Dinosat: A SAT Solver with Native DNF Support

Thomas Bartel[1], Tomáš Balyo[1], and Markus Iser[2]

[1] CAS Software, Karlsruhe, Germany
thomas.bartel@cas.de, tomas.balyo@cas.de
[2] Karslruhe Institute of Technology, Karlsruhe, Germany
markus.iser@kit.edu

Abstract

In this paper we report our preliminary results with a new kind of SAT solver called Dinosat. Dinosat's input is a conjunction of clauses, at-most-one constraints and disjunctive normal form (DNF) formulas. The native support for DNF formulas is motivated by the application domain of SAT based product configuration. A DNF formula can also be viewed as a generalization of a clause, i.e., a clause (disjunction of literals) is special case of a DNF formula, where each term (conjunction of literals) has exactly one literal. Similarly, we can generalize the classical resolution rule and use it to resolve two DNF formulas. Based on that, the CDCL algorithm can be modified to work with DNF formulas instead of just clauses. Using randomly generated formulas (with DNFs) we experimentally show, that in certain relevant scenarios, it is more efficient to solve these formulas with Dinosat than translate them to CNF and use a state-of-the-art SAT solver. Another contribution of this paper is identifying the phase transition points for such formulas.

1 Introduction

Most modern SAT solvers can only solve problems, that are defined in their "conjunctive normal form" (CNF) [6]. A few solvers support additional constraints natively, for example CryptoMiniSat [18] supports XOR constraints and Sat4j [12] supports cardinality constraints. In our case, we will support additional constraints in the form of disjunctive normal form (DNF) formulas, which will be treated similarly to clauses. This is inspired by the application domain of SAT based product configuration [11], where some product constraints are expressed as a list of allowed combinations of features, which can be best represented as a DNF formula.

Obviously, all these additional constraints can be translated to CNF, but it leads to a higher amount of variables and constraints. In addition to the lower amount of variables and clauses, supporting DNF natively has advantages that can be leveraged during the solving process, as we will show later.

The aim of this paper is to examine, whether a SAT solver, that is capable of natively solving DNF and AMO constraints in addition to the clauses, has advantages in the area of solving speed compared to more traditional modern SAT solvers. According to our experimental evaluation we can conclude that for certain types of DNF constraints it is beneficial to handle them natively.

2 Preliminaries

A *Boolean variable* has two possible values: *True* and *False*. A *literal* is a Boolean variable (positive literal) or a negation of a Boolean variable (negative literal). A *clause* is a disjunction ($\vee$) of literals and, finally, a CNF formula (or just formula) is a conjunction of clauses. A clause with only one literal is called a *unit clause*. A positive (resp. negative) literal is satisfied if the corresponding variable is assigned the value *True* (resp. *False*) A clause is satisfied, if at least one of its literals is satisfied and the formula is satisfied, if all its clauses are satisfied.

The satisfiability (SAT) problem is to determine whether a given formula has a satisfying assignment, and if so, also find it. The satisfiability of CNF formulas is an NP-Complete problem [7], nevertheless, it has many practical applications, therefore, a significant effort has been invested into developing efficient SAT solvers. Most SAT solvers take their input in the form of CNF formulas and are based on the DPLL algorithm [8] and its extension the CDCL algorithm [14, 15].

A conjunction of literals is called a term. A disjunctive normal form (DNF) formula is a disjunction of terms. In contrast to a CNF formula, the satisfiability of a DNF formula can be decided very easily. We only need to find a single term that is satisfiable, i.e., does not include a pair of contradictory literals (a literal and its negation).

If each term of a DNF formula contains exactly one literal, it has the same structure as a clause. Therefore, we can view DNF formulas as a generalization of clauses, and clauses as a special case of DNF formulas.

The resolution rule is defined for a pair of clauses. Let $(A \vee x)$ and $(B \vee \overline{x})$ be two clauses. Then the resolvent of these two clauses is $(A \vee B)$. The resolution rule can be easily generalized to work for a pair of DNF formulas (which also covers the pair DNF formula and clause): Let $(A \vee C \wedge x)$ and $(B \vee D \wedge \overline{x})$[1] be DNF formulas, then their resolvent is $(A \vee B)$.

