Chronological Backtracking

Alexander Nadel and Vadim Ryvchin

Intel Corporation, P.O. Box 1659, Haifa 31015, Israel
alexander.nadel, vadim.ryvchin@intel.com

Abstract. Non-Chronological Backtracking (NCB) has been implemented in every modern CDCL SAT solver since the original CDCL solver GRASP. NCB's importance has never been questioned. This paper argues that NCB is not always helpful. We show how one can implement the alternative to NCB–Chronological Backtracking (CB)–in a modern SAT solver. We demonstrate that CB improves the performance of the winner of the latest SAT Competition, `Maple_LCM_Dist`, and the winner of the latest MaxSAT Evaluation `Open-WBO`.

1 Introduction

Conflict-Driven Clause Learning (CDCL) SAT solving has been extremely useful ever since its the original implementation in the GRASP solver over 20 years ago [13], as it enabled solving real-world instances of intractable problems [2]. The algorithmic components of the original GRASP algorithms have been meticulously studied and modified over the years with the one notable exception of Non-Chronological Backtracking (NCB). NCB has always been perceived as an unquestionably beneficial technique whose impact is difficult to isolate, since it is entangled with other CDCL algorithms. NCB's contribution went unstudied even in [6]–a paper which aimed at isolating and studying the performance of fundamental CDCL algorithms. In this paper, we show how to implement the alternative to NCB–Chronological Backtracking (CB)–in a modern SAT solver.

Recall the CDCL algorithm. Whenever Boolean Constraint Propagation (BCP) discovers a falsified *conflicting clause* β, the solver learns a new *conflict clause* σ. Let the *conflict decision level* cl be the highest decision level in the conflicting clause β.[1] The new clause σ must contain one variable v assigned at cl (the 1UIP variable). Let the *second highest decision level* s be the highest decision level of σ's literals lower than cl ($s = 0$ for a unit clause). Let the *backtrack level* bl be the level the solver backtracks to just after recording σ and before flipping v.

Non-Chronological Backtracking (NCB) always backtracks to the second highest decision level (that is, in NCB, $bl = s$). The idea behind NCB is to improve the solver's locality by removing variables irrelevant for conflict analysis from the assignment trail. NCB's predecessor is conflict-directed backjumping, proposed in the context of the Constraint Satisfaction Problem (CSP) [11].

[1] In the standard algorithm, cl is always equal to the current decision level, but, as we shall see, that is not the case for CB.

Let *Chronological Backtracking (CB)* be a backtracking algorithm which always backtracks to the decision level immediately preceding the conflict decision level cl (that is, in CB, $bl = cl - 1$). In our proposed implementation, after CB is carried out, v is flipped and propagated (exactly as in the NCB case), and then the solver goes on to the next decision or continues the conflict analysis loop.

Implementing CB is a non-trivial task as it changes some of the indisputable invariants of modern SAT solving algorithms. In particular, the decision level of the variables in the assignment trail is no longer monotonously increasing. Moreover, the solver may learn a conflict clause whose highest decision level is higher than the current decision level. Yet, as we shall see, implementing CB requires only few short modifications to the solver.

To understand why CB can be useful consider the following example. Let $F = S \wedge T$ be a propositional formula in Conjunctive Normal Form (CNF), where S is a long satisfiable CNF formula (for example, assume that S has 10^7 variables), $T \equiv (c \vee \neg b) \wedge (c \vee b)$, and $V(S) \cap V(T) = \emptyset$, where $V(H)$ comprises the set of H's variables. Consider Minisat's [3] execution, given F. The solver is likely to start by assigning the variables in $V(S)$ (since S's variables are likely to have higher scores), satisfying S, and then getting to satisfying T. Assume that the solver has satisfied S and is about to take the next decision. Minisat will pick the literal $\neg c$ as the next decision, since the variable c has a higher index than b and 0 is always preferred as the first polarity. The solver will then learn a new *unit* conflict clause (c) and backtrack to decision level 0 as part of the NCB algorithm. After backtracking, the solver will satisfy S again from the very beginning and then discover that the formula is satisfied. Note that the solver is not expected to encounter any conflicts while satisfying S for the second time because of the phase saving heuristic [4, 10, 14] which re-assigns the same polarity to every assigned variable. Yet, it will have to re-assign all the 10^7 variables in $V(S)$ and propagate after each assignment. In contrast, a CB-based solver will satisfy F immediately after satisfying S without needing to backtrack and satisfy S once again.

