SN Computer Science (2021) 2:105
https://doi.org/10.1007/s42979-021-00476-0

ORIGINAL RESEARCH

An Efficient Local Search SAT Solver with Effective Preprocessing for Structured Instances

Md Shibbir Hossen[1] · **Md Masbaul Alam Polash**[2]

Received: 5 April 2020 / Accepted: 19 January 2021 / Published online: 20 February 2021

Abstract

Developing an efficient solver for different NP-complete problems such as propositional satisfiability (SAT) is very complicated and often takes a lot of time. A wide range of problems in different areas of computer science and artificial intelligence can be solved using SAT solvers. The SAT problem is defined as finding a logical assignment that satisfies all clauses in a Boolean formula. The recent developments of different stochastic local search (SLS) SAT solvers present various new heuristics and solving strategies. In this paper, we present an SLS-based SAT solver for structured instances that includes an efficient preprocessing technique along with a few other heuristics. We first remove all equivalence from the SAT formula and then perform searching. Experimental outcomes depict that our new solver can solve some unsolved instances of the state-of-the-art solver; for other benchmarks, our new one also responded quickly.

Keywords SAT · Configuration checking (CC) · Stochastic local search

Introduction

The Boolean Satisfiability problem (SAT) is a Constraint Satisfaction Problem (CSP) which is the problem of determining whether a Boolean formula can be satisfied by a logical (i.e. *true*, *false*) assignment of variables. In recent years, captivating improvements in SAT attract us to study further about this NP-complete problem. The attraction behind the SAT problem is the extensive application in real-world problems and theoretical aspects [1]. A framework of the SAT model is being used to solve many combinatorial problems. It is now routinely being used in bounded model checking (BMC) [2], circuit verification [3], test pattern generation and more specifically in checking complex circuits [4]. Recently, a good number of solvers have been programmed which have better run-time. Therefore, many complex problems can now be solved. In recent years, SAT solvers are successfully being used to solve different types of problems. Those solvers are now being used in artificial intelligence and optimization, more specifically in scheduling and planning problems [5], bioinformatics [6], probabilistic models, crypto-analysis [7], etc. Every year, new solvers with various encouraging techniques are introduced. Also, the SAT community all over the world arranges SAT competitions[1] to share ideas and new solvers and benchmarks.

There are many popular solvers available for SAT. So far we have studied, these SAT solvers can be categorized into two popular types. The first one is the systematic approach solvers, e.g., Conflict Driven Clause Learning (CDCL) [8], Davis-Putnam-Logemann -Loveland (DPLL) [9], etc. Here, they use a complete search strategy to solve SAT. In contrast to procedural approaches, stochastic local search (SLS) solvers are also preferred and it has a dominant performance over complete solvers on different SAT instances. It produces a better success rate in random SAT formulas. SLS solvers are mostly preferred for their speed and scalability. Therefore, mighty research and influential efforts are going on to develop more efficient SAT solvers. There are both strengths and weaknesses of those approaches. Most of the modern solvers perform unit propagation as a preprocessing phase [10]. Few other variable and clause elimination techniques are also available [11–13]. Our approach is tantamount to

✉ Md Shibbir Hossen
shibbir.hossen@live.com

Md Masbaul Alam Polash
mdmasbaul@gmail.com

1 Department of Computer Science and Engineering, Jagannath University, Dhaka, Bangladesh

2 School of Computer Science, University of Sydney, Sydney, Australia

1 https://www.satcompetition.org/.

those techniques in the case of preprocessing. It can be seen as part of improving the performance of modern SAT solvers to accelerate computational power.

In this paper, an improved version of SLS-based SAT solver CCAnrEQ is presented. This solver is mainly inherited from the state-of-the-art solver CCAnr [14], an SLS-based solver. Our thesis aims to present a simplification technique that enhances the SAT solver performance. This simplification process is done by restoring the gate structure from the CNF [15, 16]. The extraction of gates gives us directions about how the CNF encodes from the SAT. And so, it helps to exploit the intrinsic properties and to research before performing the satisfaction of the CNF formula. Our observation is encoding and restoring the gates which are the reflection of one another. We simplify the CNF by restoring gates, because it eases the SAT solver to reduce rambling formulas. This process is much easier than encoding as it is domain independent. The simplification, further improvements, and detailed experimental evaluation of our new solver help us to understand why the new method works.

As we describe above, our simplification technique can be employed during preprocessing. We focus on employing this reasoning as a preprocessor only. When restarts during SAT solving, the solver will use the simplified CNF always. We have added a rich literature review for SLS solvers along with CCAnr. Here, we present the local search strategy and the different shortcomings of the SAT solvers. The existing CCAnr solver performed very well in different SAT competitions in various tracks. Therefore, we try to improve it. These are helpful to build our new solver CCAnrEQ. We empirically demonstrate in the experimental result section that our new solver performs better than the existing one in different outlooks on several structured SAT instances. More specifically, our new solver can solve some new instances that cannot be solved by CCAnr with a specific time cut-off. Therefore, in both success rate and time comparison, our demonstrated solver outperforms CCAnr. On the other hand, we have also tried different heuristics to reduce greediness along with this preprocessing. Those heuristics also produce several good results that are shown in "Experimental Result" section.

Our paper is organized in the following way. "Related Works" section provides a study related to our research. Here, we also describe different SLS solvers along with the background of the state-of-the-art solver. Based on the previous experience, we present the approaches and implementation technique of our new solver *CCAnrEQ* in "Our Approach" section. In "Experimental Result" section, we show the comparison between our solver and the State-of-the-art solver. A general conclusion and an outlook on future works are added in "Conclusion" section.

Related Works

In this section, we first describe the basic notations related to the SAT problem. Then, a few definitions and terminologies related to this problem. After that, various existing techniques related to local search and SAT are described.

SAT Definitions

Given a set of propositional variables $V = \{x_1, x_2, x_3, ..., x_n\}$, where a *literal* can be either positive (the variable itself x_i) or negative (negation of a variable is $\neg x_i$). We say, for each variable, the domain size is fixed which is $\{true, false\}$. There are many formats to represent a SAT formula. Among all format, Conjunctive Normal Form (CNF) is very much familiar. A CNF is called clausal formula as it is a conjunction ($\wedge$) of a set of clauses $\{C_1, C_2, ..., C_n\}$. A *clause* is a disjunction ($\vee$) of *literals*. If we say l as a *literal*, a *clause* C_i can be written as $C_i = l_1 \vee l_2 \vee .... \vee l_n$ and $l \geq 1$. Therefore, a CNF formula is written as $F = C_1 \wedge C_2 \wedge ... \wedge C_n$. For a given SAT formula F, a SAT problem can be evaluated as checking whether every *clause* can be satisfied by considering some assignments of truth values to each variable. To satisfy the formula F, every *clause* C_i needs to be *true*.

