Maximilian Eibl, Martin Gaedke. (Hrsg.): INFORMATIK 2017,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 2549

A Monte Carlo Tree Search Based Conflict-Driven Clause Learning SAT Solver

Jens Schloeter[1]

Abstract: Most modern state-of-the-art Boolean Satisfiability (SAT) solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and exploit techniques like unit propagation and Conflict-Driven Clause Learning. Even though this approach proved to be successful in practice and most recent publications focus on improving it, the success of the Monte Carlo Tree Search (MCTS) algorithm in other domains led to research in using it to solve SAT problems. While a MCTS-based algorithm was successfully used to solve SAT problems, a number of established SAT solving techniques like clause learning and parallelization were not included in the algorithm. Therefore this paper presents ways to combine the MCTS-based SAT solving approach with established SAT solving techniques like Conflict-Driven Clause Learning and shows that the addition of those techniques improves the performance of a plain MCTS-based SAT solving algorithm.

Keywords: Boolean Satisfiability, SAT Solving, Monte Carlo Tree Search, Conflict-Driven Clause Learning

1 Introduction and Related Work

The Boolean Satisfiability (SAT) problem is an NP-complete decision problem, which has applications in a number of topics like automatic test case generation [SBSV88], formal verification [GGW06] and many more. One main reason for the usage of SAT in those fields is the existence of efficiently performing solvers.

As Marques-Silva et. al. [MSLM09] point out, most SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and use backtracking to determine a solution. While those solvers proved to be successful in practice, the success of the Monte Carlo Tree Search (MCTS) algorithm in other domains such as General Game Playing and different combinatorial problems [Br12] led to recent work towards using it to solve SAT, MaxSAT [GR16] and other related problems [Lo13]. For example Previti et. al. [Pr11] presented a solver that uses the MCTS algorithm in combination with classical SAT solving techniques like unit propagation [DP60]. Their experiments showed that the MCTS-based approach performed well if the SAT instance has an underlying structure.

[1] Universität Bremen, Fachbereich 3, Bibliothekstr. 5 (MZH), 28359 Bremen, Deutschland, jschloet@uni-bremen.de

doi:10.18420/in2017_257

So while they had some success to combine the MCTS-based approach with SAT solving techniques like unit propagation, they did not include other key features of modern state-of-the-art SAT solvers. One contribution of this paper is to present a method to use Conflict-Driven Clause Learning (CDCL) [Zh01] in a MCTS-based algorithm and show that the use of that technique improves the performance of the algorithm.

Even with the usage of additional features like CDCL it is uncertain if the MCTS-based approach can compete with modern state-of-the-art SAT solvers. But as Martins et. al. explained [MML12], one of the main research directions of parallel SAT solving – the portfolio approach – relies on diversity in the search strategies of the used SAT solvers and as the MCTS-based CDCL solver greatly differs from a DPLL based algorithm, it might prove useful in such scenarios. Because of that, the second contribution of this paper focuses on exploiting advantages of the MCTS algorithm when solving SAT in parallel and presents a way to use it in a portfolio approach.

2 Preliminaries

This section presents the fundamentals of the MCTS-based CDCL solver. Therefore it describes the input problem of the solver and the SAT solving technique unit propagation. Additionally a MCTS-based SAT solving algorithm similar to UCTSAT by Previti et. al. [Pr11] is introduced, that is later used as the basis of the MCTS-based CDCL solver.

2.1 Problem Description

The MCTS-based CDCL solver will accept boolean formulas in *Conjunctive Normal Form (CNF)*. A boolean formula in CNF is a conjunction of *clauses*, where a clause is a disjunction of literals and a *literal* is either a *variable* x_i or its negation $\neg x_i$.

