Improving Local Search for Structured SAT Formulas via Unit Propagation Based Construct and Cut Initialization

Shaowei Cai
State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology,
University of Chinese Academy of Sciences, Beijing, China

Chuan Luo
School of Software, Beihang University, Beijing, China

Xindi Zhang
State Key Laboratory of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology,
University of Chinese Academy of Sciences, Beijing, China

Jian Zhang
State Key Laboratory of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology,
University of Chinese Academy of Sciences, Beijing, China

Abstract

This work is dedicated to improving local search solvers for the Boolean satisfiability (SAT) problem on structured instances. We propose a construct-and-cut (CnC) algorithm based on unit propagation, which is used to produce initial assignments for local search. We integrate our CnC initialization procedure within several state-of-the-art local search SAT solvers, and obtain the improved solvers. Experiments are carried out with a benchmark encoded from a spectrum repacking project as well as benchmarks encoded from two important mathematical problems namely Boolean Pythagorean Triple and Schur Number Five. The experiments show that the CnC initialization improves the local search solvers, leading to better performance than state-of-the-art SAT solvers based on Conflict Driven Clause Learning (CDCL) solvers.

2012 ACM Subject Classification Theory of computation → Randomized local search

Keywords and phrases Satisfiability, Local Search, Unit Propagation, Mathematical Problems

Digital Object Identifier 10.4230/LIPIcs.CP.2021.5

Category Short Paper

Supplementary Material *Software (Source Code)*: `https://github.com/caiswgroup/CNC-LS`
archived at `swh:1:dir:f7ef44ee596e5f008dea01ef7e3c1ee47c8b93dc`

Funding This work was supported by Beijing Academy of Artificial Intelligence (BAAI), and Youth Innovation Promotion Association, Chinese Academy of Sciences (No. 2017150), as well as the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (Grant No. QYZDJ-SSW-JSC036).

1 Introduction

Given a Boolean formula, the Boolean Satisfiability problem (SAT) determines whether the variables of the formula can be assigned in such a way as to make the formula evaluate to TRUE. In the SAT problem, Boolean formulas are usually presented in Conjunctive Normal Form (CNF), i.e., $F = \bigwedge_i \vee_j \ell_{ij}$. SAT is the first NP-complete problem. Besides, SAT solvers have shown great success in many applications [29], including bounded model checking [10], program verification [11], and mathematical theorem proving [17].

Two popular methods for SAT are conflict driven clause learning (CDCL) [33] and local search. The CDCL based solvers evolve from the DPLL backtracking procedure [13] and combine reasoning techniques. The reasoning techniques in CDCL solvers, particularly unit propagation (UP) and clause learning, play a critical role in the good performance of CDCL solvers on application instances. Local search is an incomplete method and its process can be viewed as a random walk in the search space [19, 27]. Local search SAT solvers begin with an initial complete assignment and iteratively modify the assignment, until a model is found or a resource limit (usually the time limit) is reached [19, 28, 26]. Local search solvers are usually much simpler and lighter than CDCL ones. Indeed, they are probably the most lightweight SAT solvers. Local search has proved very effective for solving many NP-hard combinatorial problems. However, it is known that local search solvers are not effective as CDCL solvers on solving structured SAT instances, particularly those from real-world applications.

This work aims to improve local search solvers for structured SAT instances. Specifically, we propose a construct-and-cut (CnC) method for generating initial assignments for local search, which aims to produce diverse complete assignments as *consistent* as possible. The CnC method iteratively performs assigning procedures, which are also called construction tries, based on unit propagation and heuristics. In each construction try, the algorithm starts from an empty assignment and extends it to a complete assignment. Also, the algorithm records the best solution found (with fewest empty clauses) so far and its number of empty clauses which serves as an upper bound. In the subsequent tries, once the number of empty clauses reaches the upper bound, the try is cut off.

We use this CnC method to improve three state-of-the-art local search SAT solvers, by replacing the original initialization method with the CnC method.We conduct experiments with three important benchmarks, one of which arises from a recent real-world project about spectrum repacking [32] and the others consist of instances encoded from two important mathematical problems namely Boolean Pythagorean Triple [17] and Schur Number Five [16]. Experiment results show that, the CnC method brings obvious improvements to the local search solvers. Particularly, one of the CnC-enhanced local search solver outperforms modern SAT solvers based on CDCL approach on the three benchmarks.