Let $V(F)$ be the set of all variables. A structure $S : V(F) \rightarrow \{0, 1\}$ is an assignment. At local search, S maps all variables of F to a Boolean value. That is why it is called a complete assignment. In other words, S is a true assignment of 1,0 $\{true, false\}$ of each variable contained in V. If there exists any structure S such that $S(F) = 1$ then the clausal formula F is called *satisfiable*. A CNF will be a tautology if $F(S) = 1$ for every structure of S. There may be some clauses that contain only a literal, which are called a *unit clause*. A CNF is *unsatisfiable* if two *unit clauses* appear as l_i and $\neg l_i$.

A SAT problem consists of m variables and n clauses (i.e. $|V| = m, |C| = n$). The clause-to-variable ratio r is defined as $r = n/m$ (e.g. $r = 2.00001$ means each clause has two literals on average). The variables can be the neighbor of each other. A variable will be the neighbor of another variable if both appear in at least one clause. Let a and b be two variables. The neighborhood of variable a is defined as $N(a) = \{b | b$ appears in any clause simultaneously with $a\}$, where $N(a)$ contains all neighbors of a. In this paper, we use all those terms as required.

Local Search Solvers

From the early '90s, SAT is becoming much more popular because of its various practical applications. Therefore, many state-of-the-art solvers are found from that time. Every year, the SAT community celebrates an idea-sharing competition where a lot of solvers compete. That gives more opportunities to find more solvers and to enhance the solver's performance. There are many complete solvers [8, 17–20] that are mainly tree-based method to solve the SAT. In this paper, we prefer to go ahead with local search solvers [14, 21–28] only as it is much more scalable. Solvers that use a pre-processor [12, 29, 30] also perform well.

GSAT [22] is the first stochastic local search (SLS) solver. SLS solvers start with a random complete assignment that is called the initialization phase. After that, it performs the basic local search approach. It continually takes a variable to be flipped and does this until an intended assignment is found or some other terminating conditions (e.g. time out, maximum iteration, etc.) is reached. Selecting the variable to be flipped is a crucial issue and many heuristics are applied here. In GSAT, a variable with the highest *score*, see Definition 1, is picked. As SLS solvers start with complete assignments, it is more scalable than the complete solver. But it can not guarantee whether the SAT formula can be satisfied.

Definition 1 *score* : The *score* of a variable is the difference between *make* and *break* after flipping it. The *break* is the count of clauses that become unsatisfied if that variable is flipped. On the other hand, the *make* is the number of clauses that become satisfied if that variable is flipped.

Other types of solvers are based on WalkSAT [27, 28, 31]. As GSAT is a completely greedy approach, WalkSAT introduced noise parameters that reduce some greediness. It is always challenging to set the noise parameters and to tie-breaking in the variable selection phase.

The cycling problem is one of the greatest problems in local search [32] and it hinders search performance. Several strategies (e.g. Tabu [33]) are proposed to assuage it. It is also a big concern in the field of SAT. A novel strategy named *configuration checking* (CC), see Definition 2, is successfully applied in different problems [34, 35]. This is also applied in different SAT solvers named CCASat [25], swcc [36], CCAnr [14]. Similarity checking, one of our previous works, is applied to reduce the problem [37]. This approach is not integrated into this paper as it does not perform well after removing equivalence.

Definition 2 *configuration checking* (*CC*) : it is a way to prevent a variable to be flipped until one of its neighbor variables is changed (flipped) since the last flip of that variable.

The CCAnr Solver

At CCAnr [14], two types of variables are introduced. One is CCD variable, see Definition 3, and another is SD variable, see Definition 4. A *pickVar* function in CCAnr returns a variable to be flipped. According to Algorithm 1, it checks first whether a *CCD* variable exists and if it is found then returns the one that has the highest *score*. Then, it checks again whether a *SD* variable exists and does the same process. Here, searching for a *SD* variable can be described as an aspiration technique in local search. This is called CCA heuristic in CCAnr.

If none of these variables are found, then it performs a focused random walk as diversification mode. Here, a clause weighting technique is applied to diversify the search. The core idea is to increase the weight of the falsified clause, while in a stagnant situation to diversify search. At CCAnr, first, it increases the weight of every unsatisfied clause by one. After that, the average clause weight $\overline{w}$ is calculated. If $\overline{w}$ is greater than a specific threshold γ, then all clause weights are smoothed as $w(c_i) := \lfloor \rho \cdot w(c_i) \rfloor + \lfloor (1-\rho)\overline{w} \rfloor$. Here, γ and ρ are the threshold parameter and the factor parameter respectively. Finally, it picks a random unsatisfied clause and returns the most aged variable.

CCAnr is a two-mode SLS solver. One is a greedy mode where *CCD* or *SD* variables are searched and another one is diversification mode. Some other two-mode SLS solvers can be found in [21, 38, 39]. At CCAnr, *unit propagation* is included only as preprocessing. It takes a little amount of time to preprocess. Therefore, the overall solver performance is not degraded. Now, the performance can be improved by executing efficient preprocessing. In our paper, we have selected the CCAnr to make further improvements.

Definition 3 *configuration changed decreasing* (*CCD*) : It is the variable whose *configuration* has been changed (i.e. not locked by CC) and it has a positive *score* (i.e. *make* is greater than *break*).

Definition 4 *significant decreasing* (*SD*) : A variable v that *configuration* is not changed (i.e. locked by CC) but has a significant *score* where $score(v) > \overline{w}$. The average clause weight is $\overline{w}$.

Algorithm 1 CCAnr $pickVar()$ function

```
   Output: A variable to be flipped next.
1: if a CCD variable then
2:     return a CCD variable with the highest score
3: if a SD variable then
4:     return a SD variable with the highest score
5: update_clause_weight()
   /*Focused random-walk mode*/
6: pick a random unsatisfied clause c
7: return the variable having the highest score in c
   *select the least flipped one for tie breaking
```

Further Study

Path-Relinking (PR) [40], an intensification strategy, is another efficient technique and performs well in MAX-SAT [41]. Since path-relinking is well known for the different optimization problems, it cannot be applied directly in MAX-SAT. A greedy randomized adaptive technique is incorporated with path-relinking to improve search in MAX-SAT [42, 43]. IPBMR (Iterated Path-Breaking with Mutation and Restart) is another well-known mechanism that outperforms different popular solvers [44]. Here, the difficulties of incorporating path-relinking directly to MAX-SAT are described. Recent work suggests that parallel portfolio based local search works very well and the intensification and diversification states are controlled by path-relinking [45]. To the best of our knowledge, it has not been applied to structured instances of SAT. Therefore, we decide to implement a new strategy similar to PR but not in an exact way to check the efficiency of it. A basic idea of PR is described in Definition 5.

Definition 5 Path-Relinking (PR): A strategy to find new quality solutions by connecting two high-quality solutions: one is a guiding solution and another is an initial solution.

In the local search method, greediness is a very common phenomenon here. A lot of sequences of operations are done greedily. But the search performs better at the beginning but falls later. This is a typical scenario in the max–min search. Therefore, while performing the search greedily, it goes to a local optimum in a short time. But most of the time, it becomes stuck to reach the global optimum. Therefore, many strategies are applied here (e.g. simulated annealing [46], clause weighting scheme [14, 38] variable weighting [47], etc.). The study of the greedy method on SAT can be found on [48]. Therefore, adding some randomness is a solution to reduce complete greediness. Monte–Carlo method [49] is a useful method to solve different problems class like optimization. It uses randomness to solve the problem. First, the basic trend of this method is to generate inputs randomly from a set of possible inputs using a probability distribution. Second, it performs a deterministic calculation on the inputs. At last, the results of these calculations are combined to get the overall result.