Each variable x_i can be assigned to either value 0, value 1 or be unassigned. A literal is called satisfied by an assignment if it has the truth value 1, i.e. is a variable x_i that is assigned to 1 or a negated variable $\neg x_i$ where x_i is assigned to 0. A Clause is called satisfied by an assignment if at least one of its literals is satisfied. A CNF formula is called satisfied by an assignment if all of its clauses are satisfied. The goal of SAT is to find an assignment that satisfies a given formula or to determine that no such assignment exists.

In the context of SAT solving a *conflict* is the occurrence of an unsatisfied clause for an assignment.

2.2 Unit Propagation

One key operation when solving SAT is the *unit propagation* [DP60]. A clause C is called *unit* if exactly one of its literals is unassigned and the others are unsatisfied. The unassigned

literal is then called *unit literal*. The merit of unit clauses is that they enforce the assignment of their unit literal as it must be assigned to truth value 1 to satisfy the clause. SAT solvers exploit this fact and check after each assignment if clauses became unit and, if that is the case, execute the assignment of their unit literals. As this can lead to other clauses becoming unit and thus enforcing further assignments, this process is called *unit propagation*.

2.3 Monte Carlo Tree Search based SAT Solving

This section introduces the MCTS algorithm for SAT solving based on [Pr11]. Like most implementations of Monte Carlo Tree Search it is based on the $UCB1$ algorithm for multi-armed bandit problems and uses the *Upper Confidence Bounds for Trees (UCT)* formula to decide in which direction to expand the search tree [Mu14].

The main goal of using the UCT formula is to build an asymmetric search tree that is expanded at the most promising nodes, i.e. at the nodes with the highest estimated values. As the estimated value can be biased, the use of the UCT formula also encourages exploring new paths of the search tree.

In our case, every node of the search tree represents a variable assignment and is marked with a variable that is unassigned in the node but becomes assigned in its children. The root node represents the empty assignment, i.e. an assignment where every variable is unassigned. For that purpose, each node n has a reference to its parent node $n.parent$ and a property $n.assignment$ that stores the value the parent's variable is assigned to. To be able to use the UCT formula each node additionally has an estimated value as well as a counter for the number of times the algorithm has traversed through it. Both of these properties are used to calculate the UCT value of a node as in equation (1), where the first summand benefits nodes with high estimated values and the second one those that are not well explored. The factor C in equation (1) is the so-called exploration constant that can be used to weight the summands.

$$UCTValue(node) = \frac{node.estimatedValue}{node.counter} + C \cdot \sqrt{\frac{2 \ln node.parent.counter}{node.counter}} \quad (1)$$

Given a CNF formula, the UCT based solver executes iterations until a satisfying assignment is found or the formula is proved to be unsatisfiable. In each iteration one node is added to the search tree, its value estimated and the values of the tree's nodes are refreshed. During the iteration an instance of the given formula is kept up to date by executing encountered assignments and – after each assignment – unit propagation on this formula. Algorithm 1 shows the pseudo code of the complete algorithm that iterates through the four phases of the algorithm – *selection*, *expansion*, *simulation* and *backpropagation* – until a satisfying

Algorithm 1: main(formula f)

```
formula ← f;
rootNode ← newNode();
rootNode.variable ← variableOrder.next();
begin
    while true do
        if rootNode.conflict then
            return UNSAT;
        formula ← f.copy();
        selectedNode ← select(rootNode, formula);
        newNode ← expand(selectedNode, formula);
        simulationResult ← simulate(formula);
        if simulationResult.SAT then
            return SAT;
        else
            backpropagate(newNode, simulationResult);
```

assignment is found or the root node is marked as conflicting and thus the formula is unsatisfiable.

The selection phase of the algorithm – see $select(rootNode, formula)$ in Algorithm 1 – is used to decide at which point to expand the search tree. To do so, the algorithm starts from the root node and traverses trough the tree until one of its leaves is reached. During this process the next node to be traversed through is always the child of the current node with the highest UCT value. A node that is encountered during this phase can have a conflicting assignment if all of its children proved to result in conflicts. If such a node occurs, it is marked as conflicting and removed from the search tree.