2 Technical Background

2.1 Preliminary Definitions and Notations

Given a set of Boolean *variables* $\{x_1, x_2, ..., x_n\}$, a *literal* is either a variable x_i or its negation $\overline{x_i}$. A conjunctive normal form (CNF) formula F is a conjunction of clauses (i.e., $F = C_1 \wedge C_2 \wedge ... \wedge C_m$), where a *clause* is a disjunction of literals (i.e., $C_i = \ell_{i1} \vee \ell_{i2} \vee ... \vee \ell_{ij}$). Alternatively, a CNF formula can be viewed as a set of clauses, and a clause can be viewed as a set of literals. For a formula F, we denote the set of variables in F by $Var(F)$, and the number of literals whose corresponding variable is x_i is denoted by $\Delta_F(x_i)$.

For a literal ℓ, its corresponding variable is denoted by $\ell.var$, and its *phase*, denoted by $\ell.phase$, is 1 if ℓ is positive and 0 if ℓ is negative. A literal can be viewed as an ordered pair of a variable and its phase, i.e., $\ell = (\ell.var, \ell.phase)$. For a literal ℓ, we denote by $\overline{\ell}$ the literal of opposite phase. A clause containing only one literal is a *unit clause*. We denote $\ell \in C_i$ if ℓ is a literal in clause C_i.

For a formula F, an *assignment* α is a mapping $Var(F) \to \{0, 1\}$. If α maps all variables to a Boolean value, we say it is a *complete* assignment. For a variable $x_i \in Var(F)$ and an assignment α, $\alpha[x_i]$ is the value of variable x_i under α. Given an assignment α, we say that a literal ℓ is true if $\alpha[\ell.var]$ is equal to $\ell.phase$. A clause is *satisfied* if it has at least one true literal, and *unsatisfied* if all the literals in the clause are false literals. By convention the empty clause $\square$ is always unsatisfiable, and represents a conflict. SAT is the problem of deciding whether a given CNF formula is satisfiable.

The process of conditioning a CNF formula F on a literal ℓ amounts to replacing every occurrence of literal ℓ by the constant true, replacing $\overline{\ell}$ by the constant false, and simplifying accordingly. The result of conditioning F on ℓ is denoted by $F|_\ell$ and can be described succinctly as follows: $F|_\ell = \{c/\{\overline{\ell}\} | c \in F, \ell \notin c\}$. Note that $F|_\ell$ does not contain any literal ℓ or $\overline{\ell}$. When we assign a variable x with a value v, we can simplify the formula accordingly, and the simplified formula is denoted as $F|_{(x,v)}$.

Unit propagation on a CNF formula ϕ works as follows: First, we collect all unit clauses in ϕ, and then assume that variables are set to satisfy these unit clauses. If the unit clause $\{x_i\}$ appears in the formula, we set x_i to true. Also, if the unit clause $\{\overline{x_i}\}$ appears in the formula, we set x_i to false. We then condition the formula on these settings. The iterative application of this rule until no more unit clause remains is called *unit propagation* (UP).

2.2 Local Search for SAT

When solving a SAT formula by local search, the search space is organized as a network, in which each position represents a complete assignment and two positions are adjacent if they are neighbors. A commonly used neighborhood relation N maps assignments to their set of Hamming neighbors, i.e., assignments that differ in exactly one variable. Typically, a local search algorithm for SAT starts from a complete assignment, and flips a variable iteratively to search for a satisfying assignment. In this work, we focus on improving local search for SAT by generating good initial assignments.

3 Related Works and Discussions

This work utilizes a construct-and-cut method based on unit propagation (UP) to produce a good quality initial assignment for local search SAT solvers. Unit propagation is a simple form of reasoning, and has been used to improve local search solver previously.

Some local search solvers use UP to simplify the formula before the search [21, 8]. More complicated preprocessors have also been developed [31]. These preprocessing techniques are used to simplify the formula. If the formula cannot be simplified, they just do nothing.

Some algorithms use UP during local search. *UnitWalk* [18] prefers to perform UP if possible in each local search step, and only when UP is not applicable a normal local search step is executed. *QingTing* [22] is an improved version of *UnitWalk* with more efficient implementation and also switches between *UnitWalk* and a normal local search algorithm. *EagleUP* [15] also exploits UP during local search, where UP is performed only when the algorithm is stuck in local optima.