Our Approach

In this section, we describe how to detect and remove equivalence from the CNF. Then, some other heuristics are described that can improve the solver performance.

Equivalence Removal

Suppose that, we have the following two clauses:

$C_1 : a \vee \neg b$

$C_2 : \neg a \vee b$

Now, to satisfy clause C_1, we can put $a = 1$ or $b = 0$. But if we take $a = 1$ then b must be true to satisfy C_2. Similarly, we can also satisfy both clause C_1 and C_2 by putting $a = 0$ and $b = 0$. Therefore, both two variables take the same value and we tell them as equivalent variables. To find the equivalent variables, we can consider only the clauses whose $size = 2$. Algorithm 2 describes how to get those variables and the more about this algorithm can be found in [15].

Algorithm 2 *Equivalent Detection*

$Q = \emptyset$
for each *clause* C_i *in* C *where* $|C_i| = 2$ **do**
  if C_i is marked **then**
    continue
  else
    $l_1 = C_i[0]$
    $l_2 = C_i[1]$
    $x_1 = V[C_i[0]]$ //First variable of C_i
    $x_2 = V[C_i[1]]$ //Second variable of C_i
    if both literals (l_1, l_2) are negative or positive **then**
      continue
    $S_1 = \{ C_p \in C$ where $|C_p| = 2$ & $x_1 \in C_p \}$
    $S_2 = \{ C_q \in C$ where $|C_q| = 2$ & $x_2 \in C_q \}$
    $S_3 = S_1 \cap S_2$ //find common clauses
    for each *Clause* C_j *in* S_3 **do**
      $m_1 = C_j[0]$
      $m_2 = C_j[1]$
      if $(\neg l_1 = m_1 \text{ and } \neg l_2 = m_2)$ or $(\neg l_1 = m_2 \text{ and } \neg l_2 = m_1)$ **then**
        mark(C_j)
        Found two variables of $V[C_i]$ are equivalent
        $Enqueue(Q, (x_1, x_2))$ //record for future
        break
    mark(C_i)

To find the equivalences, only two literal clauses are considered in our approach. An example of two literal clauses is described before. As per example, from lines 2 to 4, Algorithm 2 considers only unmarked clauses (e.g. a clause is marked when the variables of that clause are equal, while finding equivalence and that clause will not be used in the future) from all clauses that have two literals only. Whenever two variables are found to be equal, two corresponding clauses will be marked so that those two can not appear for future detection. If it is found any unmarked clause, line 6 and line 7 will store those two literals, respectively. At line 10, it checks whether two literals l_1 and l_2 are positive or negative as we need only different literals. At line 12, S_1 is the set of all clauses which has only two literals and a literal of variable x_1 and line 13 does the same for the variable x_2. Therefore, if an intersection is performed here then it produces a set that will contain the clause numbers which have both literals of variables x_1 and x_2. And at line 14, S_3 is such a set of common clauses. Now, if the clause C_i has the two literals $(\neg l_1, l_2)$ then a second clause C_j with literals $(l_1, \neg l_2)$ is required to treat them as equivalent. Similarly, if the clause C_i is $(l_1, \neg l_2)$ then a second clause C_j with literals $(\neg l_1, l_2)$ is needed. The searching for such kind of the second clause is done from lines 15 to 20. At line 18, the second such clause is searched, and once found then C_j will be marked. Once we confirm equality, both variables will be pushed into a queue as a pair to find equivalent sets.

A lot of variables can be equivalent to each other. For example, if we have two equalities $x_1 = x_2$ and $x_2 = x_3$, then we can deduce that $x_1 = x_3$. This is the transitive property of equality. It is written as follows:

If $a = b$ and $b = c$, then $a = c$.

Finding transitive equivalence is the key to remove equivalence from the CNF. We use a new data structure to find the equivalence chain. Algorithm 3 describes more to find transitive equivalence. At first, we set the parent of a variable as the variable itself. At line 4 and line 5, the first equivalent variable is popped from the queue. Two equivalent variables should have the same parent and they belong to the same set. Here, *parent_u* and *parent_v* at line 7 and line 8 stores the parent of two variables *u* and *v*, respectively. Now, if both the parents are not equal, then we will check for all variables parent. From all variables, if the parent of a variable is the same as *parent_v* then we will replace it with *parent_u*. And therefore, we will get the same parents for all variables that belong to the same set. Algorithm 3 is time-consuming as all variable's parents are checked at line 10–12 when two variables are dequeued from the queue. Here, an efficient algorithm can be used to reduce the equivalent set finding. The disjoint set concept, one of the best ways to find disjoint sets, can be used here as an alternative to our algorithm.

Algorithm 3 *Find Equivalent Sets(Q)*

```
1:  for each variable v_i ∈ V do
2:      parent[v_i] = v_i
3:  while Q not empty do
4:      u = Q.top().first
5:      v = Q.top().second
6:      dequeue(Q)
7:      parent_u = parent[u]
8:      parent_v = parent[v]
9:      if parent_u ≠ parent_v then
10:         for i = 1 to |V| do
11:             if parent_v = parent[i] then
12:                 parent[i] = parent_u
```

In this research, removing equivalent variables before searching for the solution is the key point. There are many cases to remove equivalence from the CNF. Those cases are pointed bellow:

Case 1: If two-variable (e.g., a, b) are equal and a clause is $(\neg a,\ b)$ or $(a,\ \neg b)$, then we can delete the clause. Because if we put $b = a$ in the clause then the clause looks like $(\neg a,\ a)$ which is always *true*. Similarly, a clause, of size larger than 2, that contains $(a,\ \neg a)$ is also *true*.

Case 2: Let us consider a clause, size 2, is $(a,\ b)$ or $(\neg a,\ \neg b)$. Now if we put $b = a$ in that clause then it looks like $(a,\ a)$ or $(\neg a,\ \neg a)$ which results in a unit clause. As $(a,\ a) = a$, we can select that clause as a unit clause for *unit_propagation*.

Case 3: When two variables are equivalent but Case 1 and Case 2 does not occur, Case 3 is introduced here. Here, the second variable is replaced with the first one and clause size is reduced by one.