When the expansion point of the tree is determined, the expansion phase of the algorithm – see $expand(selectedNode, formula)$ in Algorithm 1 – is executed. In this phase a new child of the selected node is added to the search tree which assigns the variable of the selected node to a value that has not been tried yet. If that assignment proves to result in a conflict, the new node is marked as conflicting. The variable of the new node is determined using the *variable order heuristic*, which is a common part of SAT solvers and determines the next variable to be assigned. In our implementation we used a heuristic similar to the *Variable State Independent Decaying Sum (VSIDS)* order heuristic that basically assigns the variables at first that occur more often in the formula [Li15].

To estimate the value of the new node, the simulation phase is executed, which – starting with the current formula and the assignment of the new node – selects variables according to the variable order heuristic and assigns them randomly until a conflict or a satisfying assignment is reached. Again, unit propagation is executed after each assignment. The

number of satisfied clauses in the resulting formula is then used as an initial estimated value for the new node. During the selection phase a normalized form of this value is used in the UCT formula. Note that, if the simulation finds a satisfying assignment, the problem is solved and the algorithm can stop. This phase is part of Algorithm 1 as $simulate(formula)$.

After a simulation is executed, its result is back-propagated through the search tree and the estimated value as well as the counter of every node on the path from the root to the new node is refreshed, i.e. their counters are increased by one and their estimated values are increased by the estimated value of the new node. This process is indicated by $backpropagate(newNode, simulationResult)$ in Algorithm 1.

3 Conflict-Driven Clause Learning

As already mentioned, the learning and usage of new clauses is a key aspect of modern SAT solvers. This section introduces the concept of *Conflict-Driven Clause Learning* and shows how it can be used in the MCTS-based algorithm.

The clause learning usually takes place whenever a conflict occurs and determines a new clause that formulates the reason for the conflict. The new clause is learned using the unit clauses whose enforced assignments led to the conflict and added to the formula. For a more detailed explanation on clause learning see [MSLM09].

Our approach for a MCTS-based CDCL solver is to learn a new clause whenever a simulation leads to a conflict and to add it to the formula when the algorithm starts a new iteration. As no backtracking is executed, our algorithm does not exploit the clause learning to determine the backtracking level.

One aspect to consider when using this technique is that at each node that is created during the expansion phase of the algorithm, a number of variables are already assigned to a value. While no node is created that has a variable assignment that directly causes a conflict, the learning of additional clauses can lead to the existence of nodes that have assignments which lead to a conflict in one of the learned clauses. Those nodes should not be included in the selection phase of the algorithm and do not have to be further stored. As it would be too complex to check if each node still has a valid variable assignment whenever a new clause is learned, we modify our algorithm to check the nodes that are traversed through during the selection phase. Figure 1 shows such a situation, where the algorithm is determining the successor of the green colored node during the selection phase and notices that assigning the current variable x_k to 0 would lead to a conflict such that the red colored node is conflicting.

The figure shows that the algorithm basically only has the choice to select the assignment $x_k = 1$ because the alternative would lead into a conflict. At this point, there are two possible reasons for the appearance of this conflict. First, the variable assignment could directly lead to the conflict and second, the unit propagation after the assignment could lead to the conflict.