Our example may look artificial, yet in real-word cases applying NCB might indeed result in useless backtracking (not necessarily to decision level 0) and reassignment of almost the same literals. In addition, NCB is too aggressive: it might remove good decisions from the trail only because they did not contribute to the *latest* conflict resolution. Guided by these two insights, our backtracking algorithm applies CB when the difference between the CB backtrack level and the NCB backtrack level is higher than a user-given threshold `T`, but only after a user-given number of conflicts `C` passed since the beginning of solving.

We have integrated CB into the SAT Competition 2017 [5] winner, `Maple_LCM_Dist` [7], and MaxSAT Evaluation 2017 [1] winner `Open-WBO` [9] (code available in [8]). As a result, `Maple_LCM_Dist` solves 3 more SAT Competition benchmarks; the improvement on unsatisfiable instances is consistent. `Open-WBO` solves 5 more MaxSAT Evaluation benchmarks and becomes much faster on 10 families.

In the text that follows, Sect. 2 provides CB's implementation details, Sect. 3 presents the experimental results, and Sect. 4 concludes our work.

2 Chronological Backtracking

We show how CB can be integrated into a modern CDCL solver [12] starting with an example. Consider the input formula, comprising 9 clauses $c_1 \dots c_9$, shown on the left-hand side in Fig. 1. We will walk through a potential execution of a CDCL solver using CB, while highlighting the differences between CB and NCB.

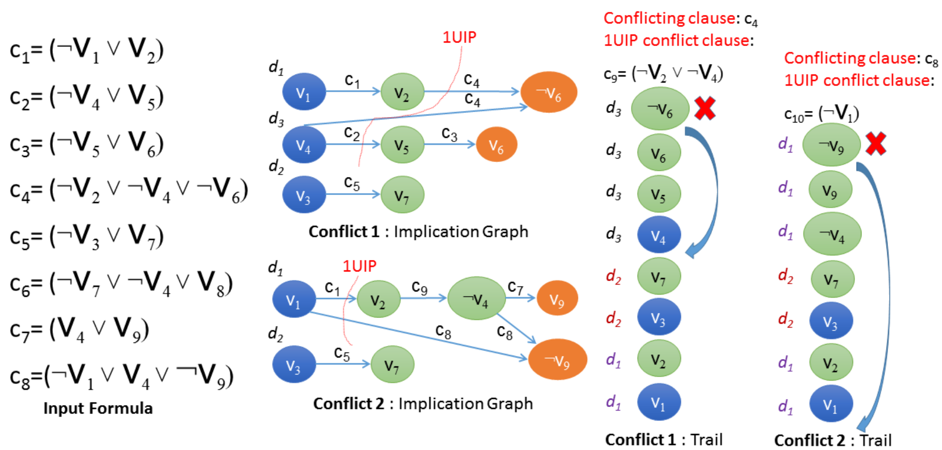

Fig. 1: CB Example

Assume the first decision at decision level d_1 is v_1, followed by the implication v_2 in clause c_1 (at the same level d_1). Then, a new decision v_3 implying v_7 in c_5 is carried out at decision level d_2. The next decision (at level d_3) is v_4. It implies v_5 in c_2 and v_6 in c_3, followed by a conflict, as all literals of c_4 are falsified under the current partial assignment. The implication graph and the trail at the time of conflict 1 are shown in Fig. 1. The conflict analysis will then learn a new 1UIP clause $c_9 = (\neg v_2 \vee \neg v_4)$ (resolution between clauses c_2, c_3, c_4).

At this point, a difference between NCB and CB is manifested. NCB would backtrack to the end of level d_1, skipping the irrelevant decision level d_2. We apply CB, which backtracks to the end of the previous decision level d_2. Backtracking to the end of d_2 undoes the assignments of v_6, v_5, v_4. Then, the algorithm asserts the unassigned 1UIP literal $\neg v_4$ and pushes it to the trail.

Our CB implementation marks $\neg v_4$'s decision level as d_1, since d_1 is the second highest level in the newly learned clause; however, $\neg v_4$ is placed into the trail after literals assigned at a higher decision level d_2. Hence, unlike in the NCB case, the decision levels of literals in the trail are not necessary monotonically increasing. It still holds, though, that each literal l implied at clause α is placed in the trail after all the other literals of α.