Although UP has been previously combined with local search, these previous works either use UP only as preprocessor, or use UP too heavily. These solvers usually improve local search on crafted and random instances, but no good result is reported on solving instances from real-world applications. Most previous local search solvers, including *CCAnr* [8], *Sattime* [21] and *ProbSAT* [4], generate the initial assignment randomly, while a recent local search solver *YalSAT* [5] also utilizes information such as the best found assignment in the last round to produce the initial assignment. On the other hand, UP-based initialization has been used in local search for MaxSAT [7, 9, 25]. However, in these works, the initialization does not use pruning techniques.

Another relevant direction is using CDCL to boost local search solvers. An incomplete hybrid solver *hybridGM* [3] calls CDCL search around local minima with only one unsatisfied clause. *SATHYS* [1] performs local search and calls a CDCL solver when it is stuck in local optima. However, these methods do not show improvement over the CDCL solvers on application benchmarks, although they show better performance than local search on crafted instances and better performance than CDCL solvers on random instances.

4 A Novel Initialization Method for Local Search SAT Solvers

This section presents the construct-and-cut (CnC) method, which can be used to produce good quality assignments for local search SAT solvers. The CnC algorithm consists of individual construction procedures, each of which constructs a complete assignment by assigning variables one by one.

4.1 The Construct-and-Cut Method

Before presenting the details of the CnC algorithm, we first introduce the key data structures used in the algorithm.

Set $\mathcal{U}$: it stores all unit clauses, noting that a unit clause has only one literal. $\mathcal{U}$ is updated during the search. Newly generated unit clauses are put into $\mathcal{U}$, and a unit clause is removed from $\mathcal{U}$ after it is picked to perform unit propagation.

Vector *value*: this vector records the assigned value for each variable. For each variable x, $value(x)$ has 4 possible values $\{-2, -1, 0, 1\}$, as explained below:

- $value(x) = -2$ means unit clauses x and $\overline{x}$ appear in F simultaneously (may be due to different UP operations).
- $value(x) = -1$ means x is unassigned.
- $value(x) = 0$ means x is assigned the value 0 (false).
- $value(x) = 1$ means x is assigned the value 1 (true).

The CnC method is depicted in Algorithm 1. The algorithm consists of individual construction procedures (also called tries), and the number of tries to be executed is controlled by a parameter *cnc_times*. We use $\#(\square)$ to denote the number of empty clauses in the formula that the CnC algorithm is currently dealing with, which is the cost of the current assignment. The cost of the best assignment found (e.g. the minimum cost) in previous construction procedures is denoted as $cost^*$. In the beginning, CnC initializes $cost^*$ as the number of clauses in the input formula, and stores all unit clauses (if any) in $\mathcal{U}$.

In each try, the algorithm works on a copy of the input formula ϕ, which is denoted as F. In the beginning of each try, $value(x)$ is initialized as -1 for each variable (line 5), indicating that all variables are unassigned. Then, a loop is executed until there is no unassigned variable; moreover, the loop is terminated if $\#(\square)$ reaches $cost^*$.

Algorithm 1 CnC(ϕ, *cnc_times*).

Input: A CNF formula ϕ, *cnc_times*
Output: An assignment α^* of variables in ϕ

```
1  cost* ← + the number of clauses in F;
2  for i ← 1 to cnc_times do
3      F ← φ;
4      U ← {all unit clauses in φ};
5      ∀x ∈ Var(F), value(x) ← −1;
6      while ∃ unassigned variables do
7          if U ≠ ∅ then
8              ℓ ← GetUL(U);
9              x ← ℓ.var;
10             if value(x) = −1 then
11                 value(x) ← ℓ.phase;
12             else
13                 value(x) ← a random value from {0,1};
14         else
15             x ← GetUnassignedVar();
16             value(x) ← a random value from {0,1};
17         Simplify F accordingly;
18         foreach newly generated unit clause r do
19             if r ∉ U & r̄ ∉ U then
20                 U ← U ∪ {r};
21             else if r̄ ∈ U then
22                 value(r.var) ← −2;
23         if #(□) ≥ cost* then break;
24     if #(□) < cost* then
25         α* ← value; cost* ← #(□);
26 return α*;
```

If $\mathcal{U}$ is not empty, one literal ℓ is extracted from $\mathcal{U}$ via the function `GetUL` to do unit propagation. Let us denote $x = \ell.var$. We know that x could not have been assigned (either to 0 or 1). This is because if x is assigned, literals of x would not appear in the formula and $\mathcal{U}$. Thus, $value(x)$ is either -1 or -2. If $value(x) = -1$ (e.g., x is unassigned), then x is assigned the value of $\ell.phase$ to satisfy the unit clause ℓ; if $value(x) = -2$, x is assigned randomly. We would like to mention that, most variables are assigned by UP in the CnC algorithm.

