseudo-code of the *CCEHC* algorithm in Algorithm 2 and describe it in detail as follows.

Initially, the *CCEHC* algorithm generates an assignment α uniformly at random as the initial solution. Then the *CCEHC* algorithm starts the search process, i.e., executing a loop. During the search process, whenever a better solution is found, the best solution α^* is updated accordingly (line 19 in Algorithm 2).

In each search step, *CCEHC* first checks whether the number of search steps exceeds the search step limit *maxSteps*, time limit is exceeded or all hard and soft clauses are satisfied: if one of these three terminating criterions is met, *CCEHC* terminates the search iteration; otherwise, *CCEHC* tries to select a flipping variable in this search step. With probability p, *CCEHC* calls the biased random walk component (lines 5–7 in Algorithm 2): if there exists any unsatisfied hard clause, an unsatisfied hard clause is selected randomly; otherwise, an unsatisfied soft clause is picked randomly; then *CCEHC* selects a variable x with greatest $sscore(x)$ in the chosen hard or soft clause as the variable to be flipped. With probability $1 - p$, *CCEHC* first checks whether the *HCSCCDvars* set is empty or not; if the *HCSCCDvars* set is not empty, *CCEHC* picks a variable randomly selected from *HCSCCDvars* (with bias towards the ones with the best *hscore*), inspired by the literature [3] (line 10 in Algorithm 2). Otherwise (the *HCSCCDvars* set is empty, meaning that the algorithm has stagnated), *CCEHC* updates hard clause weights according to the hard clause weighting scheme which is presented in Section 4.2 for diversification (line 12 in Algorithm 2), and then selects a variable according to the CCM-like heuristic (lines 13–17 in Algorithm 2): if the *CCMPvars* set is not empty, *CCEHC* picks the variable with the greatest *score* from *CCMPvars*, breaking ties randomly; otherwise (the *CCMPvars* is empty), *CCEHC* selects an unsatisfied clause randomly (if there exist unsatisfied hard clauses, then an unsatisfied hard clause is selected randomly; otherwise, an unsatisfied soft clause is selected randomly), and then *CCEHC* picks a variable randomly in the chosen hard or soft clause as the variable to be flipped. After the variable to be flipped is selected, the *CCEHC* algorithm flips the selected variable and then starts the next search step (line 18 in Algorithm 2).

Finally, once the search process terminates, the *CCEHC* algorithm checks the feasibility of the best solution α^*. If α^* is feasible (it satisfies all hard clauses), *CCEHC* reports α^* as the solution; otherwise (α^* is not feasible), *CCEHC* outputs “No feasible assignment is found”.

6. Experimental evaluations

In this section, we first introduce the benchmark instance sets, the competitors, and the experimental setup used in our experiments. Then, we report the experiments conducted on a broad range of WPMS benchmarks to evaluate the efficiency and the robustness of our *CCEHC* algorithm. Finally, we present additional empirical evaluations between *CCEHC* and a state-of-the-art complete solver *WPM-2014* on all testing WPMS benchmark. Notably, our experimental results demonstrate the effectiveness of *CCEHC* on a number of application instances, where *WPM-2014* as well as a state-of-the-art lower-bound based complete solver *Eva* could not return good-quality solutions within the cutoff time of 300 CPU seconds, indicating that *CCEHC* can be genuinely beneficial in practice.

We would like to note that in the crafted WPMS category of the incomplete solver track of the MAX-SAT Evaluation 2015,[2] *CCEHC* returns the best-quality solution on the largest number of WPMS instances, and more encouragingly finds the better-quality solution on more instances than an efficient complete solver *ILP-2015-in* [38], which uses the outputting format of incomplete solvers (i.e., printing the better-quality solution immediately once the solver finds one), while in that competition other SLS competitors were beaten by *ILP-2015-in*.

6.1. The benchmarks

We evaluate our *CCEHC* algorithm on a wide range of benchmarks including random, crafted and industrial WPMS benchmarks from the MAX-SAT Evaluation 2014 as well as four real-world application benchmarks. These testing benchmarks are described in detail below.

The first benchmark (MSE2014_WPMS_Random) is the random WPMS benchmark from the MAX-SAT Evaluation 2014,[3] which contains many random WPMS instances with a variety of clause-to-variable ratios.

The second benchmark (MSE2014_WPMS_Crafted) is the crafted WPMS benchmark from the MAX-SAT Evaluation 2014,[4] which contains many structured instances and includes many structured types such as those phase-transition ones generated according to Model RB, which have proven to be difficult both in theory [39] and in practice [40]. Solving these instances efficiently is considered to be a difficult task for SLS algorithms.

The third benchmark (MSE2014_WPMS_Industrial) is the industrial WPMS benchmark from the MAX-SAT Evaluation 2014,[5] which currently remains very difficult for SLS algorithms. On solving these industrial WPMS instances, SLS algorithms usually show poor performance. Thus, it is a challenge to improve the performance of SLS algorithms on this benchmark.

Moreover, in order to show that our *CCEHC* algorithm might be beneficial in practice, we consider four real-world application benchmarks which are encoded from real-world applications, including computational protein design[6] [4,5], advanced encryption standard[7] (AES) [41], the pedigree problem[8] [42] and cluster expansion[9] [43]. For the benchmark of computational protein design, all instances are encoded by Allouche et al. [5]. Computational protein design tries to intelligently guide the protein design process by producing a collection of proteins that is rich in functional proteins, and has been successfully applied to increase protein thermostability and solubility [5]. For the benchmark of advanced encryption standard,

[2] http://www.maxsat.udl.cat/15/results-incomplete/index.html#wpms-crafted.
[3] http://www.maxsat.udl.cat/14/benchmarks/wpms_random.tgz.
[4] http://www.maxsat.udl.cat/14/benchmarks/wpms_crafted.tgz.
[5] http://www.maxsat.udl.cat/14/benchmarks/wpms_industrial.tgz.
[6] http://genoweb.toulouse.inra.fr/~degivry/evalgm/CFN/ProteinDesign/.
[7] In the directory ‘aes/’, http://www.maxsat.udl.cat/14/benchmarks/pms_industrial.tgz.
[8] http://genoweb.toulouse.inra.fr/~degivry/evalgm/CFN/Pedigree/.
[9] http://www.ncic.ac.cn/acg/Projects/CCEHC/cluster_expansion.zip.

all instances in this benchmark are encoded by Gwynne et al. [41] using minimum box translation. These WPMS instances are translated from components of AES and small-scale variants. AES has been broadly applied in many fields, such as secure communication systems, high-performance database severs, digital video/audio recorders, RFID tags, smart cards [44] and FPGA-based designs [45,46]. For the pedigree benchmark, all instances are translated from the model of the weighted constraint satisfaction problem, and are encoded by Sanchez et al. [42]. The objective in the NP-complete pedigree problem is to decide whether a Mendelian error exists in a given pedigree [47]; this is becoming an increasingly important problem [42]. For the benchmark of cluster expansion, all instances are translated from the model of pseudo-Boolean optimization, and are encoded by Huang et al. [48]. Cluster expansion is used to solve exact ground states for generalized Ising model [49] with certain periodicity, and is physically related to the construction of ground state phase diagram for different alloys [50] or battery materials [51].

6.2. *The state-of-the-art SLS competitors*

Our *CCEHC* algorithm is compared against three state-of-the-art SLS algorithms, namely *CCLS* [22], *Dist* [3] and *DistUP* [23].

The *CCLS* algorithm is the basic framework of *CCEHC*, and is the current best SLS solver for weighted MAX-SAT instances and unweighted MAX-SAT instances. *CCLS* won several categories in the incomplete solver tracks of the MAX-SAT Evaluations 2013 and 2014. As reported in the literature [22], on many types of MAX-SAT instances, *CCLS* significantly outperforms an efficient SLS solver IRoTS [2] and finds much better-quality solution than two state-of-the-art complete solvers, akmaxsat_ls [52] and New WPM2 [14]. Also, according to the results of the MAX-SAT Evaluation 2013, *CCLS* performs much better than iraNovelty++ [53], optimax-it, SAT4Jms-ext-i [54] and SAT4Jms-int-i [54]. In our experiments, we use the version of *CCLS* which is submitted to the MAX-SAT Evaluation 2015.

The *Dist* algorithm is the current best SLS solver for PMS instances, and won several categories in the incomplete solver track of the MAX-SAT Evaluation 2014. Indeed, *Dist* is also the current best SLS solver for WPMS, and won the random WPMS category and the crafted WPMS category in the incomplete solver track of the MAX-SAT Evaluation 2014 by beating all SLS solvers and outperforming all complete solvers which participate in these categories such as *WPM-2014* [14,15] and *optimax*. As reported in the literature [3], *Dist* shows superiority over *CCLS* and an efficient SLS solver *TWO-LEVEL* [55] on a wide range of PMS benchmarks, and competes well with state-of-the-art complete solvers on some classes of PMS application instances. In our experiments, we adopt the latest version of *Dist* [23].