Let us proceed with our example. The assignment of $\neg v_4$ implies v_9 in c_7. Our algorithm marks the decision level of v_9 as d_1, since it is the highest level in the clause c_7 where v_9 is implied. Then, BCP finds a falsified clause c_8. Our algorithm identifies the decision level of the conflict as d_1, since all the literals in the conflicting clause c_8 were assigned at that level. At that point, CB will backtrack to the end of d_1 *before proceeding with conflict analysis*. Our backtrack algorithm will unassign the variables assigned at d_2, that is, v_3 and v_7, while keeping the variables assigned at d_1 (v_4 and v_9) in the same order. After the backtracking, conflict analysis is invoked. Conflict analysis will learn a new clause $c_{10} = (\neg v_1)$ (resolution between clauses c_1, c_9, c_7, c_8). The algorithm will then backtrack to the decision level $d_0 = d_1 - 1$ (to emphasize: in CB the backtrack level is the previous decision level, determined independently of the newly learned conflict clause).

2.1 Algorithm

Now we show the implementation of the high-level algorithms `CDCL` (Alg. 1), `BCP` (Alg. 2) and `Backtrack` (Alg. 3) with CB. In fact, we show both the NCB and the CB versions of each function. For `CDCL` and `BCP` most of the code is identical, except for the lines marked with either $_{ncb}$ or $_{cb}$.

Consider the high-level CDCL algorithm in Alg. 1. It operates in a loop that finishes after either all the variables are assigned (SAT) or when an empty clause is derived (UNSAT). Inside the loop, BCP is invoked. BCP returns a falsified conflicting clause if there is a conflict. If there is no conflict, a new decision is taken and pushed to the trail.

The first difference between CB and NCB shows up right after a conflict detection. The code between lines 4– 8 is applied only in the case of CB. If the conflicting clause contains one literal l from the maximal decision level, we let BCP propagating that literal at the second highest decision level in *conflicting_cls*. Otherwise, the solver backtracks to the maximal decision level in the conflicting clause before applying conflict analysis. This is because, as we saw in the example, the conflicting clause may be implied at a decision level earlier than the current level. The conflict analysis function returns the 1UIP variable to be assigned and the conflict clause σ. If σ is empty, the solver returns UNSAT. Assume σ is not empty. The backtrack level bl is calculated differently for NCB and CB. As one might expect, bl comprises the second highest decision level in σ in the case of NCB case and the previous decision level in the case of CB (note that for CB the solver has already backtracked to the maximal decision level in the conflicting clause). Subsequently, the solver backtracks to bl and pushes the 1UIP variable to the trail before continuing to the next iteration of the loop.

Consider now the implementation of `BCP` in Alg. 2. `BCP` operates in a loop as long as there exists at least one unvisited literal in the trail ν. For the first unvisited literal l, `BCP` goes over all the clauses watched by l. Assume a clause β is visited. If β is a unit clause, that is, all β's literals are falsified except for one unassigned literal k, `BCP` pushes k to the trail. After storing k's implication reason in *reason*(k), `BCP` calculates and stores k's implication level *level*(k). The

Algorithm 1 `CDCL`

```
ν: the trail, stack of decisions and implications
_ncb: marks the NCB code
_cb: marks the CB code
Input: CNF formula
Output: SAT or UNSAT
 1: while not all variables assigned do
 2:     conflicting_cls := BCP();
 3:     if conflicting_cls ≠ null then
 4:         if conflicting_cls contains one literal from the maximal level then
 5:             _cb Backtrack(second highest decision level in conflicting_cls)
 6:             _cb continue
 7:         else
 8:             _cb Backtrack(maximal level in conflicting_cls)
 9:         (1uip, σ) := ConflictAnalysis(conflicting_cls)
10:         if σ is empty then
11:             return UNSAT
12:         _ncb bl := second highest decision level in σ (0 for a unit clause)
13:         _cb bl := current decision level - 1
14:         Backtrack(bl)
15:         Push 1uip to ν
16:     else
17:         Decide and push the decision to ν
18: return SAT
```

implication level calculation comprises the only difference between CB and NCB versions of BCP. The current decision level always serves as the implication level for NCB, while the maximal level in β is the implication level for CB. Note that in CB a literal may be implied *not* at the current decision level. As usual, BCP returns the falsified conflicting clause, if such is discovered.

Finally, consider the implementation of `Backtrack` in Alg. 3. For the NCB case, given the target decision level bl, `Backtrack` simply unassigns and pops all the literals from the trail ν, whose decision level is greater than bl. The CB case is different, since literals assigned at different decision levels are interleaved on the trail. When backtracking to decision level bl, `Backtrack` removes all the literals assigned after bl, but it puts aside all the literals assigned before bl in a queue μ *maintaining their relative order*. Afterwards, μ's literals are returned to the trail in the same order.