Algorithm 4 *RemoveEquivalence*

```
1: for each v_i in V do
2:     if v_i = parent[v_i] then
3:         continue
4:     eq_var = v_i
5:     replace_var = parent[v_i]
6:     for each clause c_i in C & eq_var ∈ c_i do
7:         if delete(c_i) then
8:             continue
9:         position_of_eq_var = −1
10:        for j = 0 to |c_i| do
11:            if c_i[j] = eq_var then
12:                position_of_eq_var = j
13:                break
14:        isReplace_var = false
15:        for j = 0 to |c_i| do
16:            if c_i[j] = replace_var then
17:                isReplace_var = true
18:                if polarity(c_i[j]) ≠ polarity(eq_var) then //a -a is found
19:                    delete[c_i]
20:                else
21:                    remove(c_i[j])
22:                    if |c_i| = 1 then
23:                        unitClause(c_i)
24:                break
25:        if isReplace_var = false then // no replace variable
26:            c_i[position_of_eq_var] = replace_var
```

Algorithm 4 describes how to remove equivalence from the CNF. At line 6–8, it checks whether a clause, that contains equivalent variable *eq_var*, is deleted or not. If clause c_i is not deleted then the position of *eq_var* is picked by performing lines 10–13. At line 16, it checks whether a replace variable is also contained in c_i that means both *eq_var* and *replace_var* is present in c_i. Case 1 and Case 2 are introduced at line 18–19 and line 20–23, respectively. From lines 25 to 26, Case 3 is applied.

Step Re-linking

Path relinking (PR) strategy is a well-known concept in many problems, especially in MAX-SAT. This strategy is mainly used to intensify the search to obtain new trajectories by connecting two local minima. Those saved local minima, found previously by performing different search strategies (e.g. Tabu, Simulated Annealing, etc.), are called elite solutions. Two elite solutions, named *initial* and *guiding*, are picked randomly and perform path relinking. Therefore, if the distance between those solutions is bigger then we can conclude that a lot of space has been ignored and path relinking tries to relink two solutions that produce a new solution that may be better. At the relinking process, only uncommon elements are considered and tried to change. In this paper, we have tried to implement a new strategy named step relinking (SR) which is inspired by PR. Here, we save the local minima as elite solutions. When the search falls in local minima or stagnant situation, we try to diversify the search. But at that time, most of the clauses are satisfied and it is a draconian task to find the best assignment. Here, when the search steps into diversification state, we save the solution, and after a certain number of steps if the search cannot get rid of that state, we perform a relinking to get a better solution. Now, we describe how this strategy is added to CCAnr and differs from the typical way to implement it.

As PR is applied between two solutions, we first say how elite solutions are picked during the search. Algorithm 5 describes the procedure of adding solutions in an *elite* array. We save the current solution along with the step or iteration number while our search is stuck. The *elite_pointer* indicates the last solution index *elite* array and we save few recent local minima always. At line 1, if *elite_pointer* is at *max_elite_pointer*, the highest number of solutions we want to save as elite solution, then we confirm that we have enough solutions. The *elite_memory_full* is made true when we have stored the maximum number of solutions. To save the most recent local minima, we start *elite_pointer* from 0 or begin like a circular queue. Now, it will replace the most aged elite solutions. At line 6–7, the solution is saved at *elite* array. An *elite_iteration* array, at line 8, saves the current iteration number of the saved solution. We need to save the iteration number, because we do not want to save all stagnant solutions.

Algorithm 5 *add_solution(cur_soln[ ])*

```
if elite_pointer == max_elite_pointer then
    elite_pointer = 0
    elite_memory_full = true
else
    elite_pointer ++
    for each v in |V| do
        elite[elite_pointer][v] = cur_soln[v]
    elite_iteration[elite_pointer] = step // 'step' indicates the iteration number
```

Now, we describe how the SR algorithm works and Algorithm 6 describes the main procedure. The previous solutions named *initial* and a *guiding* solution are needed to relink. We call *cur_soln* as the initial solution and a *guiding* solution is picked randomly from the *elite* array. At line 1, if we have fewer solutions than the maximum number of solutions then we pick a solution randomly within that range. Otherwise, at line 4, we pick randomly from all solutions. While performing the relinking process, first we save the solution. After that, a solution from the *elite_array* is picked randomly to perform relinking with the current one. Therefore, a randomly picked solution can be the current one that's why line 5–6 is introduced. Now we check all variables of the *initial* and *guiding* solution and pick only uncommon variables value as the PR strategy. Hence, if a variable has no similar value, then we check the *score* of that variable. After checking all the variables, we return a variable that has the highest *score*.

Algorithm 6 *step_relinking()*

```
1: if !elite_memory_full then
2:     guideSoln = rand()%(elite_pointer)
3: else
4:     guideSoln = rand()%(max_elite_pointer)
5:     while elite_iteration[guideSoln] = step do // step indicates current iteration number
6:         guideSoln = rand()%(max_elite_pointer)
7: max_score = −1
8: best_var = 0
9: for each v in V do
10:    if elite[guideSoln][v] ≠ cur_soln[v] then
11:        if score[v] > max_score then
12:            max_score = score[v]
13:            best_var = v
14: return best_var
```

SR approach with CCAnr is described in Algorithm 7. Before that, adding solutions to the *elite_array* and the SR process is described. Here, we perform the SR approach during diversification state (e.g. focused random-walk mode). At line 6, it checks an interval of last saved solutions to avoid every stagnant solution. SR can not be performed without enough elite solutions. That is why line 8 is introduced. As the SR approach is a time-consuming task, we do not perform it in every step of the diversification state. After at least γ iterations, we perform it.

Table 1 The detailed description of benchmarks

Serial	Instance type	Number of instances
1	parity_games	11
2	parity16	5
3	QuasiGroup (QG)	4
4	BMC	4

Table 2 Effects of Equivalence Removal

Instance	Vars	Clauses	After unit Prop.		After Eq. remove		Fixed % after EQ.	
			FV	DC	FV	DC	FV %	FC%
par16-1.cnf	1015	3310	408	1466	273	546	44.98	29.61
par16-2.cnf	1015	3374	383	1416	265	530	41.93	27.07
par16-3.cnf	1015	3344	395	1440	272	544	43.87	28.57
par16-4.cnf	1015	3324	396	1442	278	556	44.91	29.54
par16-5.cnf	1015	3358	388	1426	268	536	42.74	27.74
bmc-ibm-1.cnf	9685	55870	2600	20437	722	1451	10.19	4.09
bmc-ibm-2.cnf	2810	11683	1666	8357	348	704	30.41	21.16
bmc-ibm-5.cnf	9396	41207	4533	24457	1286	2580	26.44	15.40
bmc-ibm-7.cnf	8710	39774	4844	25388	911	1830	23.56	12.72
qg5-11.cnf	1331	64054	507	35566	76	152	9.22	0.53
qg6-09.cnf	729	21844	340	14191	18	36	4.62	0.47
qg7-09.cnf	729	22060	395	16222	28	56	8.38	0.95
qg7-13.cnf	2197	97072	785	51105	48	96	3.39	0.20
n3_i3_pp_ci_ce.cnf	525	2336	–	–	136	399	25.9	17.08
n3_i3_pp.cnf	525	2276	–	–	120	339	22.85	14.89
n4_i4_pp_ci_ce.cnf	1572	9175	–	–	363	1092	23.09	11.90
n4_i4_pp.cnf	1572	9007	–	–	318	924	20.22	10.25
n5_i5_pp.cnf	3655	726264	–	–	655	1940	17.90	7.38
n5_i6_pp_ci_ce.cnf	4380	3198	–	–	900	2764	20.54	8.64
n5_i6_pp.cnf	4380	31540	–	–	784	2324	17.89	7.36
n6_i6_pp_ci_ce.cnf	7278	63935	–	–	1342	4164	18.43	6.51
n6_i6_pp.cnf	7278	63275	–	–	1167	3504	16.03	5.54
n6_i7_pp_ci_ce.cnf	8484	7464	–	–	1564	4863	18.434	6.51
n6_i7_pp.cnf	8484	73860	–	–	1359	4083	16.01	5.52