The *DistUP* algorithm equips *Dist* with an unit propagation initialization and is the most recent improvement of *Dist*. In our experiments, we adopt the version of *DistUP* presented in the literature [23].

6.3. *Experimental preliminaries*

Our *CCEHC* algorithm is implemented in the programming language C++ and compiled by the compiler g++ (version 4.6.3) with the option '-O2'. The source codes[10] of *CCEHC* and the detailed experimental results[11] of *CCEHC* are both available online. In this paper, all experimental evaluations are carried out on a cluster of workstations equipped with Intel Xeon E7-8830 2.13 GHz CPU, 24 MB L3 cache and 1.0 TB RAM under the operating system CentOS (version: 7.0.1406).

In our experiments, the evaluation methodology is the same as the one adopted in the incomplete solver track of the MAX-SAT Evaluation 2014, and is described as follows. Each solver performs one run on each instance. For each run, we record the best solution and utilize the *runsolver* tool (version 3.3.4) [56] to record the time for finding the best solution. For each solver on each instance class or benchmark, we report the number of instances where the solver finds the best solution among all solvers in the same table corresponding to the related experiment, denoted by '#win.', and the averaged time of doing so on such winning instances, denoted by 'time' (the unit is CPU second). The number of instances in each instance class or benchmark is indicated in the column '#inst.'. The cutoff time of each run is set to 300 CPU seconds, suggested by the rules of the incomplete solvers track of the MAX-SAT Evaluation 2014, unless we clearly specify the cutoff time in the tables which present the related experimental results. The rules of the MAX-SAT Evaluation 2014 establish that the winner is the solver which finds the best solution for the most instances among all competing solvers in the related experiment, breaking ties by preferring the solver with the least averaged time. The results in the **bold** font indicate the best performance for the related instance class or benchmark in the related experiment.

Additionally, in order to study the complementarity of the competing solvers, in each experiment, we also report the results of the Virtual Best Solver (*VBS*), i.e., the perfect selector – on each instance, the solution of *VBS* is the best one of the solutions reported by all competing solvers included in this experiment on solving this instance; if more than one competing solver report the best-quality solution, the computing time of *VBS* on this instance is the shortest one. We would like to note that *VBS* is not an actual solver, and it gives an upper bound on the performance of an actual per-instance selector of all competing solvers in an experiment.

[10] http://www.ncic.ac.cn/acg/Projects/CCEHC/CCEHC_source_codes.zip.

[11] http://www.ncic.ac.cn/acg/Projects/CCEHC/CCEHC_detailed_experimental_results.zip.

Table 1
The parameter settings reported by *SMAC* for *CCEHC*, *Dist* and *CCLS* on random WPMS instances, crafted WPMS instances, as well as industrial and application WPMS instances.

Instance type	*CCEHC*		*Dist*			*CCLS*
	p	*sp*	*wp*	*sp*	*t*	*p*
Random WPMS	0.2	0.0001	0.1	0.001	15	0.535
Crafted WPMS	0.177	0.003	0.1	0.001	15	0.204
Industrial and application WPMS	0.279	0.085	0.038	0.002	6	0.2

6.4. Parameter tuning

It is well acknowledged that parameter settings are important to the empirical behavior for many high-performance algorithms, especially heuristic algorithms for solving computationally hard problems [57]. At present, general-purpose automatic configuration tools [57,58] have been applied to tuning the parameter settings for practical algorithms and have shown their power on solving SAT and mixed integer programming.

In this work, in order to conduct a fair performance comparison, we utilize a powerful automatic configuration tool called *SMAC* (*Sequential Model-based Algorithm Configuration*, version: 2.10.03) [58] to tune the parameter settings for our SLS algorithm *CCEHC* and all the SLS competitors.

The *CCEHC* algorithm has 2 parameters to be tuned: p (which is a real number and controls the balance between the greedy mode and the diversification mode) and *sp* (which is a real number and controls the probability of smoothing hard clause weights); the *Dist* algorithm has 3 parameters to be tuned: *wp* (which is a real number and controls the probability of activating the random walk heuristic), *sp* (which is a real number and controls the probability of smoothing hard clause weights) and t (which is an integer and controls the cardinality of the candidate variable set applied in the 'Best from Multiple Selections' strategy); the *CCLS* algorithm has 1 parameter to be tuned: p (which is a real number and controls the balance between the greedy mode and the diversification mode). For *CCEHC*, the parameter domain of real number p is $[0, 0.5]$, and the default value of p is 0.2; the parameter domain of real number *sp* is $[0, 0.1]$, and the default of *sp* is 0.001 for solving industrial and application instances and 0.0001 for solving random and crafted WPMS instances. For *Dist*, the parameter domain of real number *wp* is $[0, 1]$, and the default value of *wp* is 0.1; the parameter domain of real number *sp* is $[0, 1]$, and the default value of *sp* is 0.00001 for solving industrial and application WPMS instances and 0.001 for random and crafted WPMS instances; the parameter domain of integer t is $[1, 100]$, and the default value of t is 40 for industrial and application WPMS instances and 15 for random and crafted WPMS instances. For *CCLS*, the parameter domain of real number p is $[0, 1]$, and the default value of p is 0.2.

In the parameter tuning procedure, we use the whole MSE2014_WPMS_Random benchmark, the whole MSE2014_WPMS_Crafted benchmark as well as the whole MSE2014_WPMS_Industrial benchmark as the training sets for tuning the parameter settings for each solver on solving random WPMS instances, crafted WPMS instances as well as industrial and application WPMS instances, respectively. The configuration objective for *SMAC* is the solution-quality (i.e., minimizing the weight of unsatisfied soft clauses). The cutoff time of each run for *SMAC* is 300 CPU seconds, and the time budget of all runs for *SMAC* is 172800 CPU seconds (i.e., 2 days).

The parameter settings found by *SMAC* for *CCEHC*, *Dist* and *CCLS* are summarized in Table 1, and we use these parameter settings in the following experimental evaluations. As the only difference between *DistUP* and *Dist* is the initialization procedure, in our experiments *DistUP* uses the same parameter settings as *Dist* does.

6.5. Experimental results with SLS competitors

In this subsection, we conduct extensive experiments of our *CCEHC* algorithm and its state-of-the-art SLS competitors, i.e., *DistUP*, *Dist* and *CCLS* on all testing benchmarks, i.e., the MSE2014_WPMS_Random, MSE2014_WPMS_Crafted and MSE2014_WPMS_Industrial benchmarks and four real-world application WPMS benchmarks.

6.5.1. Experiments on the MSE2014_WPMS_Random benchmark

Table 2 presents the comparative results of our *CCEHC* algorithm and its state-of-the-art SLS competitors *DistUP*, *Dist* and *CCLS* on the MSE2014_WPMS_Random benchmark. On this benchmark, *CCEHC* provides a performance advantage over *CCLS* in terms of solution-quality. On the overall performance of this benchmark, although *CCEHC* is slower than both *DistUP* and *Dist* in terms of averaged time, *CCEHC*, *DistUP* and *Dist* show the same performance in terms of solution-quality (they all find the best solution for 279 instances). The results of *VBS* over all random WPMS instances show that these competing SLS solvers are complementary in terms of averaged time, and indicate that it is promising to construct an algorithm-selector consisting of these competing SLS solvers.

6.5.2. Experiments on the MSE2014_WPMS_Crafted benchmark

The comparative results of *CCEHC* and its state-of-the-art SLS competitors *DistUP*, *Dist* and *CCLS* on the MSE2014_WPMS_Crafted benchmark are illustrated in Table 3. According to Table 3, it is clear that our *CCEHC* algorithm stands out as the best solver on this benchmark. From the experimental results, on all 310 instances, *CCEHC* finds the best solution for 282

Table 2
Experimental results of *CCEHC*, *DistUP*, *Dist* and *CCLS* on the MSE2014_WPMS_Random benchmark. The run time is measured in CPU second.