2.2 Combining CB and NCB

Our algorithm can easily be modified to heuristically choose whether to use CB or NCB for any given conflict. The decision can be made, for each conflict, in the main function in Alg. 1 by setting the backtrack level to either the second highest decision level in σ for NCB (line 12) or the previous decision level for CB (line 13).

Algorithm 2 `BCP`

dl: current decision level
ν: the trail, stack of decisions and implications
$_{ncb}$: marks the NCB code
$_{cb}$: marks the CB code

```
BCP()
while ν contains at least one unvisited literal do
    l := first literal in ν, unvisited by BCP
    wcls := clauses watched by l
    for β ∈ wcls do
        if β is unit then
            k := the unassigned literal of β
            Push k to the end of ν
            reason(k) := β
            _ncb level(k) := dl
            _cb level(k) := max level in β
        else
            if β is falsified then
                return β
return null
```

Algorithm 3 `Backtrack`

dl: current decision level
ν: the trail, stack of decisions and implications
level_index$(bl + 1)$: the index in ν of $bl + 1$'s decision literal

`Backtrack(`*bl*`)` **: NCB version**
Assume: $bl < dl$

```
while ν.size() ≥ level_index(bl + 1) do
    Unassign ν.back()
    Pop from ν
```

`Backtrack(`*bl*`)` **: CB Version**
Assume: $bl < dl$

```
Create an empty queue μ
while ν.size() ≥ level_index(bl + 1) do
    if level(ν.back()) ≤ bl then
        Enqueue ν.back() to μ
    else
        Unassign ν.back()
    Pop from ν
while μ is not empty do
    Push μ.first() to the end of ν
    Dequeue from μ
```

In our implementation, NCB is always applied before `C` conflicts are recorded since the beginning of the solving process, where `C` is a user-given threshold. After `C` conflicts, we apply CB whenever the difference between the CB backtrack level (that is, the previous decision level) and the NCB backtrack level (that is, the second highest decision level in σ) is higher than a user-given threshold `T`.

We introduced the option of delaying CB for `C` first conflicts, since backtracking chronologically makes sense only after the solver had some time to aggregate variable scores, which are quite random in the beginning. When the scores are random or close to random, the solver is less likely to proceed with the same decisions after NCB.

3 Experimental Results

We have implemented CB in `Maple_LCM_Dist` [7], which won the main track of the SAT Competition 2017 [5], and in `Open-WBO`, which won the complete unweighted track of the MaxSAT Evaluation 2017 [1]. The updated code of both solvers is available in [8]. We study the impact of CB with different values of the two parameters, `T` and `C`, in `Maple_LCM_Dist` and `Open-WBO` on SAT Competition 2017 and MaxSAT Evaluation 2017 instances, respectively. For all the tests we used machines with 32Gb of memory running Intel® Xeon® processors with 3Ghz CPU frequency. The time-out was set to 1800 seconds. All the results refer only to benchmarks solved by at least one of the participating solvers.

3.1 SAT Competition

In preliminary experiments, we found that $\{\mathtt{T} = 100, \mathtt{C} = 4000\}$ is the best configuration for `Maple_LCM_Dist`. Table 1 shows the summary of run time and unsolved instances of the default `Maple_LCM_Dist` vs. the best configuration in CB mode, $\{\mathtt{T} = 100, \mathtt{C} = 4000\}$, as well as "neighbor" configurations $\{\mathtt{T} = 100, \mathtt{C} = 3000\}$, $\{\mathtt{T} = 100, \mathtt{C} = 5000\}$, $\{\mathtt{T} = 90, \mathtt{C} = 4000\}$ and $\{\mathtt{T} = 110, \mathtt{C} = 4000\}$. Fig. 2 and Fig. 3 compare the default `Maple_LCM_Dist` vs. the overall winner $\{\mathtt{T} = 100, \mathtt{C} = 4000\}$ on satisfiable and unsatisfiable instances respectively. Several observations are in place.

First, Table 1 shows that $\{\mathtt{T} = 100, \mathtt{C} = 4000\}$ outperforms the default `Maple_LCM_Dist` in terms of for both the number of solved instances and the run-time. It solves 3 more benchmarks and is faster by 4536 seconds.