If $\mathcal{U}$ is empty, then an unassigned variable is chosen by the `GetUnassignedVar` function, and is assigned a random value.

Whenever a variable x is assigned a value v, the formula F is simplified accordingly. The result of simplifying F on a literal ℓ can be described succinctly as $F|_\ell = \{c/\{\bar{\ell}\} | c \in F, \ell \notin c\}$ [12]. Moreover, for any newly generated unit clause r, if neither r nor $\bar{r}$ is in $\mathcal{U}$, then r is added into $\mathcal{U}$; if $\bar{r}$ is already in $\mathcal{U}$, we set $value(r.var)$ to -2 to indicate the conflicting status.

4.2 Main Functions

There are two functions that need to be specified in the CnC algorithm, and they are presented below.

Algorithm 2 Local Search with CnC.

Input: A CNF formula ϕ
Output: A satisfying assignment of ϕ if found
while *not reach time limit* **do**
  $\alpha_0 \leftarrow CnC(\phi, cnc\_times)$;
  if α_0 *satisfies* ϕ **then return** α_0;
  $\alpha \leftarrow LocalSearch(\alpha_0, StepLimit)$;
  if α *satisfies* ϕ **then return** α;
return "UNKNOWN";

`GetUL`: the function picks a unit clause in $\mathcal{U}$ to perform UP. In the first construction procedure, the function simply picks a random unit clause to perform UP. For the following construction procedures, the function utilizes a strategy as follows. The idea is to employ assigning orders as distant as possible in different tries, so as to exploit diverse reason chains, among which a good one may be touched. Our heuristic is based on a diversification property. For a variable, we use $prev\_assign\_step(x)$ to denote the step number in which it was assigned in the previous try of CnC. Our heuristic prefers to pick a variable with the largest $prev\_assign\_step$ value.

`GetUnassignedVar`: the function picks an unassigned variable to assign value. We use the same heuristic as in `GetUL`. In the first construction procedure, a randomized strategy is used, and in other procedures a diversification strategy is employed to pick the one with the largest $prev\_assign\_step$ values.

An important implementation detail is that we use a sampling method for approximately implementing the heuristic of picking a variable with the largest $prev\_assign\_step$ value. We randomly pick a certain number (which is fixed to 10 according to the preliminary experiments) of candidate variables and pick the one with the largest $prev\_assign\_step$ value. So, we do not need to sort the variables or scan all of them in each iteration. This allows the linear complexity of our method, as picking a variable to assign can always be done in $O(1)$ time, and the unit propagation in one iteration can be done in $\Delta_\phi(x_i)$ in the worst case, where x_i is the chosen variable. Since $\sum_{i=1}^{n} \Delta_\phi(x_i) = L(\phi)$, the worst case complexity of one CnC try is bounded by $O(L(\phi))$, where $L(\phi)$ is the length of the formula ϕ.

5 Integrating CnC to Local Search SAT Solvers

In this section, we apply the CnC method to improve local search SAT solvers. The framework of a local search SAT solver equipped with CnC is depicted in Algorithm 2. As it shows, the solver calls CnC to produce an initial assignment, which is handed to a local search algorithm for further improvements, trying to find a satisfying assignment. Local search SAT solvers may have different restart criterion, which is based on a limit on the steps. So, for each time the solver restarts, an initial assignment is produced by CnC and then modified by a local search process.

We apply the CnC method to three state-of-the-art local search SAT solvers for structured formulas, including *Sattime* [20], *ProbSAT* [4] and *CCAnr* [8]. *Sattime* is the only example that a local search solver beats all CDCL solvers in the crafted track of a SAT competition (in 2011) [20]. *ProbSAT* is a local search solver based on probability distribution and won the random track in SAT Competition 2013; it is an improved version of another local search solver *Sparrow* [2], which also uses probability distribution functions. *CCAnr* is a local search designed with the purpose of solving non-random (structured) SAT instances, and has been found effective on some application benchmarks [14].