Instance class	#inst.	*CCEHC*		*DistUP*		*Dist*		*CCLS*		*VBS*	
		#win.	time	#win.	time	#win.	time	#win.	time	#win.	time
wmax2sat/100v	40	**40**	0.05	**40**	**<0.01**	**40**	0.04	**40**	**<0.01**	40	<0.01
wmax2sat/120v	40	**40**	**<0.01**	**40**	**<0.01**	**40**	**<0.01**	**40**	**<0.01**	40	<0.01
wmax2sat/140v	40	**40**	0.06	**40**	**<0.01**	**40**	0.01	**40**	0.02	40	<0.01
wmax3sat/hi	40	**40**	0.34	**40**	0.28	**40**	0.31	**40**	**<0.01**	40	<0.01
wpmax2sat/hi	30	**29**	0.09	**29**	**<0.01**	**29**	0.06	26	4.76	29	<0.01
wpmax2sat/lo	30	**30**	0.10	**30**	**<0.01**	**30**	0.02	26	46.54	30	<0.01
wpmax2sat/me	30	**30**	0.13	**30**	**<0.01**	**30**	0.02	29	27.97	30	<0.01
wpmax3sat/hi	30	**30**	0.03	**30**	**<0.01**	**30**	0.01	**30**	0.13	30	<0.01
Overall	280	**279**	0.10	**279**	**0.04**	**279**	0.06	271	7.93	279	<0.01

Table 3
Experimental results of *CCEHC*, *DistUP*, *Dist* and *CCLS* on the MSE2014_WPMS_Crafted benchmark. The run time is measured in CPU second.

Instance class	#inst.	*CCEHC*		*DistUP*		*Dist*		*CCLS*		*VBS*	
		#win.	time	#win.	time	#win.	time	#win.	time	#win.	time
auctions/auc-paths	20	**20**	0.59	**20**	1.18	**20**	0.82	**20**	**0.05**	20	0.02
auctions/auc-scheduling	20	**20**	**<0.01**	**20**	**<0.01**	**20**	**<0.01**	**20**	**<0.01**	20	<0.01
CSG	10	**6**	74.49	4	3.51	0	0	0	0	10	46.10
frb	34	**34**	6.70	33	24.28	31	9.11	33	3.90	34	4.52
min-enc/planning	30	23	20.78	**29**	6.40	**29**	**2.27**	0	0	29	1.28
min-enc/planning_old	26	**26**	3.22	**26**	**0.19**	**26**	0.40	7	10.09	26	0.03
min-enc/warehouses	18	**15**	90.16	1	<0.01	4	25.26	1	<0.01	18	80.74
pseudo/miplib	12	2	0.14	3	3.39	**4**	66.97	1	<0.01	5	55.57
ramsey	15	12	6.13	12	20.60	12	20.67	**14**	34.57	15	36.16
random-net	32	**32**	146.25	0	0	0	0	0	0	32	146.25
set-covering/scp4x	10	**10**	173.42	0	0	0	0	1	142.49	10	173.42
set-covering/scp5x	10	**10**	141.42	0	0	0	0	0	0	10	141.42
set-covering/scp6x	5	**5**	91.66	0	0	0	0	0	0	5	91.66
set-covering/scpn	20	**20**	68.95	0	0	0	0	2	67.07	20	68.95
wmaxcut/dimacs_mod	43	**43**	2.65	42	<0.01	42	0.03	42	<0.01	43	<0.01
wmaxcut/spinglass	5	4	61.22	4	9.72	4	9.60	**5**	34.20	5	34.20
Overall	310	**282**	45.03	194	6.83	192	5.37	146	7.75	302	42.26

Table 4
Experimental results of *CCEHC*, *DistUP*, *Dist* and *CCLS* on the MSE2014_WPMS_Industrial benchmark and four real-world application WPMS benchmarks. The run time is measured in CPU second.

Benchmark	#inst.	*CCEHC*		*DistUP*		*Dist*		*CCLS*		*VBS*	
		#win.	time	#win.	time	#win.	time	#win.	time	#win.	time
MSE2014_WPMS_Industrial	410	**226**	124.43	154	96.53	90	95.50	35	39.59	389	125.15
Computational protein design	20	**11**	94.85	2	174.88	2	31.73	3	162.02	15	93.70
Advanced encryption standard	7	**5**	11.08	1	0.00	2	150.74	**5**	**2.31**	7	47.97
Pedigree	20	**11**	94.85	2	174.88	2	31.73	3	162.02	15	93.70
Cluster expansion	6	**6**	0.05	**6**	**<0.01**	**6**	0.01	3	<0.01	6	<0.01

of them, while this figure for *DistUP*, *Dist* and *CCLS* is only 194, 192 and 146, respectively. In detail, on all 16 instance classes, *CCEHC* gives the best performance on 12 of them, while this figure for *DistUP*, *Dist* and *CCLS* is only 4, 5 and 4, respectively. *Dist* won the crafted WPMS category in the incomplete solver track of the MAX-SAT Evaluation 2014 and *DistUP* also exhibits good performance on this benchmark, so it is challenging to improve such performance over *DistUP* and *Dist* on the MSE2014_WPMS_Crafted benchmark. The experimental results show that our *CCEHC* algorithm achieves the state-of-the-art performance on crafted WPM instances.

Then we focus on the investigation of the complementarity of all competing SLS solvers on this benchmark. The results of *VBS* show that these competing SLS solvers are complementary on 4 instance classes (i.e., ‘CSG’, ‘min-enc/warehouses’, ‘pseudo/miplib’ and ‘ramsey’) in terms of the solution quality, and indicate that these solvers might be complementary to each other. The experimental results also indicate that it is promising to build an algorithm-selector consisting of all competing SLS solvers on these 4 instance classes.

6.5.3. Experiments on the MSE2014_WPMS_Industrial benchmark and four real-world application WPMS benchmarks

The comparative results of *CCEHC* and its state-of-the-art SLS competitors *DistUP*, *Dist* and *CCLS* on the MSE2014_WPMS_Industrial benchmark and four real-world application WPMS benchmarks are summarized in Table 4. According to the em-

Table 5
Experimental results of *CCEHC*, *DistUP*, *Dist* and *CCLS* with 10 runs on selected random WPMS instances. The run time is measured in CPU second.

Instance	*CCEHC*		*DistUP*		*Dist*		*CCLS*		*VBS*	
	min. max. avg.	time c.v.	min. max. avg.	time c.v.	min. max. avg.	time c.v.	min. max. avg.	time c.v.	min. max. avg.	time c.v.
file_rwms_wcnf_L3_V70_C1000_0.wcnf	**198** **198** **198.0**	0.10 219.84%	**198** **198** **198.0**	**<0.01** 0%	**198** **198** **198.0**	**<0.01** 0%	**198** **198** **198.0**	**<0.01** 0%	198 198 198.0	<0.01 0%
file_rwpms_wcnf_L2_V150_C5000_ H150_0.wcnf	**5362** **5362** **5362.0**	0.04 178.43%	**5362** **5362** **5362.0**	**<0.01** 0%	**5362** **5362** **5362.0**	**<0.01** 0%	**5362** **5362** **5362.0**	1.53 115.58%	5362 5362 5362.0	<0.01 0%
file_rwpms_wcnf_L3_V100_C800_ H100_0.wcnf	**95** **95** **95.0**	0.01 300.00%	**95** **95** **95.0**	**<0.01** 0%	**95** **95** **95.0**	**<0.01** 0%	**95** **95** **95.0**	3.32 102.79%	95 95 95.0	<0.01 0%
s2v140c1600_1.wcnf	**1143** **1143** **1143.0**	**<0.01** 0%	**1143** **1143** **1143.0**	**<0.01** 0%	**1143** **1143** **1143.0**	**<0.01** 0%	**1143** **1143** **1143.0**	0.08 98.44%	1143 1143 1143.0	<0.01 0%

pirical results presented in Table 4, although *CCEHC* performs slightly slower than *CCLS* on the benchmark of advanced encryption standard and performs slightly slower than *DistUP* and *Dist* on the benchmark of cluster expansion, it is very apparent that, among these four competing SLS algorithms, our *CCEHC* algorithm performs best on the MSE2014_WPMS_Industrial benchmark and two real-world application WPMS benchmarks (i.e., 'Computational Protein Design' and 'Pedigree'). Particularly, the related experimental results indicate that our *CCEHC* algorithm is able to return better-quality solutions compared to *CCLS*, *Dist* and *DistUP* on a large number of real-world application instances. In fact, SLS algorithms are considered to be ineffective on solving industrial WPMS instances and real-world application instances, so it is very difficult to improve such performance of SLS algorithms on these benchmarks.

The results of *VBS* show that these competing SLS solvers are complementary on 4 benchmarks (i.e., 'MSE2014_WPMS_Industrial', 'Computational Protein Design', 'Advanced Encryption Standard' and 'Pedigree') in terms of the solution quality, and indicate the complementarity among these competing solvers. This also indicates that building an algorithm selector consisting of all competing SLS solvers could give performance advantage on these 4 benchmarks.

6.5.4. Experiments on evaluating performance variability of SLS solvers

SLS solvers are randomized, and their performance (e.g., solution quality found after a fixed amount of running time) can vary substantially over multiple independent runs on the same problem instance. We conduct empirical evaluations to study the performance variability of SLS solvers. We first select 4 random WPMS instances, 11 crafted WPMS instances and 6 industrial and application WPMS instances as the testing instances. Then all SLS solvers (*CCEHC*, *CCLS*, *Dist* and *DistUP*) are run 10 times (the cutoff time of each run is set to 300 CPU seconds) on each selected instance.