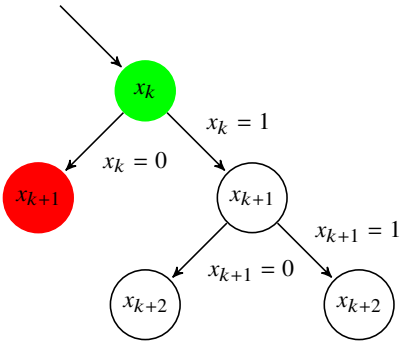

Fig. 1: Encountering a conflicting node during the selection phase

Note that if the variable assignment directly leads into a conflict, it means that a clause C must exist which is unit and has the unit literal x_k. This is an important aspect as we defined the algorithm to execute unit propagation after each variable assignment, so when assigning the precursor variable of x_k, the unit propagation must have led to the assignment $x_k = 1$. Because of that, not only the red colored node becomes unnecessary but also the green colored one as its variable must already have been assigned, so the algorithm can prune both nodes[2] and continue in the state shown in Figure 2. To detect this case, the algorithm only has to check if the variable of the current node has already been assigned due to unit propagation. If on the other hand the unit propagation after the assignment of the red node leads to the conflict, the green node does not become redundant as its assignment was not already executed due to unit propagation in the previous nodes. In this case, the red node can still be pruned, but the green one needs to be retained. This case can be detected by checking whether a variable assignment led to a conflict during the selection phase.

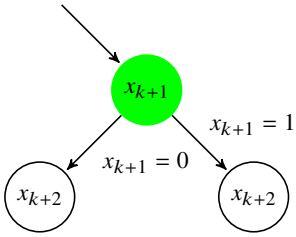

Fig. 2: The search tree of Figure 1 after pruning nodes

The learning and addition of new clauses yields two major benefits for the MCTS-based CDCL solver, where the first benefit is the pruning of the created search tree and the second

[2] Note that the conflicting node can have child nodes which will be pruned as well

one is additional unit propagation caused by the learned clauses during the simulation phase. While the first benefit should lead to less memory usage and a shorter selection phase, the second benefit should lead to an improved simulation phase and thus fewer simulations and less time should be needed to find a solution. Our experiments in Section 6 show, that the additional usage of clause learning greatly improves the performance of the MCTS-based algorithm.

4 Parallelization

In the previous sections we showed that the performance of the MCTS-based algorithm can be improved by using SAT solving techniques such as clause learning and unit propagation. However, the current implementation of the algorithm cannot compete with established solvers. But, as Previti et. al. [Pr11] already pointed out, there may be value in using it in parallel environments. In this section we will focus on how to integrate the MCTS-based approach into a parallel SAT solver.

Most parallel SAT solvers follow one of two approaches namely search splitting and portfolio searches [MML12]. In the first one, the search space is divided into multiple sub spaces that are searched by different solvers. In the second approach multiple solvers that differ in their search strategies are used to solve the complete problem. In both approaches the different solver instances may communicate through the sharing of learned clauses.

When using backtracking based algorithms, the import of a learned clause into another solver causes some overhead, as it may be conflicting for the current variable assignment or a unit clause may exist whose propagation would lead to a conflict. When importing a clause, one has to make sure that the solver stays in a non-conflicting state [MML12]. Especially when multiple clauses are imported that process may cause computational overhead.

While this overhead exists for backtracking based solvers, MCTS-based algorithms start their search from the root node after each iteration. At that point of the algorithm, an arbitrary number of clauses can be added without creating computational overhead. Our approach to use the MCTS-based algorithm in one of the described parallel SAT solving scenarios is to pass the learned clause to the other solvers after the simulation phase and import the new clauses at the beginning of each iteration.

When following this approach the import of new clauses may not cause overhead, but as for example Goldberg et. al. [GN02] point out, the storage of all learned clauses can require too much memory usage. While this is already a problem for single threaded SAT solvers, the usage of multiple solvers in a parallel solver causes a higher number of learned clauses and thus even more memory usage. Additionally, the usage of more clauses in the MCTS-based CDCL solver leads to additional effort to keep the clauses up to date when assigning variables. To reduce this effect we use a similar approach to Goldberg et. al. [GN02] and delete the learned clauses with the lowest value after a fixed number of MCTS

iterations, where the value of each clause is calculated using its length, i.e. the number of variables in the clause, its global activity, i.e. the number of times the clause was responsible for a conflict in one of the used solvers, and its age.