FV fixed variables, *DC* deleted clauses

Algorithm 7 *pickVar* fuction in CCAnr+SR

if *CCD variable is found* **then**
  return a *CCD variable* with the highest *score*
if *SD variable is found* **then**
  return a *SD variable* with the highest *score*
update_clause_weight()
/*Focused random-walk mode*/
if $step - elite\_iteration[elite\_pointer] > \gamma$ **then**
  add_solution(*cur_soln*[ ])
  if *elite_pointer* > 0 *or elite_memory_full* **then**
    best_var = *step_relinking*()
    return *best_var*
  else
    pick a random unsatisfied clause *c*
    return the variable with the highest *score* in *c*
else
  pick a random unsatisfied clause *c*
  return the variable with the highest *score* in *c*
*select the least flipped one for tie breaking

Random Sampling (RS) Approach

As mentioned earlier, in *pickVar* function returns a variable to be flipped, it first checks whether a *CCD* variable exists. If *CCD* variables exist, it returns the variable that has the highest score. Therefore, a variable is selected greedily from the *pickVar* function. Here, we want to introduce some randomness rather than selecting a variable greedily as the greedy approach does not guarantee to reach an optimal solution. As a lot of *CCD* variables can exist during the search, we will take a random sample within a certain probability. And this method is related to the Monte Carlo method [49] where a

Table 3 Time required to remove equivalence from CNF

Instance type	Removing time (s)
parity_games	0.06
BMC	0.08
parity16	0.006
QuasiGroup (QG)	0.075

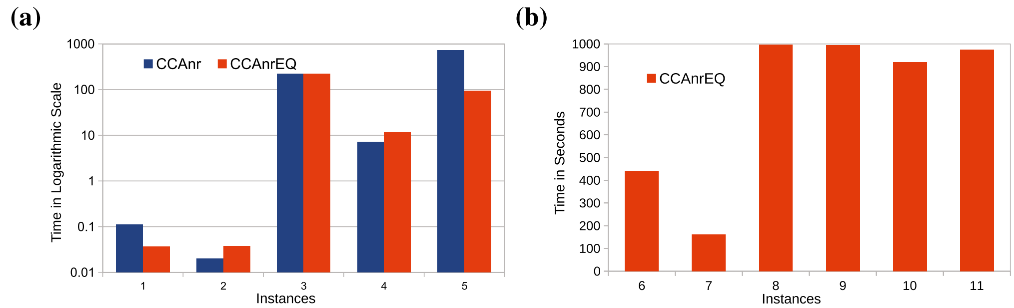


Fig. 1 Time comparison in parity games instances with equivalence removal

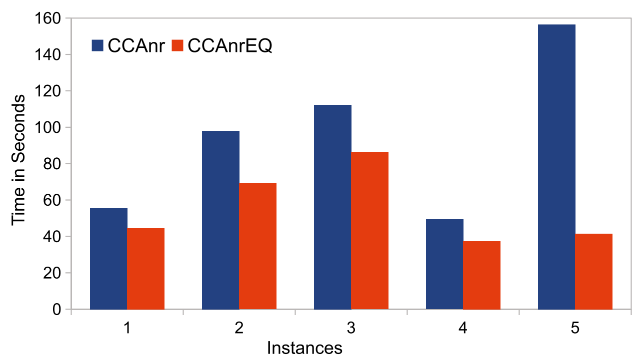


Fig. 2 Time comparison in parity16 instances with equivalence removal

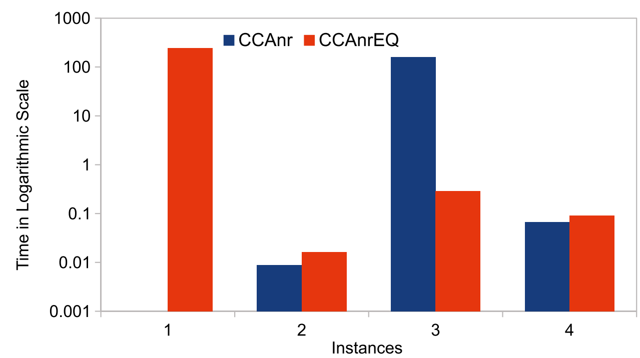


Fig. 3 Time comparison in BMC instances with equivalence removal

random sample is taken from the set of all *CCD* variables. Now, we will describe how the RS method is co-related with the Monte Carlo method [49].

Algorithm 8 $pickVar$ fuction in CCAnr+RS

if $CCD\ variable\ is\ found$ **then**
  if within certain probability **then**
    Take α random CCD variables
    return the $variable$ with the highest $score$ from α
  else
    return a $CCD\ variable$ with the highest $score$
*select the least flipped one for tie breaking

Although the Monte Carlo method varies with each problem, it follows a basic pattern always. First, it defines a set of possible inputs and at Algorithm 8, *CCD* variables are the inputs. Second, a random sample is taken from the full probability distribution and it checks a probability to take the random set. After that, it performs a deterministic approach to get the result. We have done this also after taking α random variables. We select the variable which holds the highest score.

Experimental Result

We have chosen $C++$ to implement all of our algorithms that are described earlier and then compiled with $g++$ with $-O3$ options. All of our experiments are performed on a GNU/Linux machine, having four cores of intel i7 @2.40GHz and 8 GByte RAM. We run each of the instances ten times with a time cut-off of 1000 s. All of our searching time, shown in the figure or table, is considered as the average of ten runs.