Second, CB is consistently more effective on unsatisfiable instances. Table 1 demonstrates that the best configuration for unsatisfiable instances $\{\mathtt{T} = 100, \mathtt{C} = 5000\}$ solves 4 more instances than the default configuration and is faster by 5783 seconds. The overall winner $\{\mathtt{T} = 100, \mathtt{C} = 4000\}$ solves 3 more unsatisfiable benchmarks than the default and is faster by 5113 seconds. Fig. 3 shows that CB is beneficial on the vast majority of unsatisfiable instances. Interestingly, we found that there is one family on which CB consistently yields significantly better results: the 27 instances of the g2-T family. On that family, the run-time in CB mode is never worse than that in NCB mode. In addition, CB helps to solve 4 more benchmarks than the default version and causes the solver to be faster by 1.5 times on average.

Finally, although the overall winner is slightly outperformed by the default configuration on satisfiable instances, CB can be tuned for satisfiable instances too. $\{\texttt{T} = 100, \texttt{C} = 3000\}$ solves 2 additional satisfiable instances, while $\{\texttt{T} = 110, \texttt{C} = 4000\}$ solves 1 additional instance faster than the default. We could not pinpoint a family, where CB shows a significant advantage on satisfiable instances.

		Base	T = 100			C = 4000	
			C = 3000	C = 4000	C = 5000	T = 90	T = 110
SAT	Unsolved	13	11	13	16	20	12
	Time	50003	53362	50580	59167	59482	47748
UNSAT	Unsolved	6	5	3	2	4	6
	Time	58414	54034	53301	52631	52481	53991
ALL	Unsolved	19	16	16	18	24	18
	Time	108417	107396	103881	111798	111963	101739

Table 1: Results of `Maple_LCM_Dist` on SAT Competition 2017 Instances

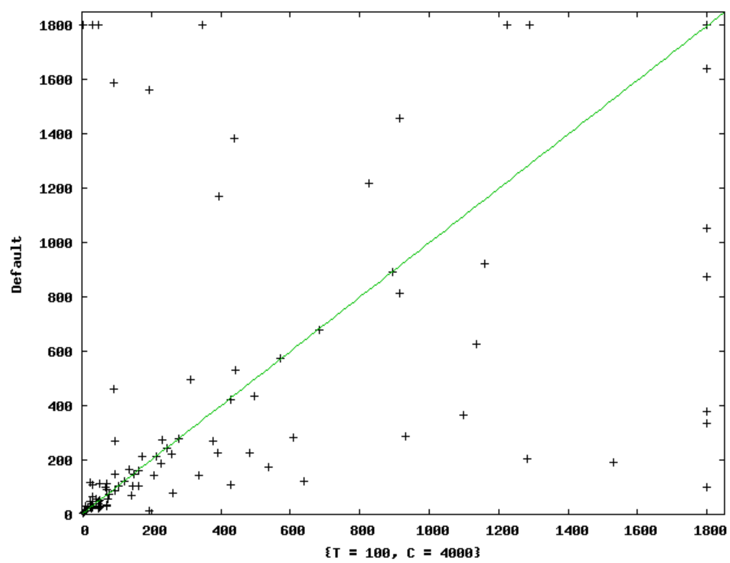

Fig. 2: `Maple_LCM_Dist` on SAT

3.2 MaxSAT Evaluation

In preliminary experiments, we found that $\{\texttt{T} = 75, \texttt{C} = 250\}$ is the best configuration for `Open-WBO` with CB. Consider the five left-most columns of Table 2. They present the number of solved instances and the run-time of the default `Open-WBO` vs. $\{\texttt{T} = 75, \texttt{C} = 0\}$ (abbreviated to $\{75, 250\}$) over the MaxSAT Evaluation families (complete unweighted track). The second row shows the overall results. CB helps `Open-WBO` to solve 5 more instances in less time. The subsequent rows of Table 2 show the results for families, where either `Open-WBO` or