5 Probability Heuristics

The previous sections defined the simulations of the MCTS-based CDCL solver to assign the variables of the formula uniformly at random. While this is an approach that can be used to solve SAT and is quite standard when using MCTS, it is also common to use domain specific knowledge – if it is available – to weight the probabilities in the simulation phase [Br12]. This section focuses on using knowledge that can be extracted from the given SAT formula as well as from the learned clauses to weight the probabilities.

In backtracking based SAT solvers the decision to which value to assign the variables is made using the *Phase Selection Heuristic*, where one approach is to assign a variable x_i to 1 if the literal x_i occurs more often in the formula than $\neg x_i$ and vice versa. We adapted this approach and used probabilities according to equation (2) during the simulation phase, where $num(x_i)$ is the number of occurrences of literal x_i in the formula and $P(x_i = 1)$ is the probability of assigning variable x_i to 1.

$$P(x_i = 1) = \frac{num(x_i)}{num(x_i) + num(\neg x_i)} \tag{2}$$

Note that the number of occurrences of a literal in the formula changes during the course of the algorithm because of the newly added clauses. We experimented with different probabilities when solving benchmark problems, but were not able to reach clear results, meaning that for some problems the probabilities according to equation (2) performed better and for others the uniform random assignment outperformed the competitors.

6 Experiments

To verify the existence of the mentioned benefits and the general improvement of the MCTS-based solver by using CDCL, we implemented a MCTS-based SAT solver as specified in Section 2.3 as well as all extensions of the algorithm that were introduced in this paper. We used them to solve a number of SAT encoded graph coloring and planing problems[3] and compared the results of the different variants. To reach results that are more robust against other activities on the benchmark system, we ran every variant a hundred times on every problem and averaged the results.

[3] All SAT encodings were taken from `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

Figure 3 shows the average running time for solving graph coloring problems when using clause learning and when not. The data shows that the algorithm only needed less than half of the running time when using clause learning. While the pure MCTS solver performed similar or even slightly better on easy problems, i.e. problems that could generally be solved in little time, the MCTS-based CDCL solver performed significantly better on harder problems. One reason for this might be, that the clause learning produces computational overhead that is not necessary for easy problems, while the benefits of the clause learning have a bigger impact on harder problems. For the planning problems, which are of bigger size than the graph coloring problems, the algorithm without clause learning mostly did not terminate in reasonable time and thus only the results of the MCTS-based CDCL solver are mentioned below. This phenomenon may indicate, that the clause learning will have an even bigger impact on harder problems.

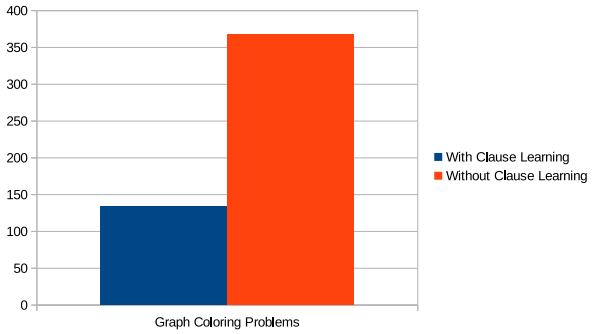

Fig. 3: Average running time in milliseconds for solving graph coloring problems

Table 1 shows the average number of created and pruned nodes for the graph coloring and planing problems. It shows that for the first problem class, nine percent of the created nodes could be pruned due to information gained by the learned clauses and that for the second class, the pruning percentage is even higher at nearly 25 percent.

	Created Nodes	Pruned Nodes	Ratio
Graph Coloring Problems	26.26	2.61	0.09
Planning Problems	193.79	48.26	0.25

Tab. 1: Average number of created and pruned nodes for graph coloring and planing problems

These results show that the algorithm performs significantly better when clause learning is used and that the percentage of pruned nodes even increases for the planning problems, which consist of more than twice as many clauses than the graph coloring problems.