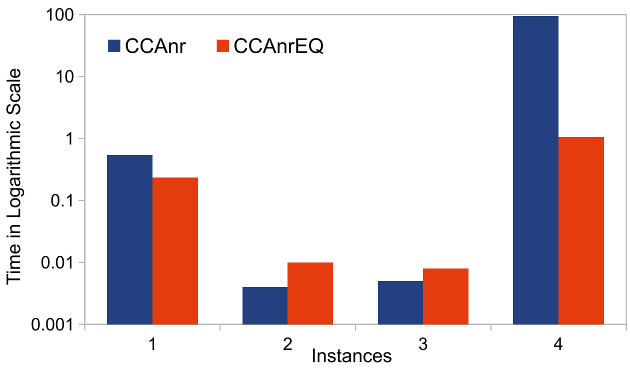
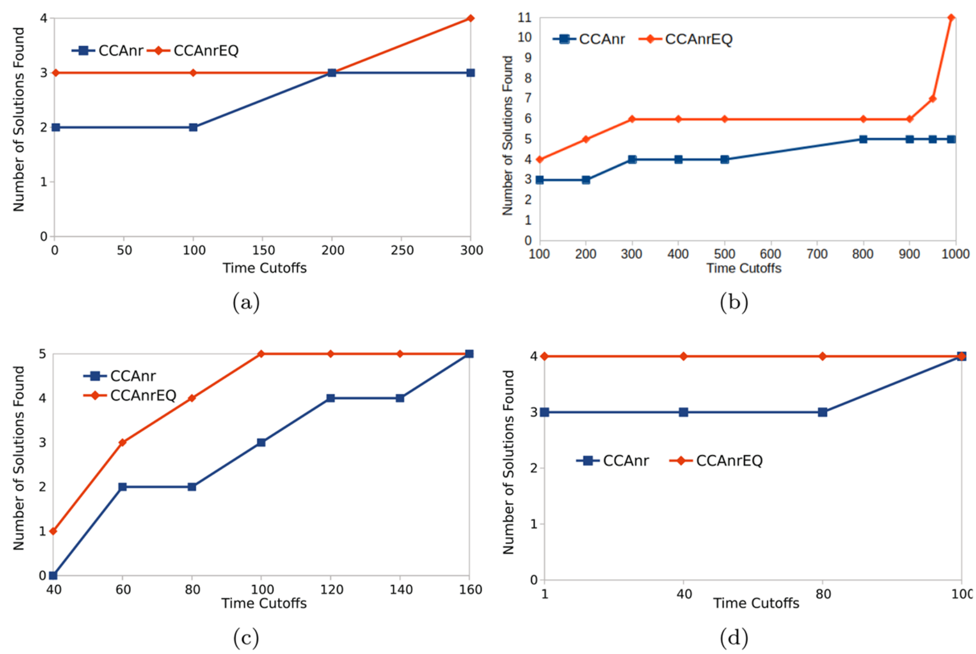


Fig. 4 Time comparison in QG instances with equivalence removal

Table 4 Success rate and time comparison of different types of instances

Instance	CCAnr		CCAnrEQ	
	Success rate (%)	Time (avg)s	Success rate (%)	Time (avg)s
parity_games	45	633.32	**100**	**437.98**
BMC	75	290.10	**100**	**61.08**
parity16	100	94.30	100	**55.76**
QuasiGroup (QG)	100	23.801	100	**0.37**

Fig. 5 Number of solutions found within time interval



The Benchmarks

Here, we work with four different types of benchmarks to evaluate our algorithms. The first type, parity_games, is taken from SAT 2010 competition.[2] The other types of instances are collected from SATLIB.[3] Table 1 shows the number of instances of each type. Here, the instances which contained equivalent gates are mostly preferred for the discussion. We have presented the instances that the State-of-Art solver, named CCAnr [14], or our new solvers can solve.

Evaluation of Equivalence Removal

In this subsection, we will first describe the effect of equivalence removal, explained in "Equivalence Removal" section, of a CNF. After that, the effect of other methods, described in subsection 3.3 and 3.2, will be shown as well.

When we remove equivalence from a CNF, a lot of clauses and variables are removed or fixed. A variable is fixed means no further calculation or assignment is needed for that variable while searching. A clause is fixed means that we need not satisfy that clause while employing the search. The number of equivalence gates is mainly problem-specific. Not in every CNF, an equal number of equivalence gates may be found.

Table 2 shows the effect of equivalence removal from a CNF. Let, we have α variables in a CNF. Now, suppose after *Unit_propagation*, U_α variables are removed from the CNF. Therefore, the number of remaining variables will be

[2] https://www.satcompetition.org/.

[3] https://www.cs.ubc.ca/hoos/SATLIB/benchm.html.

Table 5 Comparison with other approaches

Instance	CCAnr		RS		SR	
	Success rate (%)	Time (avg)s	Success rate (%)	Time (avg)s	Success rate (%)	Time (avg)s
parity_games	45	**193.09**	45	247.02	45	201.14
BMC	75	53.44	75	**43.15**	75	77.58
parity16	100	**94.3034**	100	145.19	100	145.66
QuasiGroup(QG)	100	23.8	100	**23.01**	100	32.81

Table 6 Time comparison of RS with equivalence removal

Instance	EQ		EQ+RS	
	Success rate (%)	Time (avg)s	Success rate (%)	Time (avg)s
parity_games	100	**437.98**	100	442.19
BMC	100	61.08	100	**60.01**
parity16	100	**55.76**	100	70.16
QuasiGroup(QG)	100	0.37	100	**0.11**

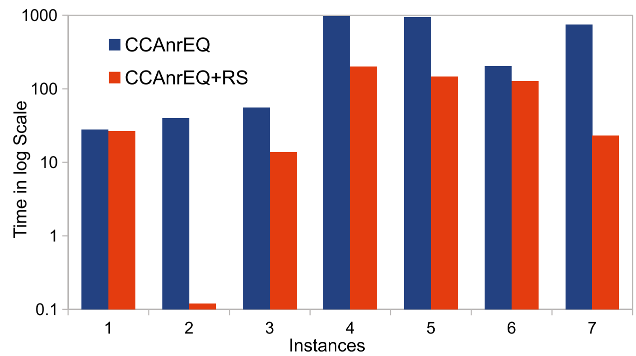


Fig. 6 Time comparison in parity games instances with EQ and RS

$\alpha - U_\alpha$. Again, after performing *equivalence_removal*, suppose E_α amount of more variables are reduced. The percentage of total fixed variables can be derived by Eq. 1.

$$FV = \frac{(\alpha - U_\alpha) - E_\alpha}{(\alpha - U_\alpha)} \times 100\% \tag{1}$$

For example, let us consider the instance par16-1.cnf, the total number of variables is 1015. After performing *Unit_propagation*, the number of variables fixed is 408. The remaining variables are 1015 − 408 = 607. Next employing the *equivalence_removal*, about 273 more variables can be fixed from 607 variables. Therefore, the percentage of equivalence variable reduction is 44.98%. The percentage of fixed variables (FV%) is done concerning the number of variables reduced after performing *Unit_propagation*. But if we consider parity_games instances, we will see that no variable was reduced after *Unit_propagation*. A similar calculation is applied for clause reduction also.

Table 3 depicts the average time required to remove the equivalence from a CNF of the specific type. Here, we can see that a very negligible amount of time is needed to remove equivalence. Therefore, if we add before starting the search, this will not take so much time from the total time. This time can be larger for big formulas. A lot of techniques with efficient data structures can be applied here.

Comparing EQ with the State-of-the-Art Solver

In this subsection, we show the comparison of equivalence removal (EQ) with the State-of-the-Art solver CCAnr [14].

On Parity Games Instances Figure 1 depicts the average time comparison of parity games instances when all equivalences are removed from the CNF. Here, among 11 instances our new solver can solve all the instances whereas CCAnr can solve only 5 instances. Therefore, after removing equivalence, our solver can solve 55% more instances than CCAnr. Figure 1a presents the average time, in logarithmic scale, needs to solve the instances by both solvers. Here, to solve instance number 5, the CCAnr takes 733s whereas our new one takes only 100. Therefore, for instance 5, our solver performs 87% better. We also find that it takes less time in instance 1 and 3. For other instances, our solver is tantamount to the CCAnr. Figure 1b shows the average time taken to solve the instances that are not solvable by CCAnr within the time cut-off. Therefore, it is clear that our new solver CCAnrEQ outperforms CCAnr in this type of instance.