The detailed experimental results on random instances, crafted instances as well as industrial and application instances are reported in Tables 5, 6 and 7, respectively. In Tables 5, 6 and 7, for each SLS solver, 'min.' and 'max.' denote the minimum and maximum weight of unsatisfied soft clauses among all 10 runs, respectively, while 'avg.' denotes the averaged weight of unsatisfied soft clauses over all 10 runs; 'time' denotes the averaged consuming time over all 10 runs; 'c.v.' denotes the coefficient of variance (i.e., the standard deviation divided by the average value) in terms of consuming time over all 10 runs. According to Tables 5, 6 and 7, in terms of the minimum unsatisfied weight among 10 runs, *CCEHC* gives the best performance on 4 random WPMS instances, 11 crafted WPMS instances and 5 industrial and application instances; those of *DistUP* are 4, 6 and 2, respectively; those of *Dist* are 4, 6 and 1, respectively; those of *CCLS* are 4, 5 and 1, respectively. In terms of the averaged unsatisfied weight over 10 runs, *CCEHC* gives the best performance on 4 random WPMS instances, 10 crafted WPMS instances and 5 industrial and application WPMS instances; those of *DistUP* are 4, 4, and 2, respectively; those of *Dist* are 4, 4 and 1, respectively; those of *CCLS* are 4, 4 and 1, respectively. The experiments show that *CCEHC* stands out as the best solver in this comparison. According to the experimental results presented in Tables 5, 6 and 7, the metric of coefficient of variance in terms of consuming time, denoted by notation 'c.v', witnesses that the variability of consuming time of *CCEHC* on these instances is significant.

6.6. Comparing CCEHC *against state-of-the-art complete solvers*

In this subsection, we conduct more empirical analyses to evaluate our *CCEHC* algorithm against a state-of-the-art complete solver *WPM-2014* [14,15]. In this experiment, we adopt the version of *WPM-2014* which uses the outputting format of incomplete solvers (i.e., printing the better-quality solution immediately once the solver finds one). According to the competition results of the MAX-SAT Evaluation 2014, this version of *WPM-2014* won the industrial WPMS category in the incomplete solver track of the MAX-SAT Evaluation 2014. Additionally, we also report the experimental results for *Eva* [16], which won the industrial WPMS category in the complete solver track of the MAX-SAT Evaluation 2014. As the *Eva* solver

Table 6
Experimental results of *CCEHC*, *DistUP*, *Dist* and *CCLS* with 10 runs on selected crafted WPMS instances. As *CCLS* could not find feasible solutions in all 10 runs on solving instance 'driverlog02bc.wcsp.wcnf', so we mark 'N/A' for *CCLS* on solving instance 'driverlog02bc.wcsp.wcnf'. The run time is measured in CPU second.

Instance	*CCEHC*		*DistUP*		*Dist*		*CCLS*		*VBS*	
	min. max. avg.	time c.v.	min. max. avg.	time c.v.	min. max. avg.	time c.v.	min. max. avg.	time c.v.	min. max. avg.	time c.v.
cat_paths_60_170_ 0000.txt.wcnf	**122170**	16.17	**122170**	1.67	**122170**	1.79	**122170**	**0.69**	122170	0.18
	122170	80.44%	**122170**	100.75%	**122170**	85.80%	**122170**	150.44%	122170	181.03%
	122170.0		**122170.0**		**122170.0**		**122170.0**		122170.0	
cat_sched_60_200_ 0000.txt.wcnf	**242832**	**<0.01**	**242832**	**<0.01**	**242832**	**<0.01**	**242832**	**<0.01**	242832	<0.01
	242832	0%	**242832**	0%	**242832**	0%	**242832**	0%	242832	0%
	242832.0		**242832.0**		**242832.0**		**242832.0**		242832.0	
driverlog02bc. wcsp.wcnf	**2085**	124.26	**2085**	10.39	**2085**	**3.21**	N/A	N/A	2085	2.39
	2505	60.40%	**2085**	136.65%	**2085**	130.98%	N/A	N/A	2085	170.92%
	2146.8		**2085.0**		**2085.0**		N/A		2085.0	
frb40-19-1.wcnf	**720**	**21.10**	**720**	75.43	**720**	82.17	**720**	24.12	720	9.80
	720	118.01%	721	88.62%	721	80.15%	**720**	134.31%	720	104.00%
	720.0		720.4		720.6		**720.0**		720.0	
normalized-mps- v2-20-10-p0033. opb.msat.wcnf	**3089**	1.60	**3089**	2.70	**3089**	**1.06**	**3089**	169.20	3089	0.50
	3089	96.30%	**3089**	199.99%	**3089**	237.68%	3343	60.36%	3089	294.03%
	3089.0		**3089.0**		**3089.0**		3177.9		3089.0	
scp41_weighted. wcnf	**433**	141.53	488	165.32	489	169.36	441	197.14	433	141.53
	439	55.27%	507	44.60%	512	58.00%	484	41.77%	439	55.27%
	435.8		499.5		501.2		466.7			435.8
scp51_weighted. wcnf	**265**	170.57	336	101.75	341	182.38	272	145.59	265	170.57
	269	45.34%	374	60.78%	363	27.43%	302	63.24%	269	45.34%
	267.3		356.8		355.7		289.2		267.3	
scp61_weighted. wcnf	**138**	109.25	185	142.60	172	146.88	143	142.39	138	109.25
	141	53.19%	197	58.64%	205	43.63%	145	58.69%	141	53.19%
	138.5		191.0		193.4		143.5		138.5	
scpnre1_weighted. wcnf	**29**	13.79	147	137.81	39	0.79	31	118.40	29	13.79
	29	146.44%	188	47.81%	52	0.38%	35	83.82%	29	146.44%
	29.0		164.2		45.8		32.8		29.0	
scpnrf1_weighted. wcnf	**14**	3.04	54	141.20	17	1.59	14	104.00	14	3.04
	14	43.06%	70	64.77%	23	0.19%	15	91.72%	14	43.06%
	14.0		62.5		18.6		14.2		14.0	
t6g3-8888.spn.wcnf	**7844119**	89.52	**7844119**	173.48	**7844119**	122.96	**7844119**	**16.57**	7844119	16.39
	7844119	85.71%	7942134	49.41%	7952319	66.04%	**7844119**	122.73%	7844119	124.67%
	7844119.0		7877106.8		7908542.9		**7844119.0**		7844119.0	

performs a lower bound based core-guided search, it only finds one feasible solution finally when it proves optimality. Thus, it is unfair to directly compare *CCEHC* with *Eva*, so, as a reference, here we report the experimental results for *Eva* just to indicate the performance of the current state-of-the-art complete solver *Eva* on these benchmarks. The related experimental results are reported in Table 8.

According to Table 8, although *CCEHC* performs worse than *WPM-2014* on the MSE2014_WPMS_Industrial benchmark and the pedigree benchmark, *CCEHC* significantly outperforms the complete solver *WPM-2014* on the MSE2014_WPMS_Random benchmark, the MSE2014_WPMS_Crafted benchmark and three real-world application benchmarks ('Computational Protein Design', 'Advanced Encryption Standard' and 'Cluster Expansion'). The *WPM-2014* solver is a SAT-based complete MAX-SAT solver, which successively calls an efficient SAT solver. As this SAT-based technique is efficient in solving industrial problems and all state-of-the-art MAX-SAT algorithms that perform well on industrial instances are based on this technique [15], it is not surprising that *WPM-2014* performs better than *CCEHC* on the MSE2014_WPMS_Industrial benchmark and the pedigree benchmark. However, the results of *VBS* present that *CCEHC* are complementary to *WPM-2014* on the MSE2014_WPMS_Industrial benchmark and the pedigree benchmark, which indicates that the combination of SLS algorithms and complete algorithms could achieve better performance on solving industrial WPMS instances.

More encouragingly, the results of *VBS*, on the MSE2014_WPMS_Crafted benchmark, the MSE2014_WPMS_Industrial benchmark and three real-world application benchmarks of computational protein design, advanced encryption standard and pedigree, show that our *CCEHC* algorithm is complementary to *WPM-2014*, and indicate that it is promising to build an algorithm-selector consisting of *CCEHC* and *WPM-2014* to achieve better performance.

Table 7
Experimental results of *CCEHC*, *DistUP*, *Dist* and *CCLS* with 10 runs on selected industrial and application WPMS instances. As *CCLS* could not find feasible solutions in all 10 runs on solving instances 'berrichon1_direct.wcnf', 'langlade3_direct.wcnf' and 'WCNF_pathways_p05.wcnf', so we mark 'N/A' for *CCLS* on solving these instances. The run time is measured in CPU second.