Additionally we tested how the performance of the algorithm changes when using the probability heuristic as introduced in section 5. Figure 4 shows that the MCTS-based CDCL solver that uses the probability heuristic performs slightly worse than the uniform random variant, but still better than the algorithm that does not use clause learning. However, the

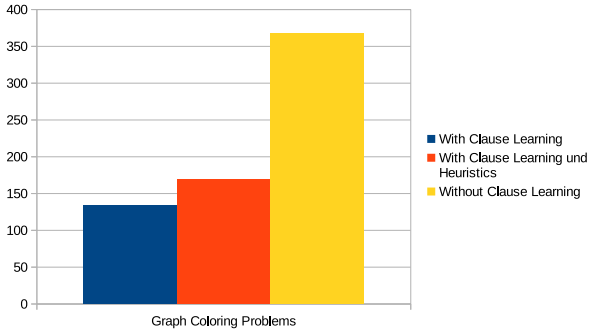

Fig. 4: Average running time in milliseconds for solving graph coloring problems when using the probability heuristic as introduced in section 5 in comparison to the variants that use uniform random simulations.

usage of the probability heuristic did improve the performance of the MCTS-based CDCL solver on single problems and thus may be useful when used as part of a solver portfolio.

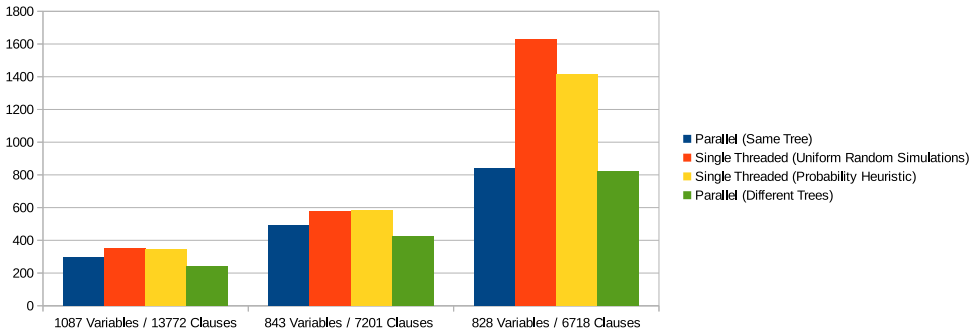

Fig. 5: Average running time for solving different planning problems in parallel and single threaded.

To test this hypothesis and, whether the performance of the algorithm can be improved when running four different solver instances in a parallel approach as described in section 4, we used this approach to solve different SAT instances and compared the results to those of the single threaded variants. Additionally we used two different variants to parallelize our solver. In the first variant the instances of the solver work on the same search tree, while in the second variant each instance uses its own search tree. In both variants we used different values for the exploration constant of the UCT formula as well as different simulation probabilities for the different solver instances. Figure 5 shows the average running time of the parallel and the single threaded variants when solving planning problems of different sizes.

Our experiments showed that the usage of multiple MCTS-based CDCL solvers improves

the performance of the algorithm when using the same search tree as well as when using different search trees, while the latter variant performed slightly better. Only on some smaller graph coloring problems the single threaded variant performed similar or even better. Again, this phenomenon can be explained by the overhead of creating threads and synchronizing the clause sharing, which has a bigger impact on smaller problems.

7 Conclusion

This paper introduced a Monte Carlo Tree Search based Conflict-Driven Clause Learning SAT solver and showed that the usage of CDCL significantly improves the performance of the solver in both, memory usage and computing time, by pruning the search tree and directing the simulations in better directions via unit propagation on the learned clauses.

But even with this performance improvement our current implementation of the MCTS-based CDCL solver could not compete with established SAT solvers. Nevertheless the performance improvement that was reached by the addition of CDCL showed that the algorithm benefits from the integration of known SAT solving techniques. In order to determine whether our approach can be further improved and be the basis of a competitive SAT solver, one additional research topic would be to investigate the usage of preprocessing techniques, restarts and other features of state-of-the-art SAT solvers in the MCTS-based algorithm.