We also note that a recent local search solver *YalSAT* performs well on a wide range of benchmarks, winning the random track of SAT Competition 2017, and is able to solve some hard crafted and application instances in SAT competitions [5]. Nevertheless, *YalSAT* utilizes the Lubby restarting scheme [24] and has very frequent restart in the early stage. This makes it ineffective to integrate CnC into *YalSAT*, due to the heavy overhead.

6 Experiments

To evaluate the effectiveness of our CnC method, we compare state-of-the-art local search solvers with their CnC enhanced versions on three important benchmarks of structured SAT formulas. Also, we compare the best CnC-enhanced local search solver against state-of-the-art CDCL solvers.

6.1 Benchmarks

Our experiments are conducted with three important benchmarks, including instances encoded from a real-world project and two important mathematical problems.

FCC: Recently, SAT solvers have been used by the US Federal Communication Commission (FCC) for spectrum repacking in the context of bandwidth auction which resulted in about 7 billion dollar revenue [32]. The SAT instances from this project are available on line [1] [32]. This benchmark contains 10000 instances, 9482 of which are known to be satisfiable and 121 unsatisfiable, while the satisfiability of the remaining 397 instances are unknown. As local search solvers such as UPLS are unable to prove unsatisfiability, we discard the unsatisfiable instances, leading to 9879 instances in this benchmark.

PTN: This benchmark consists of instances encoded from a mathematical problem named Boolean Pythagorean Triples. This problem used to be a long-term open mathematical problem and recently has been solved by SAT techniques, resulting in the currently largest-sized mathematical proof [17]. Marijn et al. proved the answer to Boolean Pythagorean Triples (PTN) problem is NO, by encoding PTN into SAT instances, including both satisfiable and unsatisfiable ones, and solving them. Our PTN benchmark contains only the satisfiable instances.[2] There are 23 instances in this benchmark.

SN5: The instances in this benchmark are encoded from a mathematical problem called Schur Number Five (SN5) and its variants. [16] proved the solution by encoding the century-old problem into SAT instances, and the proof of the solution is about two petabytes in size. Our SN5 benchmark contains 6 satisfiable instances.[3]

6.2 Solvers

The CnC method is implemented into the local search solvers in C++. For a local search solver A, the solver which integrates CnC is denoted as A+cnc in our experiments. The *cnc_times* parameter is set to 20, which is tuned on a training set consisting of 100 random FCC instances, half PTN instances and all SN5 instances.

[1] https://www.cs.ubc.ca/labs/beta/www-projects/SATFC/cacm_cnfs.tar.gz
[2] https://www.cs.utexas.edu/~marijn/ptn/
[3] https://www.cs.utexas.edu/~marijn/Schur/

Table 1 Results of local search solvers and CnC-enhanced local search solvers on all benchmarks.

Benchmark	*CCAnr*		*CCAnr+cnc*		*ProbSAT*		*ProbSAT+cnc*		*Sattime*		*Sattime+cnc*		*YalSAT*	
	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2
FCC (9879)	7878	2091.6	**8110**	**1868.2**	5407	4577.7	5477	4506.5	7054	2911.8	7078	2900.0	7136	2881.1
PTN (23)	13	4718.0	**23**	**127.0**	5	7885.0	20	2161.7	9	6790.7	18	2945.3	14	4490.3
SN5 (6)	2	7364.5	**4**	**4969.5**	0	10000.0	0	10000.0	0	10000.0	1	8708.7	0	10000.0

Table 2 Results of *CCAnr+cnc* and its CDCL competitors on all benchmarks.

	CCAnr+cnc		*CaDiCaL*		*CaDiCaL_sat*		*Maple_LCM_Dist*		*MapleCOMSPS*		*Kissat*		*Kissat_sat*	
	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2
FCC (9879)	8110	1868.2	7674	2326.9	7783	2211.9	7788	2183.2	7783	2183.0	7949	2042.8	**8163**	**1819.1**
PTN (23)	**23**	**127.0**	17	3274.2	17	3007.4	0	10000.0	1	9639.0	19	2215.7	21	1402.5
SN5 (6)	**4**	**4969.5**	0	10000.0	0	10000.0	0	10000.0	0	10000.0	0	10000.0	1	9130.7