On Parity16 Instances There are five instances at parity16 and the average time comparison is shown in Fig. 2. Here, our solver performs better than CCAnr in the case of time for every instance. In instance 5, CCAnr takes 155s to solve and our new solver takes only 40s which is 75% less time needed. Our solver takes maximum time of 82s to solve instance 3, whereas CCAnr takes 110s. Therefore, it takes 25% less time in case of the worst one. As a result, we can deduce that if we remove equivalence from the problem then the result improves a lot for many instances.

On BMC Instances Figure 3 shows time comparison in logarithmic scale for BMC instances. Here, instance 3 needed 160 s to solve by CCAnr whereas our new solver can solve it within a second. That means our new method takes 99% less time. On the other hand, CCAnr alone can not solve instance 1 anyway within a certain time cutoff.

But after removing equivalence, it can be solved within 200 s. In other instances, our solver is also comparable as those instances need less than a second.

On QG Instances For QG instances, the average time comparison in the logarithmic scale is shown in Fig. 4. Here, if we look at instance number 4 then we can get that our solver takes only a second to solve this whereas CCAnr alone takes 100s. The new one also performs better on instance 1. For other instances, it takes the almost same time to solve those instances.

We want to show here that if we remove equivalence, the search improves a lot as many clauses and variables are reduced. In total, after removing equivalence, our solver can solve 7 more new instances that can not be solved by CCAnr alone, and in other instances, our search performs better in case of time.

Table 4 refers to the success rate and time comparison of different types of instances by both solvers. Here, we can see that after removing equivalence, all those instances can be solved 100%. Most importantly, at parity games instances, the solver performs a huge improvement. Here, our solver solves 100% instances, whereas the CCAnr can solve only 45% of instances. In the case of average time, in seconds, comparison, our solver can solve those instances more quickly. Therefore, our solver outperforms in both the success rate and average time.

Figure 5 depicts the total number of instances that can be solved for different time cutoffs. Here, for every type of instance, we see that our new solver is better regarding any time cutoff. In Fig. 5a, if we give a time cutoff of one second then our solver can solve 75% instances whereas CCAnr can 50%. After that, for a time cutoff of 300 s, the new one can solve 100% instances but CCAnr can not solve it yet. Again, in Fig. 5b, if a time cutoff of 300 s is set then the new one can solve 55% instances whereas CCAnr can solve 36% only. And again, CCAnr can not solve any more instances although more than 800 s is given. In Fig. 5c, for parity16 instances, our solver takes only 100 seconds to solve 100% instances, whereas the other one can solve 60% instances at that time. For a time cutoff of 1 s, our solver can solve 100% instances and CCAnr can solve 75% only.

Further Methods

In this subsection, we explain the impact of some further methods added in the CCAnr [14]. Here, those methods are part of our future improvement. We have added those methods to ensure that it can improve the performance of the solver. We carry out our experiments to evaluate those methods in the same benchmarks.

The RS and SR Methods

The Random Sampling (RS) and the Step Relinking (SR) are described in subsection 3.3 and 3.2, respectively. Those are two alternatives to the greedy heuristics of CCAnr. The comparative results are shown in Table 5. Here, the result table shows the average time in seconds and success rates. In the case of the success rate, both approaches are equal to CCAnr. But in timing constraints, SR is not performing so well. It is taking more time than the other two. Therefore, we have decided not to go further with this approach. On the other hand, if we perform RS, then sometimes it returns a good outcome. Table 5 shows that the RS approach takes lower time in the case of BMC and QG instances.

As we have found that the RS approach performs better in different instances, we have tried it with equivalence removal. Table 6 shows the effect of adding RS with equivalence removal in the case of solving time. Here, we observe that the performance of RS with EQ is better in BMC and QG instances.

Figure 6 presents a time comparison on parity games instances. It is described as the time required to solve the instance after running it several times. In our case, the minimum time needed to solve while running those instances ten times. Here, we observe that in every instance, RS with EQ gets the solution more quickly than EQ. Although in the case of average time comparison EQ with RS is not better, at this point, it is far better. The comparison for other instances of parity games is not shown here, because the time required to solve those instances is very small.

There are two parameters used in SR: the number of elite solutions to be saved or the size of the elite array as *max_elite_pointer* and the number of iterations as γ. We set the parameter *max_elite_pointer* $= 40$ and $\gamma = 100$. On the other hand, we take $\alpha = 20$ numbers of random *CCD* variables in RS.

In summary, the experimental results demonstrate that if we remove equivalence from the problem, not only a lot of variables and clauses are removed only but also the search performs better. The CCAnr solver has a *unit* processing before performing local search approach. Now, we want to add an equivalence removal option here to improve the CCAnr.

Conclusion

In conclusion, this paper proposes a binary equivalence removal technique for SAT which works with the input file to reduce the volume of a CNF. To implement the idea, here two different algorithms, namely *equivalence detection*, and *equivalence removal* are added. The *equivalence detection* part is used to detect whether any equivalent variables exist and the other part removes those equivalences. This technique effectively removes many variables from the CNF and produces a reduced input. Thus, a lot of variables are exempted from assigning during the search.

We examined the performance comparison for different structured SAT instances. Based on this technique, our new solver *CCAnrEQ* can solve many hard structured instances that were unsolved by the existing state-of-the-art solver *CCAnr*. In addition, for other instances, our solver performs significantly faster than its original version.

As a part of future work, we would like to implement equivalence removal for all clauses rather than binary clauses only. An efficient preprocessing can lead to solver performances many times. Therefore, we believe, by extracting other gates (e.g., *AND*, *OR*, *NAND*, etc.) and calculating the value of those gates, this solver can perform better.

Acknowledgements Special thanks to the Information and Communication Technology Division of the Government of People's Republic Bangladesh for student fellowship.

Compliance with Ethical Standards

Conflicts of Interest The authors declare that they have no conflict of interest.