Instance	*CCEHC*		*DistUP*		*Dist*		*CCLS*		*VBS*	
	min. max. avg.	time c.v.	min. max. avg.	time c.v.	min. max. avg.	time c.v.	min. max. avg.	time c.v.	min. max. avg.	time c.v.
507.wcsp.log.wcnf	**27391** **27392** **27391.3**	102.79 95.87%	27396 27398 27396.9	141.46 69.97%	27396 27397 27396.5	68.96 86.73%	27395 27397 27396.2	123.70 75.85%	27391 27392 27391.3	102.79 95.87%
berrichon1_direct.wcnf	**29** **31** **29.8**	140.21 51.44%	281 348 307.9	90.13 81.97%	720 844 774.4	247.36 17.72%	N/A N/A N/A	N/A N/A	29 31 29.8	140.21 51.44%
langlade3_direct.wcnf	**42** **45** **44.3**	136.19 45.75%	63 71 66.2	182.16 51.43%	72 88 78.3	197.41 36.03%	N/A N/A N/A	N/A N/A	42 45 44.3	136.20 45.75%
ped2.B.recomb1-0.01-1.wcnf	**7** **7** **7.0**	26.91 2.64%	**7** **7** **7.0**	**4.63** 34.96%	**7** **7** **7.0**	14.94 11.49%	**7** **7** **7.0**	78.30 47.07%	7 7 7.0	4.63 34.96%
random-net-80-2_network-1.net.wcnf	**69537** **70510** **69998.9**	112.73 71.40%	74843 76741 75672.8	128.12 66.25%	74565 76495 75693.9	164.84 49.31%	71077 72828 72193.8	151.08 60.54%	69537 70510 69998.9	112.73 71.40%
WCNF_pathways_p05.wcnf	78 85 82.9	70.36 102.28%	**75** **80** **77.6**	42.80 83.05%	82 132 103.7	135.84 78.86%	N/A N/A N/A	N/A N/A	75 80 77.6	41.13 85.94%

Table 8
Experimental results of *CCEHC* and the state-of-the-art complete solver *WPM-2014* on all testing WPMS benchmarks. As a reference, we also report the number of instances on which *Eva* proves optimality within the cutoff time, and the related results are summarized in the column '#prov. by *Eva*'. The run time is measured in CPU second.

Benchmark	#inst.	#prov. by *Eva*	*CCEHC*		*WPM-2014*		*VBS*	
			#win.	time	#win.	time	#win.	time
MSE2014_WPMS_Random	280	1	**279**	0.10	0	0	279	0.10
MSE2014_WPMS_Crafted	310	141	**247**	29.76	170	16.82	310	25.16
MSE2014_WPMS_Industrial	410	355	59	37.74	**396**	14.08	410	16.02
Computational protein design	20	0	**14**	130.34	6	42.01	20	103.84
Advanced encryption standard	7	1	**6**	17.57	1	96.03	7	28.78
Pedigree	20	17	14	65.48	**15**	28.09	20	64.65
Cluster expansion	6	0	**6**	0.05	0	0	6	0.05

Table 9
Experimental results of *CCEHC* and the state-of-the-art complete solver *WPM-2014* on the MSE2014_WPMS_Random benchmark, the MSE2014_WPMS_Crafted benchmark, and three application WPMS benchmarks of computational protein design, advanced encryption standard and cluster expansion. As a reference, we also report the number of instances on which *Eva* proves optimality within the cutoff time, and the related results are summarized in the column '#prov. by *Eva* (1800 CPU seconds)'. We note that in this table the cutoff time of complete solvers *WPM-2014* and *Eva* is enlarged to 1800 CPU seconds, while the cutoff time of *CCEHC* is still 300 CPU seconds. The run time is measured in CPU second.

Benchmark	#inst.	#prov. by *Eva* (1800 CPU seconds)	*CCEHC* (300 CPU seconds)		*WPM-2014* (1800 CPU seconds)		*VBS*	
			#win.	time	#win.	time	#win.	time
MSE2014_WPMS_Random	280	1	**279**	0.10	4	583.27	279	0.10
MSE2014_WPMS_Crafted	310	150	**245**	28.44	187	96.36	310	37.11
Computational protein design	20	0	**14**	130.34	6	329.56	20	190.11
Advanced encryption standard	7	1	**6**	17.57	1	96.49	7	28.85
Cluster expansion	6	0	**6**	0.05	0	0	6	0.05

We observe that the complete solvers perform worse than *CCEHC* on several groups of application instances, including the benchmarks of computational protein design, advanced encryption standard and cluster expansion, as well as instances in the MSE2014_WPMS_Random and MSE2014_WPMS_Crafted benchmarks. To study the performance of the complete solvers with longer cutoff time on these benchmarks, we test complete solvers *WPM-2014* and *Eva* on these three application WPMS benchmarks with a longer cutoff time of 1800 CPU seconds. The experimental results are reported in Table 9, which demonstrate that *WPM-2014* within the cutoff time of 1800 CPU seconds still performs worse than our *CCEHC* algorithm

Table 10
Experimental results of *CCEHC* and the state-of-the-art complete solver *WPM-2014* on all testing WPMS benchmarks within the cutoff time of 50 CPU seconds. As a reference, we also report the number of instances on which *Eva* proves optimality within the cutoff time, and the related results are summarized in the column '#prov. by *Eva*'. The run time is measured in CPU second.

Benchmark	#inst.	#prov. by *Eva* (50 CPU seconds)	*CCEHC* (50 CPU seconds)		*WPM-2014* (50 CPU seconds)		*VBS*	
			#win.	time	#win.	time	#win.	time
MSE2014_WPMS_Random	280	0	**279**	0.10	0	0	279	0.10
MSE2014_WPMS_Crafted	310	130	**239**	6.76	162	3.64	310	6.07
MSE2014_WPMS_Industrial	410	292	55	16.37	**394**	5.49	410	6.09
Computational protein design	20	0	**13**	27.72	7	6.14	20	20.17
Advanced encryption standard	7	1	**6**	16.03	1	6.10	7	14.61
Pedigree	20	10	**15**	14.66	14	2.34	20	10.41
Cluster expansion	6	0	**6**	0.05	0	0	6	0.05

Table 11
Experimental results of *CCEHC* and the state-of-the-art complete solver *WPM-2014* on all testing WPMS benchmarks within the cutoff time of 1800 CPU seconds. As a reference, we also report the number of instances on which *Eva* proves optimality within the cutoff time, and the related results are summarized in the column '#prov. by *Eva*'. The run time is measured in CPU second.

Benchmark	#inst.	#prov. by *Eva* (1800 CPU seconds)	*CCEHC* (1800 CPU seconds)		*WPM-2014* (1800 CPU seconds)		*VBS*	
			#win.	time	#win.	time	#win.	time
MSE2014_WPMS_Random	280	1	**279**	0.10	4	583.27	279	0.10
MSE2014_WPMS_Crafted	310	150	**253**	167.10	183	98.46	310	139.32
MSE2014_WPMS_Industrial	410	368	59	126.01	**401**	69.31	410	73.14
Computational protein design	20	0	**14**	579.84	6	329.56	20	504.76
Advanced encryption standard	7	1	**7**	144.81	0	0	7	144.81
Pedigree	20	19	13	206.28	**16**	92.77	20	206.25
Cluster expansion	6	0	**6**	0.05	0	0	6	0.05

Table 12
Experimental results of *CCEHC* and the state-of-the-art complete solver *WPM-2014* on all testing WPMS benchmarks within the cutoff time of 5400 CPU seconds. As a reference, we also report the number of instances on which *Eva* proves optimality within the cutoff time, and the related results are summarized in the column '#prov. by *Eva*'. The run time is measured in CPU second.

Benchmark	#inst.	#prov. by *Eva* (5400 CPU seconds)	*CCEHC* (5400 CPU seconds)		*WPM-2014* (5400 CPU seconds)		*VBS*	
			#win.	time	#win.	time	#win.	time
MSE2014_WPMS_Random	280	1	**279**	0.10	4	590.32	279	0.10
MSE2014_WPMS_Crafted	310	154	**257**	515.45	184	192.66	310	428.03
MSE2014_WPMS_Industrial	410	368	56	327.12	**406**	172.79	410	184.16
Computational protein design	20	0	**16**	2121.05	4	489.11	20	1794.66
Advanced encryption standard	7	1	**7**	144.12	0	0	7	144.12
Pedigree	20	19	13	819.00	**16**	90.47	20	602.64
Cluster expansion	6	0	**6**	0.06	0	0	6	0.06

within the cutoff time of 300 CPU seconds. This confirms the effectiveness of *CCEHC* on these three application WPMS benchmarks, the MSE2014_WPMS_Random benchmark and the MSE2014_WPMS_Crafted benchmark.