The solvers *Sattime* and *ProbSAT* are downloaded from the website of SAT Competition 2013. For *CCAnr*, we used the latest version which is available online[4]. We include *YalSAT* in our experiment, which is downloaded from the website of SAT Comptition 2017.[5] Additionally, we tested *UnitWalk* [18] – a typical local search solver using unit propagation.[6]

We also compare the best local search solver obtained by CnC (namely *CCAnr+cnc*) against four state-of-the-art CDCL solvers, including *MapleCOMSPS* [23], *Maple_LCM_Dist* [30], *CaDiCaL* [5] and *Kissat* (including *Kissat_default* and *Kissat_sat*) [6]. *MapleCOMSPS* won the gold medal of Main Track of SAT Competition 2016 and the silver medal of Main Track of SAT Competition 2017, while *Maple_LCM_Dist* won the gold medal of Main Track of SAT Competition 2017 and the winner of the main track of SAT Competition 2018 is also a version of *Maple_LCM_Dist*. *CaDiCaL* solved the most instances in the Main Track of SAT Competition 2019. Particularly, *CaDiCaL* solved the most satisfiable instances in the track. Also, *Kissat_sat* won the gold medal of Main Track of SAT Competition 2020. All these CDCL solvers are downloaded from the website of SAT Competitions.

6.3 Experiment Results

All experiments were conducted on a cluster of computers with 2.10GHz Intel Xeon CPUs and 94GB RAM under the operating system CentOS. For each instance, each solver was performed one run, with 5000 CPU seconds as cutoff. For each solver for each benchmark, we report the number of solved SAT instances denoted "#SAT" and the penalized run time denoted "PAR2" (as used in SAT Competitions), where the run time of a failed run is penalized as twice the cutoff time. The results in **bold** indicates the best performance for a benchmark.

Table 1 presents the results of the local search solvers on the three benchmarks. *UnitWalk* performs much worse than other solvers (solving 4597 FCC instances and none of the other two benchmarks) and is not listed in the table. The CnC method improves local search solvers, particularly on the PTN instances. *CCAnr+cnc* gives the best performance on all the benchmarks. It solves 8110 out of 9879 FCC instances, 4 out of 6 SN5 instances and all PTN instances, showing significantly superiority over all other local search solvers.

[4] `https://lcs.ios.ac.cn/~caisw/Code/CCAnr-1.1.zip`
[5] `https://baldur.iti.kit.edu/sat-competition-2017/solvers/`
[6] `https://logic.pdmi.ras.ru/~arist/UnitWalk/unitwalk3.tar.gz`

We compare *CCAnr+cnc* with state-of-the-art CDCL solvers. Table 2 shows the results of *CCAnr+cnc* and its CDCL competitors. The best CDCL solver is *Kissat_sat*, which outperforms other CDCL solvers on all the benchmarks. Encouragingly, *CCAnr+cnc* is able to solve more instances than the CDCL solvers on all the benchmarks, with only one exception – *CCAnr+cnc* performs a bit fewer FCC instances than Kissat_sat. Particularly, *CCAnr+cnc* solves four SN5 instances, while *Kissat_sat* solves only one SN5 instance and other CDCL solvers fail to solve any of them. Note that these benchmarks are encoded from real-world applications or mathematical problems of importance. Our results show that local search solvers can be complementary to CDCL solvers in applications.

We also calculate the overhead of CnC in SLS+CnC solvers. Averaging over all instances, the run time of CnC occupies about 1% run time of the whole process.

7 Conclusions

This work presented an effective method named construct-and-cut (CnC for short) for generating initial assignments for local search SAT solvers. Our experiments on three benchmarks from real-world project and mathematical problems showed that, the CnC method can significantly improve the performance of local search SAT solvers on the benchmarks. More encouragingly, one CnC-enhanced local search solver *CCAnr+cnc* outperformed state-of-the-art CDCL solvers on these benchmarks. The source code of *CCAnr+cnc* is available at `https://github.com/caiswgroup/CNC-LS`.

References

1 Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. Boosting local search thanks to CDCL. In *Proceedings of LPAR 2010*, pages 474–488, 2010.

2 Adrian Balint and Andreas Fröhlich. Improving stochastic local search for SAT with a new probability distribution. In *Proceedings of SAT 2010*, pages 10–15, 2010.

3 Adrian Balint, Michael Henn, and Oliver Gableske. A novel approach to combine a SLS- and a DPLL-solver for the satisfiability problem. In *Proceedings of SAT 2009*, pages 284–297, 2009.