References

1. Michel L, Van Hentenryck P. Localizer constraints. Set. 2000;5(1–2):43–844.
2. Sheeran M, Singh S, Staalmarck G. Checking safety properties using induction and a sat-solver. In: International conference on formal methods in computer-aided design. New York: Springer; 2000. p. 127–44.
3. Smith A, Veneris A, Ali MF, Viglas A. Fault diagnosis and logic debugging using boolean satisfiability. IEEE Trans Computer-Aided Design Integrated Circuits Syst. 2005;24(10):1606–21.
4. Marques-Silva J, Practical applications of boolean satisfiability. In, . 9th international workshop on discrete event systems. IEEE. 2008;2008:74–80.
5. Kautz HA, Selman B et al. Planning as satisfiability. In: ECAI, vol. 92, Citeseer; 1992. p. 359–63.
6. Lynce I, Marques-Silva J. Efficient haplotype inference with boolean satisfiability. In: National conference on artificial intelligence (AAAI). AAAI Press; 2006.
7. Mironov I, Zhang L. Applications of sat solvers to cryptanalysis of hash functions. In: International conference on theory and applications of satisfiability testing. New York: Springer; 2006. p. 102–15.
8. Hoos HH, Stutzle T. Systematic vs. local search for SAT. In: Annual conference on artificial intelligence. New York: Springer; 1999. p. 289–93.
9. Davis M, Logemann G, Loveland D. A machine program for theorem-proving. J Symbolic Logic. 1967;32(1):118. https://doi.org/10.2307/2271269.
10. Sinz C. Towards an optimal cnf encoding of Boolean cardinality constraints. In: International conference on principles and practice of constraint programming. Springer, 2005. p. 827–31.
11. Brafman RI. A simplifier for propositional formulas with many binary clauses. IEEE Trans Syst Man Cybern B (Cybernetics). 2004;34(1):52–9.
12. Bacchus F, Winter J. Effective preprocessing with hyper-resolution and equality reduction. In: International conference on theory and applications of satisfiability testing. Springer, 2003. p. 341–355.
13. Een N, Biere A. Effective preprocessing in sat through variable and clause elimination. In: International conference on theory and applications of satisfiability testing. New York: Springer; 2005. p. 61–75.
14. Cai S, Luo C, Su K. Ccanr: a configuration checking based local search solver for non-random satisfiability. In: International conference on theory and applications of satisfiability testing. New York: Springer; 2015. p. 1–8.
15. Roy JA, Markov IL, Bertacco V. Restoring circuit structure from sat instances. In: Proceedings of international workshop on Logic and synthesis. Citeseer; 2004. p. 663–78.
16. Ostrowski R, Gregoire E, Mazure B, Sais L. Recovering and exploiting structural knowledge from cnf formulas. In: International conference on principles and practice of constraint programming. New York: Springer; 2002. p. 185–99.
17. Audemard G, Simon L. Predicting learnt clauses quality in modern sat solvers. In: Twenty-first international joint conference on artificial intelligence; 2009a.
18. Audemard G, Simon L. Glucose: a solver that predicts learnt clauses quality. SAT Competition; 2009b. p. 7–8.
19. Heule MJH, Kullmann O, Wieringa S, Biere A. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In: Haifa Verification Conference. New York: Springer; 2011. p. 50—65.
20. Sorensson N, Een N. Minisat v113-a sat solver with conflict-clause minimization. SAT. 2005;2005(53):1–2.
21. Li CM, Huang WQ. Diversification and determinism in local search for satisfiability. In: International conference on theory and applications of satisfiability testing. New York: Springer; 2005. p. 158–72.
22. Selman B, Levesque HJ, Mitchell DG et al. A new method for solving hard satisfiability problems. In: AAAI, volume 92, Citeseer; 1992. p. 440–6.
23. Gent IP, Walsh T. Towards an understanding of hill-climbing procedures for SAT. In: AAAI, volume 93. Citeseer, 1993. p. 28–33.
24. Benoist T, Estellon B, Gardi F, Megel R, Nouioua K. Localsolver 1x: a black-box local-search solver for 0–1 programming. For. 2011;9(3):299.
25. Cai S, Su K. Configuration checking with aspiration in local search for SAT. In: AAAI; 2012.
26. Ashiqur RK, Lin X, Holger HH, Kevin L-B. Automatically building local search sat solvers from components. Artif Intell. 2016;232:20–42.
27. Selman B, Kautz HA, Cohen B. Noise strategies for improving local search. In: AAAI, volume 94; 1994. p. 337–43
28. McAllester D, Selman B, Kautz H. Evidence for invariants in local search. In: AAAI/IAAI, pp 321–326. Rhode Island, USA, 1997.
29. Heule M, Van Maaren H. Aligning cnf-and equivalence-reasoning. In: International conference on theory and applications of satisfiability testing. Springer, 2004. p. 145–56.
30. Subbarayan S, Pradhan DK. Niver: Non-increasing variable elimination resolution for preprocessing sat instances. In: International conference on theory and applications of satisfiability testing. New York: Springer; 2004. p. 276–91.
31. Gent IP, Walsh T. Unsatisfied variables in local search. Hybrid problems, hybrid solutions. 1995. p. 73–85.
32. Michiels W, Aarts E, Korst J. Theoretical aspects of local search. New York: Springer; 2007.
33. Al-Sultan KS. A tabu search approach to the clustering problem. Pattern Recognit. 1995;28(9):1443–511.

34. Polash MM, Newton MA, Sattar A. Constraint-based local search for golomb rulers. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems. New York: Springer; 2015. p. 322–31.
35. Cai S, Kaile S, Sattar A. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. Artif Intell. 2011;175(9–10):1672–96.
36. Cai S, Kaile S. Local search for boolean satisfiability with configuration checking and subscore. Artif Intell. 2013;204:75–98.
37. Hossen MS, Polash MM. Implementing an efficient sat solver for structured instances. In: 2019 Joint 8th International Conference on Informatics, Electronics \& Vision (ICIEV) and 2019 3rd International Conference on Imaging, Vision \& Pattern Recognition (icIVPR). IEEE; 2019. p. 238–242.
38. Balint A, Frohlich A. Improving stochastic local search for SAT with a new probability distribution. In: International Conference on Theory and Applications of Satisfiability Testing. New York: Springer; 2010. p. 10–5.
39. Li CM, Li Y. Satisfying versus falsifying in local search for satisfiability. In: International Conference on Theory and Applications of Satisfiability Testing. Springer, 2012. p. 477–478.
40. Glover F, Marti R. Fundamentals of scatter search and path relinking 681 scheduling rules. In: Carnegie Mellon University; 1963.
41. Resende MGC, Ribeiro CC. Grasp with path-relinking: Recent advances and applications. In: Metaheuristics: progress as real problem solvers. New York: Springer; 2005. p. 29–63.
42. Festa P, Pardalos M, Pitsoulis S, Resende GC. Grasp with path relinking for the weighted maxsat problem. J Exp Algorithmics (JEA). 2007;11:2–4.
43. Festa P, Resende MGC. Hybridizations of grasp with path-relinking. In: Hybrid metaheuristics. New York: Springer; 2013. p. 135–55.
44. Zhenxing X, He K, Li C-M. An iterative path-breaking approach with mutation and restart strategies for the max-sat problem. Comput Operations Res. 2019;104:49–58.
45. Jarvis P, Arbelaez A. Cooperative parallel sat local search with path relinking. In: European Conference on Evolutionary Computation in Combinatorial Optimization (Part of EvoStar). New York: Springer; 2020. p. 83–98.
46. Van Laarhoven PJM, Aarts EHL. Simulated annealing. In: Simulated annealing: theory and applications. New York: Springer; 1987. p. 7–15.
47. Prestwich S. Random walk with continuously smoothed variable weights. In: International Conference on Theory and Applications of Satisfiability Testing. New York: Springer; 2005. p. 203–215.
48. Selman B, Kautz HA. An empirical study of greedy local search for satisfiability testing. In: AAAI, volume 93; 1993. p. 46–51.
49. Metropolis N, Ulam S. The monte carlo method. J Am Stat Association. 1949;44(247):335–41.