Additionally, we conduct more empirical evaluations of our *CCEHC* algorithm and the complete solver *WPM-2014* on all the testing WPMS benchmarks with various cutoff time (50 CPU seconds, 1800 CPU seconds and 5400 CPU seconds), in order to further study the performance of these solvers for WPMS. We also report the experimental results of the lower bound based complete solver *Eva* with these various cutoff time, as a reference. The experimental results of empirical evaluations within the cutoff time of 50 CPU seconds, 1800 CPU seconds and 5400 CPU seconds are demonstrated in Tables 10, 11 and 12, respectively. Table 10 shows that, within the cutoff time of 50 CPU seconds, *CCEHC* performs better than *WPM-2014* on the MSE2014_WPMS_Random benchmark, the MSE2014_WPMS_Crafted benchmark, and all four real-world application benchmarks. According to Tables 11 and 12, the experimental results present that, within the cutoff time of 1800 CPU seconds and 5400 CPU seconds, *CCEHC* outperforms *WPM-2014* on the MSE2014_WPMS_Random benchmark, the MSE2014_WPMS_Crafted benchmark, and three real-world application benchmarks of computational protein design, advanced encryption standard and cluster expansion. In conclusion, the empirical results with various cutoff time confirm the effectiveness of *CCEHC* on a large number of WPMS instances, which indicates that our *CCEHC* algorithm can be genuinely beneficial in practice.

Table 13
The parameter settings reported by *SMAC* for *CCEHC_alt1*, *CCEHC_alt2* and *CCEHC_alt3* on random WPMS instances, crafted WPMS instances, as well as industrial and application WPMS instances.

Instance type	*CCEHC_alt1*		*CCEHC_alt2*	*CCEHC_alt3*	
	p	*sp*	*p*	*p*	*sp*
Random WPMS	0.2	0.0001	0.2	0.2	0.0001
Crafted WPMS	0.354	0.035	0.514	0.048	0.763
Industrial and application WPMS	0.045	0.24	0.03	0.012	0.003

Table 14
Experimental results of *CCEHC* and its three alternative versions on all testing WPMS benchmarks. The run time is measured in CPU second.

Benchmark	#inst.	*CCEHC*		*CCEHC_alt1*		*CCEHC_alt2*		*CCEHC_alt3*		*VBS*	
		#win.	time	#win.	time	#win.	time	#win.	time	#win.	time
MSE2014_WPMS_Random	280	**279**	**0.10**	**279**	0.16	262	6.83	**279**	0.31	279	0.02
MSE2014_WPMS_Crafted	310	**221**	18.21	189	30.05	194	46.28	164	9.76	301	46.62
MSE2014_WPMS_Industrial	410	**190**	139.92	68	56.42	102	52.59	121	65.51	367	102.83
Computational protein design	20	8	104.51	9	19.30	9	56.82	**12**	56.07	15	71.94
Advanced encryption standard	7	5	11.08	5	2.46	5	9.12	**6**	75.66	7	47.97
Pedigree	20	**17**	70.66	8	57.51	9	28.52	7	30.34	19	72.71
Cluster expansion	6	**6**	**0.05**	**6**	0.13	**6**	0.08	**6**	0.36	6	0.05

7. Discussion

In this section, we present a detailed discussion of our *CCEHC* algorithm. First, we conduct further empirical analyses to present the effectiveness of each underlying component in the EHC heuristic. Then, we discuss the major differences between *CCEHC* and *CCLS*, as well as the main differences between *CCEHC* and *Dist*. Finally, we conduct more empirical evaluations to study the performance of the combination of the *CCEHC* algorithm and the unit propagation initialization.

7.1. Effectiveness of the EHC components

As explained in Section 4, there are three components underlying the EHC heuristic, i.e., the variable selection mechanism focusing on configuration checking based only on the states of hard clauses, the weighting scheme for hard clauses, and the biased random walk component. Thus, in order to demonstrate the effectiveness of these three components in the EHC heuristic, we conduct experiments to compare *CCEHC* with the three alternative versions in the following.

- *CCEHC_alt1:* This alternative version of *CCEHC* does not utilize the variable selection mechanism focusing on configuration checking based only on the states of hard clauses. In another word, this alternative version does not select variables to be flipped from the *HCSCCDvars* set during the search process (i.e., removing lines 9–10 in Algorithm 2). We note that the *CCEHC_alt1* algorithm would activate the weighting scheme for hard clauses when it works in the greedy mode.
- *CCEHC_alt2:* This alternative version of *CCEHC* does not utilize the weighting scheme for hard clauses (i.e., removing line 12 in Algorithm 2).
- *CCEHC_alt3:* This alternative version of *CCEHC* uses the standard random walk component rather than the biased random walk component, as the *CCLS* algorithm does (i.e., replacing the biased random walk component, i.e., lines 5–7 in Algorithm 2, with the standard random walk component, i.e., lines 5–6 in Algorithm 1).

Before the empirical study of *CCEHC* and its three alternative versions, we utilize the automatic configuration tool *SMAC* to tune the parameters for *CCEHC*'s three alternative versions. We use the parameter settings of *CCEHC* tuned by *SMAC* as the default values of the parameters for *CCEHC*'s three alternative versions. For *CCEHC_alt1* and *CCEHC_alt3*, the parameter domain of real number *p* is [0, 1], and the default value of *p* is 0.279 for industrial and application WPMS instances, 0.2 for random WPMS instances, and 0.177 for crafted WPMS instances; the parameter domain of real number *sp* is [0, 1], and the default value of *sp* is 0.085 for industrial and application WPMS instances, 0.0001 for random WPMS instances and 0.003 for crafted WPMS instances. For *CCEHC_alt2*, the parameter domain of real number *p* is 0.279 for industrial and application WPMS instances, 0.2 for random WPMS instances, and 0.177 for crafted WPMS instances (as *CCEHC_alt2* does not utilize the weighting scheme for hard clauses, thus it omits the parameter *sp*). We note that the configuration protocol for *SMAC* utilized here is the same as the one which is adopted in Section 6.4.

The parameter settings found by *SMAC* for *CCEHC_alt1*, *CCEHC_alt2* and *CCEHC_alt3* are summarized in Table 13, and we use these parameter settings in the following empirical study.

Empirical results for *CCEHC* and its three alternative versions on all testing WPMS benchmarks are reported in Table 14. As can be seen from Table 14, *CCEHC* performs generally better than its all three alternative versions, which indicates the effectiveness of all three components of our EHC heuristic. Particularly, on the MSE2014_WPMS_Crafted benchmark, the MSE2014_WPMS_Industrial benchmark and a real-world application WPMS benchmark (‘Pedigree’), *CCEHC* performs much better than all its three alternative versions in terms of the solution-quality, which confirms that our proposed EHC heuristic contributes to the performance of *CCEHC* on a number of structured WPMS instances and real-world application WPMS instances.

Then we investigate the complementarity of *CCEHC* and its alternative versions. The results of *VBS* indicate that these competing solvers are complementary in the MSE2014_WPMS_Crafted benchmark, the MSE2014_WPMS_Industrial benchmark, and three real-world application benchmarks of computational protein design, advanced encryption standard and pedigree, which indicates that it is promising to build an algorithm-selector consisting of *CCEHC* and its three alternative versions to improve the empirical performance.

7.2. *Main differences between* CCEHC *and* CCLS

Although our *CCEHC* algorithm is conceptually related to the *CCLS* algorithm [22], there exist major differences between *CCEHC* and *CCLS*. In this subsection, we summarize these major differences, and directly list them as follows.

- **How to deal with the input WPMS instance:** The *CCEHC* algorithm is designed for solving WPMS, so *CCEHC* is able to solve the input WPMS instance directly, while the *CCLS* algorithm is developed for solving weighted MAX-SAT, so *CCLS* needs to translate the input WPMS instance to weighted MAX-SAT, and then solve the resulting encoded weighted MAX-SAT instance.
- **Variable selection mechanism:** The *CCEHC* algorithm prefers to select the variable to be flipped by emphasizing the information of hard clauses, i.e., preferring the variable with the greatest *hscore*, while the *CCLS* algorithm does not distinguish hard clauses and soft clauses, and simply uses the general *score* property to choose the variable to be flipped.
- **Hard clause weighting scheme:** The *CCEHC* algorithm employs a clause weighting scheme that only works for hard clauses to diversify the search, while the *CCLS* algorithm does not use any clause weighting schemes.
- **Type of the random walk component:** In order to give a higher priority to hard clauses, the *CCEHC* algorithm applies a biased random walk component that first tries to select an unsatisfied hard clause, while the *CCLS* algorithm uses a standard random walk component, which is designed for non-partial MAX-SAT, and treats hard clauses and soft clauses as same.
- **Empirical performance on a wide range of WPMS benchmarks:** As can be clearly seen from the extensive experiments illustrated in Tables 2, 3, 4, 5, 6 and 7 in Section 6, the *CCEHC* algorithm performs much better than the *CCLS* algorithm on a wide range of WPMS benchmarks, indicating that the significant performance improvements of *CCEHC* over *CCLS* are due to the above major differences between these two SLS algorithms.