4 Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In *Proceedings of SAT 2012*, pages 16–29, 2012.

5 Armin Biere. Splatz, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT competition 2016. In *Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions*, pages 44–45, 2016.

6 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. In *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*, pages 50–53, 2020.

7 Shaowei Cai, Chuan Luo, Jinkun Lin, and Kaile Su. New local search methods for partial MaxSAT. *Artificial Intelligence*, 240:1–18, 2016.

8 Shaowei Cai, Chuan Luo, and Kaile Su. CCAnr: A configuration checking based local search solver for non-random satisfiability. In *Proceedings of SAT 2015*, pages 1–8, 2015.

9 Shaowei Cai, Chuan Luo, and Haochen Zhang. From decimation to local search and back: A new approach to MaxSAT. In *Proceedings of IJCAI 2017*, pages 571–577, 2017.

10 Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

11 Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving for program verification. In *Proceedings of CAV 2005*, pages 296–300, 2005.

12 Adnan Darwiche and Knot Pipatsrisawat. Complete algorithms. In *Handbook of Satisfiability*, pages 99–130. IOS Press, 2009.

13 Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

14 Andreas Fröhlich, Armin Biere, Christoph M. Wintersteiger, and Youssef Hamadi. Stochastic local search for satisfiability modulo theories. In *Proceedings of AAAI 2015*, pages 1136–1143, 2015.

15 Oliver Gableske and Marijn Heule. EagleUP: Solving random 3-SAT using SLS with unit propagation. In *Proceedings of SAT 2011*, pages 367–368, 2011.

16 Marijn J. H. Heule. Schur number five. In *Proceedings AAAI 2018*, pages 6598–6606, 2018.

17 Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In *Proceedings of SAT 2016*, pages 228–245, 2016.

18 Edward A. Hirsch and Arist Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1):91–111, 2005.

19 Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

20 Chu Min Li and Yu Li. Satisfying versus falsifying in local search for satisfiability. In *Proceedings of SAT 2012*, pages 477–478, 2012.

21 Chu Min Li and Yu Li. Description of Sattime 2013. In *Proceedings of SAT Competition 2013 : Solver and Benchmark Descriptions*, pages 77–78, 2013.

22 Xiao Yu Li, Matthias F. M. Stallmann, and Franc Brglez. A local search SAT solver using an effective switching strategy and an efficient unit propagation. In *Proceedings of SAT 2003*, pages 53–68, 2003.

23 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Proceedings of SAT 2016*, pages 123–140, 2016.

24 Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

25 Chuan Luo, Shaowei Cai, Kaile Su, and Wenxuan Huang. CCEHC: An efficient local search algorithm for weighted partial maximum satisfiability. *Artificial Intelligence*, 243:26–44, 2017.

26 Chuan Luo, Shaowei Cai, Kaile Su, and Wei Wu. Clause states based configuration checking in local search for satisfiability. *IEEE Transactions on Cybernetics*, 45(5):1014–1027, 2015.

27 Chuan Luo, Shaowei Cai, Wei Wu, Zhong Jie, and Kaile Su. CCLS: An efficient local search algorithm for weighted maximum satisfiability. *IEEE Transactions on Computers*, 64(7):1830–1843, 2015.

28 Chuan Luo, Shaowei Cai, Wei Wu, and Kaile Su. Double configuration checking in stochastic local search for satisfiability. In *Proceedings of AAAI 2014*, pages 2703–2709, 2014.

29 Chuan Luo, Holger H. Hoos, and Shaowei Cai. PbO-CCSAT: Boosting local search for satisfiability using programming by optimisation. In *Proceedings of PPSN 2020*, pages 373–389, 2020.

30 Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In *Proceedings of IJCAI 2017*, pages 703–711, 2017.

31 Norbert Manthey. Coprocessor 2.0 – A flexible CNF simplifier – (tool presentation). In *Proceedings of SAT 2012*, pages 436–441, 2012.

32 Neil Newman, Alexandre Fréchette, and Kevin Leyton-Brown. Deep optimization for spectrum repacking. *Communications of the ACM*, 61(1):97–104, 2018.

33 João P. Marques Silva and Karem A. Sakallah. GRASP – A new search algorithm for satisfiability. In *Proceedings of ICCAD 1996*, pages 220–227, 1996.