Additionally we introduced a way to use multiple solver instances in parallel and showed that our approach can easily be integrated in a multi threading environment. While our experiments showed, that the single threaded variant yielded similar or slightly better results on smaller problems, the parallel variant performed superior on bigger problems. However, we used very similar configured solver instances in the parallel variant, that only differed in their values for the exploration constant in the UCT formula and their simulation probabilities, and thus were not able to fully benefit from the usage of different search strategies. A next research step would be to integrate the MCTS-based CDCL solver in a portfolio solver that uses a range of different search strategies and algorithms and investigate its performance.

References

[Br12] Browne, Cameron; Powley, Edward; Whitehouse, Daniel; Lucas, Simon M.; Cowling, Peter I.; Rohlfshagen, Philipp; Tavener, Stephen; Perez, Diego; Samothrakis, Spyridon; Colton, Simon: A Survey of Monte Carlo Tree Search Methods. 4(1):1–43, 2012.

[DP60] Davis, Martin; Putnam, Hilary: A computing procedure for quantification theory. Journal of the ACM (JACM), 7(3):201–215, 1960.

[GGW06] Gupta, Aarti; Ganai, Malay K.; Wang, Chao: SAT-Based Verification Methods and Applications in Hardware Verification. In (Bernardo, Marco; Cimatti, Alessandro,

eds): Formal Methods for Hardware Verification: 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006, Bertinoro, Italy, May 22-27, 2006, Advanced Lectures. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 108–143, 2006.

[GN02] Goldberg, E.; Novikov, Y.: BerkMin: A fast and robust SAT-solver. In: Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition. pp. 142–149, 2002.

[GR16] Goffinet, Jack; Ramanujan, Raghuram: Monte-Carlo Tree Search for the Maximum Satisfiability Problem. In (Rueher, Michel, ed.): Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings. Springer International Publishing, Cham, pp. 251–267, 2016.

[Li15] Liang, Jia Hui; Ganesh, Vijay; Zulkoski, Ed; Zaman, Atulan; Czarnecki, Krzysztof: Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. In: Haifa Verification Conference. Springer, pp. 225–241, 2015.

[Lo13] Loth, Manuel; Sebag, Michèle; Hamadi, Youssef; Schoenauer, Marc: Bandit-based Search for Constraint Programming. In (Schulte, Christian, ed.): International Conference on Principles and Practice of Constraint Programming. volume 8124, Springer Verlag, Uppsala, Sweden, pp. 464–480, September 2013.

[MML12] Martins, Ruben; Manquinho, Vasco; Lynce, Inês: An overview of parallel SAT solving. Constraints, 17(3):304–347, 2012.

[MSLM09] Marques-Silva, Joao P.; Lynce, Ines; Malik, Sharad: Conflict-Driven Clause Learning SAT Solvers. chapter 4, pp. 131–153, 2009.

[Mu14] Munos, Rémi: From Bandits to Monte-Carlo Tree Search: The Optimistic Principle Applied to Optimization and Planning. Technical report, 2014. 130 pages.

[Pr11] Previti, Alessandro; Ramanujan, Raghuram; Schaerf, Marco; Selman, Bart: Monte-Carlo Style UCT Search for Boolean Satisfiability. In (Pirrone, Roberto; Sorbello, Filippo, eds): AI*IA 2011: Artificial Intelligence Around Man and Beyond: XIIth International Conference of the Italian Association for Artificial Intelligence, Palermo, Italy, September 15-17, 2011. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 177–188, 2011.

[SBSV88] Stephan, P.; Brayton, R.K.; Sangiovanni-Vincentelli, A.L.: Combinational test generation using satisfiability. Pointer swizzling at page fault time: efficiently and compatibly supporting huge address spaces on standard hardware, 7(9):1167–1176, 1988.

[Zh01] Zhao, Ying; Zhang, Lintao; Malik, Sharak; Madigan, Conor F.; Moskewicz, Matthew W.: Chaff: Engineering an Efficient SAT Solver. 2001.