7.3. *Main differences between* CCEHC *and* Dist

As the *Dist* algorithm [3] also separates hard clauses and soft clauses (the pseudo-code of *Dist* can be found in the literature [3]), in this subsection, we discuss the main differences between *CCEHC* and *Dist* in detail. We would like to note that the only difference between *Dist* and *DistUP* [23] is the initialization procedure, so we here discuss the differences between *CCEHC* and *Dist*. The main differences between *CCEHC* and *Dist* are summarized below.

- **Basic framework:** The *CCEHC* algorithm is based on the *CCLS* algorithm which has been applied successfully to solving weighted MAX-SAT, while the *Dist* algorithm is developed following the simple *GSAT* algorithm which is the earliest SLS algorithm for solving SAT.
- **Variable selection mechanism:** The *CCEHC* algorithm utilizes the new HCSCC strategy to select variables to be flipped and thus prefers to select the HCSCCD variable, while the *Dist* algorithm does not use the HCSCC strategy and simply prefers choosing the decreasing variable (i.e., variable x with $hscore(x) > 0$ or with $hscore(x) = 0$ and $sscore(x) > 0$).
- **Time to activate the hard clause weighting scheme:** The *CCEHC* algorithm activates the hard clause weighting scheme when there is no HCSCCD variable, while *Dist* does that when no decreasing variable exists.
- **Time to call biased random walk:** The *CCEHC* algorithm calls the biased random walk component with a fixed probability at the start of each search step, while the *Dist* algorithm calls it after activating the hard clause weighting scheme.
- **Empirical performance on a broad range of WPMS benchmarks:** According to the experimental results presented in Tables 2, 3, 4, 5, 6 and 7 in Section 6, it can be clearly observed that *CCEHC* significantly outperforms *Dist* as well as *Dist*’s most recent improvement *DistUP* on a broad range of WPMS benchmarks, which indicates that the obvious superiority of *CCEHC* over *Dist* and *DistUP* is owing to these main differences.

7.4. *Initializing* CCEHC *by unit propagation*

Inspired by the success of the *DistUP* solver, which equips *Dist* with an unit propagation initialization [23], we combine *CCEHC* with the unit propagation initialization, and empirically evaluate the resulting hybrid solver on all testing benchmarks. By replacing *Dist* with *CCEHC* in the *DistUP* solver, we obtain a new solver namely *CCEHC+UP*. Then we conduct

Table 15
Experimental results of *CCEHC* and *CCEHC+UP* on all testing WPMS benchmarks. The run time is measured in CPU second.

Benchmark	#inst.	*CCEHC*		*CCEHC+UP*		*VBS*	
		#win.	time	#win.	time	#win.	time
MSE2014_WPMS_Random	280	**279**	0.10	**279**	**0.06**	279	0.06
MSE2014_WPMS_Crafted	310	249	32.10	**261**	37.55	301	46.42
MSE2014_WPMS_Industrial	410	153	119.38	**269**	78.41	361	98.95
Computational protein design	20	10	132.28	**11**	137.79	15	150.99
Advanced encryption standard	7	5	11.08	**6**	24.23	6	22.62
Pedigree	20	**16**	81.27	15	43.19	19	65.75
Cluster expansion	6	**6**	0.05	**6**	**<0.01**	6	<0.01

experiments to show the efficiency of *CCEHC+UP*. The comparative results of *CCEHC+UP* and *CCEHC* are reported in Table 15. According to Table 15, *CCEHC+UP* performs better than *CCEHC* on all the testing benchmarks but one ('Pedigree'). The results of *VBS* indicate that *CCEHC+UP* is complementary to *CCEHC* on the MSE2014_WPMS_Crafted benchmark, the MSE2014_WPMS_Industrial benchmark, and two real-world application benchmarks ('Computational Protein Design' and 'Pedigree'), which indicates that it is promising to build an algorithm-selector consisting of *CCEHC* and *CCEHC+UP* on solving WPMS instances. The experimental results also show that *CCEHC* could cooperate well with the unit propagation initialization on solving WPMS instances.

8. Conclusions and future work

In this work, we design a heuristic with emphasis on hard clauses, called EHC, for SLS algorithms for solving the WPMS problem. By incorporating the EHC heuristic into *CCLS*, we present a new SLS algorithm named *CCEHC* for solving WPMS.

To demonstrate the effectiveness and the robustness of our *CCEHC* algorithm, we evaluate *CCEHC* on a broad range of WPMS benchmarks, including random, crafted, industrial and real-world application instances. Our experimental results show that *CCEHC* significantly outperforms state-of-the-art SLS algorithms namely *CCLS*, *Dist* and *DistUP* on these WPMS benchmarks, indicating that *CCEHC* establishes a new state of the art on SLS algorithms for solving WPMS. Our further experimental results on evaluating *CCEHC* with a state-of-the-art complete solver *WPM-2014* show the effectiveness of *CCEHC* on random WPMS instances, crafted WPMS instances and many WPMS instances based on real-world applications. Particularly, on a number of important real-world application WPMS instances, where *WPM-2014* and a state-of-the-art lower-bound based complete solver *Eva* could not return good-quality solutions quickly (e.g., within the cutoff time of 5400 CPU seconds), our *CCEHC* algorithm is able to produce good-quality solutions fast, which indicates that *CCEHC* might be genuinely beneficial in practice. Further, we perform more empirical evaluations to analyze the effectiveness of the EHC heuristic, and the related experiments confirm the effectiveness of each underlying component in the EHC heuristic and demonstrate that our EHC heuristic contributes to the performance of *CCEHC* on a number of structured WPMS instances and real-world application WPMS instances. Also, we conduct empirical evaluations to study the combination of *CCEHC* and the unit propagation initialization and the related experiments show that *CCEHC* is able to cooperate well with the unit propagation initialization.

In future work, we would like to further study the flexibility of our EHC heuristic, and improve the performance of *CCEHC* on solving WPMS instances by integrating other powerful algorithmic techniques, such as the 'Best from Multiple Selections' strategy [59]. We would also like to build an algorithm-selector consisting of our *CCEHC* algorithm and other solvers to achieve better performance on solving WPMS instances.

Acknowledgements

This work is partially supported by the National Key Research and Development Program of China under Grant 2016YFB0200803 and Grant 2016YFC1401700, partially supported by the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing under Grant 2016A06, partially supported by the National Natural Science Foundation of China under Grant 61502464, Grant 61572234, Grant 61472369 and Grant 61370072, and partially supported by the Australian Research Council under Grant DP150101618. The authors would like to thank the anonymous reviewers for their constructive comments.

References

[1] P.W. Purdom, Solving satisfiability with less searching, IEEE Trans. Pattern Anal. Mach. Intell. 6 (4) (1984) 510–513.
[2] K. Smyth, H.H. Hoos, T. Stützle, Iterated robust tabu search for MAX-SAT, in: Proceedings of Canadian Conference on AI 2003, 2003, pp. 129–144.
[3] S. Cai, C. Luo, J. Thornton, K. Su, Tailoring local search for partial MaxSAT, in: Proceedings of AAAI 2014, 2014, pp. 2623–2629.
[4] D. Allouche, S. Traoré, I. André, S. de Givry, G. Katsirelos, S. Barbe, T. Schiex, Computational protein design as a cost function network optimization problem, in: Proceedings of CP 2012, 2012, pp. 840–849.
[5] D. Allouche, I. André, S. Barbe, J. Davies, S. de Givry, G. Katsirelos, B. O'Sullivan, S.D. Prestwich, T. Schiex, S. Traoré, Computational protein design as an optimization problem, Artif. Intell. 212 (2014) 59–79.