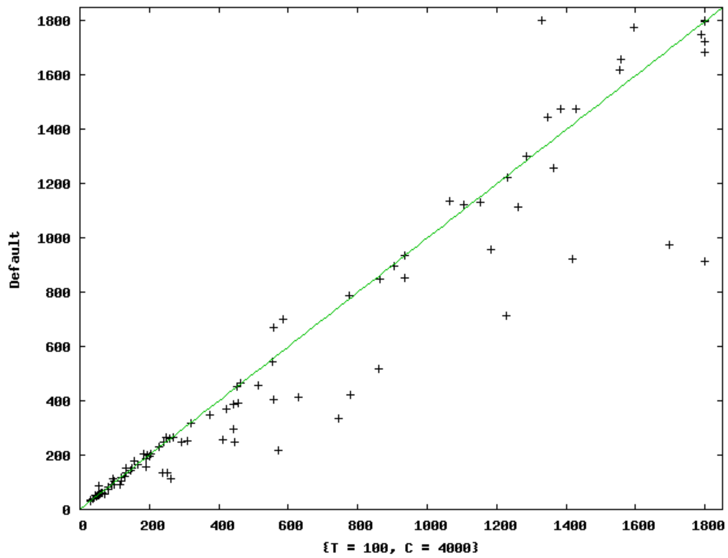

Fig. 3: `Maple_LCM_Dist` on UNSAT

$\{\mathtt{T} = 75, \mathtt{C} = 250\}$ was significantly faster than the other solver, that is, it either solved more instances or was at least two times as fast. One can see that CB significantly improved the performance of `Open-WBO` on 10 families, while the performance was significantly deteriorated on 3 families only. The other columns of Table 2 present the results of 4 configurations neighbor to $\{\mathtt{T} = 75, \mathtt{C} = 250\}$ for reference.

Family	Default		{75, 250}		{75, 0}		{75, 500}		{50, 250}		{100, 250}	
	#S	Time	#S	Time	#S	Time	#S	Time	#S	Time	#S	Time
Grand Total	639	53048	644	50704	642	51370	640	52406	640	53582	643	51022
kbtree	0	3600	2	2756	1	3332	2	2921	2	2771	2	2733
atcoss-sugar	11	2179	12	1812	12	1328	11	2013	11	2004	12	1889
close-solutions	32	2692	33	4235	32	2711	32	2597	33	2589	32	4382
extension-enforcement	7	1963	8	828	7	1975	7	1942	8	1093	8	1306
gen-hyper-tw	5	4348	6	3871	6	3219	5	4057	5	3901	7	3383
treewidth-computation	24	3407	25	2306	24	3661	25	2169	23	4527	24	3778
atcoss-mesat	11	1660	11	605	11	703	11	610	11	674	11	534
min-fill	4	1105	4	413	4	384	4	910	4	244	4	349
packup	35	697	35	253	35	172	35	460	35	252	35	253
scheduling	1	206	1	92	1	153	1	164	1	141	1	130
bcp-syn	21	2535	20	2643	21	2247	21	2642	20	3145	20	2733
mbd	35	1327	34	1982	34	1972	34	2006	35	1275	35	1222
hs-timetabling	1	48	1	317	1	276	1	968	1	396	1	453

Table 2: Results of `Open-WBO` on MaxSAT Evaluation 2017 Instances

4 Conclusion

We have shown how to implement Chronological Backtracking (CB) in a modern SAT solver as an alternative to Non-Chronological Backtracking (NCB), which has been commonly used for over two decades. We have integrated CB into the winner of the SAT Competition 2017, `Maple_LCM_Dist`, and the winner of MaxSAT Evaluation 2017 `Open-WBO`. CB improves the overall performance of both solvers. In addition, `Maple_LCM_Dist` becomes consistently faster on unsatisfiable instances, while `Open-WBO` solves 10 families significantly faster.

References

1. Carlos Ansotegui, Fahiem Bacchus, Matti Järvisalo, and Ruben Martins, editors. *MaxSAT Evaluation 2017: Solver and Benchmark Descriptions*, volume B-2017-2 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2017.
2. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
3. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
4. Daniel Frost and Rina Dechter. In search of the best constraint satisfaction search. In *AAAI*, pages 301–306, 1994.
5. Marijn Heule, Matti Järvisalo, and Tomas Balyo. Sat competition 2017. `https://baldur.iti.kit.edu/sat-competition-2017/`.
6. Hadi Katebi, Karem A. Sakallah, and João P. Marques Silva. Empirical study of the anatomy of modern sat solvers. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 343–356. Springer, 2011.
7. Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 703–711. ijcai.org, 2017.
8. Alexander Nadel and Vadim Ryvchin. Chronological backtracking: Solvers. `goo.gl/ssukuu`.
9. Miguel Neves, Ruben Martins, Mikolás Janota, Inês Lynce, and Vasco M. Manquinho. Exploiting resolution-based representations for maxsat solving. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2015.
10. Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, pages 294–299, 2007.
11. Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299, 1993.
12. João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Biere et al. [2], pages 131–153.
13. João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.
14. Ofer Strichman. Tuning SAT checkers for bounded model checking. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2000.