[6] Z. Naji-Azimi, P. Toth, L. Galli, An electromagnetism metaheuristic for the unicost set covering problem, Eur. J. Oper. Res. 205 (2) (2010) 290–300.
[7] X. Liao, M. Koshimura, H. Fujita, R. Hasegawa, Solving the coalition structure generation problem with MaxSAT, in: Proceedings of ICTAI 2012, 2012, pp. 910–915.
[8] H.L. Chieu, W.S. Lee, Relaxed survey propagation for the weighted maximum satisfiability problem, J. Artif. Intell. Res. 36 (2009) 229–266.
[9] H. Lin, K. Su, Exploiting inference rules to compute lower bounds for MAX-SAT solving, in: Proceedings of IJCAI 2007, 2007, pp. 2334–2339.
[10] H. Lin, K. Su, C.M. Li, Within-problem learning for efficient lower bound computation in Max-SAT solving, in: Proceedings of AAAI 2008, 2008, pp. 351–356.
[11] C.M. Li, F. Manyà, N.O. Mohamedou, J. Planes, Exploiting cycle structures in Max-SAT, in: Proceedings of SAT 2009, 2009, pp. 467–480.
[12] M. Davis, H. Putnam, A computing procedure for quantification theory, J. ACM 7 (3) (1960) 201–215.
[13] M. Davis, G. Logemann, D.W. Loveland, A machine program for theorem-proving, Commun. ACM 5 (7) (1962) 394–397.
[14] C. Ansótegui, M.L. Bonet, J. Gabàs, J. Levy, Improving WPM2 for (weighted) partial MaxSAT, in: Proceedings of CP 2013, 2013, pp. 117–132.
[15] C. Ansótegui, M.L. Bonet, J. Levy, SAT-based MaxSAT algorithms, Artif. Intell. 196 (2013) 77–105.
[16] N. Narodytska, F. Bacchus, Maximum satisfiability using core-guided MaxSAT resolution, in: Proceedings of AAAI 2014, 2014, pp. 2717–2723.
[17] J.P.M. Silva, K.A. Sakallah, GRASP – a new search algorithm for satisfiability, in: Proceedings of ICCAD 1996, 1996, pp. 220–227.
[18] J.P.M. Silva, K.A. Sakallah, GRASP: a search algorithm for propositional satisfiability, IEEE Trans. Comput. 48 (5) (1999) 506–521.
[19] H.H. Hoos, T. Stützle, Stochastic Local Search: Foundations & Applications, Elsevier/Morgan Kaufmann, 2004.
[20] B. Selman, H.J. Levesque, D.G. Mitchell, A new method for solving hard satisfiability problems, in: Proceedings of AAAI 1992, 1992, pp. 440–446.
[21] B. Selman, H.A. Kautz, B. Cohen, Noise strategies for improving local search, in: Proceedings of AAAI 1994, 1994, pp. 337–343.
[22] C. Luo, S. Cai, W. Wu, Z. Jie, K. Su, CCLS: an efficient local search algorithm for weighted maximum satisfiability, IEEE Trans. Comput. 64 (7) (2015) 1830–1843.
[23] S. Cai, C. Luo, J. Lin, K. Su, New local search methods for partial MaxSAT, Artif. Intell. 240 (2016) 1–18.
[24] S. Cai, K. Su, Local search for Boolean satisfiability with configuration checking and subscore, Artif. Intell. 204 (2013) 75–98.
[25] S. Cai, C. Luo, K. Su, Scoring functions based on second level score for k-SAT with long clauses, J. Artif. Intell. Res. 51 (2014) 413–441.
[26] S. Cai, K. Su, C. Luo, A. Sattar, NuMVC: an efficient local search algorithm for minimum vertex cover, J. Artif. Intell. Res. 46 (2013) 687–716.
[27] C. Luo, K. Su, S. Cai, Improving local search for random 3-SAT using quantitative configuration checking, in: Proceedings of ECAI 2012, 2012, pp. 570–575.
[28] C. Luo, S. Cai, W. Wu, K. Su, Focused random walk with configuration checking and break minimum for satisfiability, in: Proceedings of CP 2013, 2013, pp. 481–496.
[29] C. Luo, S. Cai, K. Su, W. Wu, Clause states based configuration checking in local search for satisfiability, IEEE Trans. Cybern. 45 (5) (2015) 1014–1027.
[30] S. Cai, Z. Jie, K. Su, An effective variable selection heuristic in SLS for weighted Max-2-SAT, J. Heuristics 21 (3) (2015) 433–456.
[31] C. Luo, S. Cai, W. Wu, K. Su, Double configuration checking in stochastic local search for satisfiability, in: Proceedings of AAAI 2014, 2014, pp. 2703–2709.
[32] Z. Wu, B.W. Wah, An efficient global-search strategy in discrete lagrangian methods for solving hard satisfiability problems, in: Proceedings of AAAI 2000, 2000, pp. 310–315.
[33] D. Schuurmans, F. Southey, R.C. Holte, The exponentiated subgradient algorithm for heuristic Boolean programming, in: Proceedings of IJCAI 2001, 2001, pp. 334–341.
[34] F. Hutter, D.A.D. Tompkins, H.H. Hoos, Scaling and probabilistic smoothing: efficient dynamic local search for SAT, in: Proceedings of CP 2002, 2002, pp. 233–248.
[35] J. Thornton, D.N. Pham, S. Bain, V. Ferreira Jr., Additive versus multiplicative clause weighting for SAT, in: Proc. of AAAI 2004, 2004, pp. 191–196.
[36] A. Ishtaiwi, J. Thornton, A. Sattar, D.N. Pham, Neighbourhood clause weight redistribution in local search for SAT, in: Proceedings of CP 2005, 2005, pp. 772–776.
[37] Y. Jiang, H. Kautz, B. Selman, Solving problems with hard and soft constraints using a stochastic algorithm for MAX-SAT, in: First International Joint Workshop on Artificial Intelligence and Operations Research, 1995.
[38] C. Ansótegui, J. Gabàs, Solving (weighted) partial MaxSAT with ILP, in: Proceedings of CPAIOR 2013, 2013, pp. 403–409.
[39] K. Xu, W. Li, Many hard examples in exact phase transitions, Theor. Comput. Sci. 355 (3) (2006) 291–302.
[40] K. Xu, W. Li, Exact phase transitions in random constraint satisfaction problems, J. Artif. Intell. Res. 12 (2000) 93–103.
[41] M. Gwynne, O. Kullmann, Towards a better understanding of SAT translations, in: The Twelfth International Workshop on Logic and Computational Complexity, 2011.
[42] M. Sánchez, S. de Givry, T. Schiex, Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques, Constraints 13 (1–2) (2008) 130–154.
[43] J.M. Sanchez, Cluster expansions and the configurational energy of alloys, Phys. Rev. B 48 (18) (1993) 14013–14015.
[44] B. Liu, B.M. Baas, Parallel AES encryption engines for many-core processor arrays, IEEE Trans. Comput. 62 (3) (2013) 536–547.
[45] S. Trimberger, Security in SRAM FPGAs, IEEE Des. Test Comput. 24 (6) (2007) 581.
[46] M.M. Kermani, A. Reyhani-Masoleh, Concurrent structure-independent fault detection schemes for the advanced encryption standard, IEEE Trans. Comput. 59 (5) (2010) 608–622.
[47] L. Aceto, J.A. Hansen, A. Ingólfsdóttir, J. Johnsen, J. Knudsen, The complexity of checking consistency of pedigree information and related problems, J. Comput. Sci. Technol. 19 (1) (2004) 42–59.
[48] W. Huang, D.A. Kitchaev, S. Dacek, Z. Rong, A. Urban, S. Cao, C. Luo, G. Ceder, Finding and proving the exact ground state of a generalized Ising model by convex optimization and MAX-SAT, Phys. Rev. B 94 (2016) 134424.
[49] E. Ising, Beitrag zur theorie des ferromagnetismus, Z. Phys. 31 (1) (1925) 253–258.
[50] A. van de Walle, G. Ceder, Automating first-principles phase diagram calculations, J. Phase Equilib. 23 (4) (2002) 348–359.
[51] Y. Hinuma, Y.S. Meng, G. Ceder, Temperature–concentration phase diagram of $P2$-Na_xCoO_2 from first-principles calculations, Phys. Rev. B 77 (2008) 224111.
[52] A. Kügel, Improved exact solver for the weighted Max-SAT problem, in: Proceedings of Pragmatics of SAT 2010, 2010, pp. 15–27.
[53] A. Abramé, D. Habet, Inference rules in local search for Max-SAT, in: Proceedings of ICTAI 2012, 2012, pp. 207–214.
[54] D.L. Berre, A. Parrain, The Sat4j library, release 2.2, J. Satisf. Boolean Model. Comput. 7 (2–3) (2010) 59–64.
[55] J. Thornton, S. Bain, A. Sattar, D.N. Pham, A two level local search for MAX-SAT problems with hard and soft constraints, in: Proceedings of Australian Joint Conference on AI 2002, 2002, pp. 603–614.
[56] O. Roussel, Controlling a solver execution with the runsolver tool, J. Satisf. Boolean Model. Comput. 7 (4) (2011) 139–144.
[57] F. Hutter, H.H. Hoos, K. Leyton-Brown, T. Stützle, ParamILS: an automatic algorithm configuration framework, J. Artif. Intell. Res. 36 (2009) 267–306.
[58] F. Hutter, H.H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: Proceedings of LION 2011, 2011, pp. 507–523.
[59] S. Cai, Balance between complexity and quality: local search for minimum vertex cover in massive graphs, in: Proceedings of IJCAI 2015, 2015, pp. 747–753.