Based on this generalized resolution, we can generalize the DPLL and CDCL algorithms to work with the conjunction of DNF formulas instead of conjunctions of clauses. By generalizing CDCL we do not only mean the core algorithm but also all the related optimizations and heuristics, such as the two-watched-literal scheme [16], VSIDS [16], LBD Scores [2], Luby restarts [13] and even most preprocessing/inprocessing alogorithms.

3 Related Work

This Section gives an overview of the different areas of SAT solving research, the current state of the art and how the approach of this paper differs from the current research topics.

3.1 Evolution of SAT Solvers

First we take a look at historically relevant SAT solvers and the concepts they introduced, which are still used by many modern SAT Solvers today.

In the paper "Chaff: Engineering an Efficient SAT solver" [16] the authors Moskewicz et al. describe the implementation of their SAT solver "CHAFF". The two big contributions this SAT solver made are the "two-watched-literals" algorithm and the "Variable State Independent Decaying Sum (VSIDS) heuristic", which are still used today in many SAT solvers. The two-watched-literals algorithm allows the SAT solver to only visit a clause if it is absolutely necessary and therefore offers a drastic reduction of the computational complexity. The VSIDS heuristic is a branching heuristic that assigns more weight to variables that recently appeared in newly learned clauses. It is still used today in many modern SAT solvers because it outperforms most other heuristics.

The paper "An Extensible SAT-solver" [9] by the authors Een and Sörensson describes the implementation of the minimal SAT solver "MiniSat". The paper presents an efficient implementation of a SAT solver that makes use of the CDCL-algorithm and is inspired by the

[1]where A and B are DNF formulas and C and D are terms.

SAT solver Chaff [16]. An important contribution of the MiniSat SAT solver, is a different implementation of the VSIDS heuristic, which will later on be referred to as exponential VSIDS (EVSIDS) in literature. The original VSIDS heuristic increments the score of every variable that is involved in the conflict and then the decays the score of every variable periodically. In constrast to that EVSIDS has no inbuilt score decay but instead increases the score of each conflict variable by exponentially growing values. In order to prevent overflow, the scores eventually have to be adjusted.

In the paper "Adapative Restart Strategies for Conflict Driven SAT Solver" [3] Armin Biere introdcues another implementation of VSIDS called the normalized VSIDS (NVSIDS) heuristic. The advantage that NVSIDS offers, are scores between 0 and 1. These can be interpreted as the percentage of times the variable was involved in a conflict. The disadvantage of NVSIDS is that the score of every variable needs to be adjusted after every conflict. Armin Biere then proves that EVSIDS produces the same variable order as NVSIDS. EVSIDS therefore has the same advantages as NVSIDS, while being more efficient because only the scores of variables involved in a conflict need an adjustment.

In the paper "Predicting learnt clauses quality in modern SAT solvers" [2] the authors Audemard and Simon describe their SAT solver "GLUCOSE", which also makes use of conflict driven clause learning. With their solver they try to solve the problem of CDCL-solvers either using up too much memory because of the learned constraints, or forgetting too many clauses and therefore making the solving process slower. The core of the authors work is the identification of good clauses to learn. They defined the "Literals Blocks Distance" (LBD), which is defined as the number of subsets of literals in a clause that belong to the same decision level. This score is then computed for every new learned clause and every clause can alternatively be re-scored when it is used in the unit propagation procedure. The authors highlight the importance of clauses with an LBD score of two. They only contain a single literal of the last decision level and are therefore a "First Unique Implication Point". The authors refer to those clauses as "Glue Clauses" [2]. The authors make use of the LBD score during the deletion process of learned clauses. They use a strategy that deletes half of all learned clauses every 20000 + 500 * x conflicts, with x referring to the number of conflicts that already occured. The LBD score is used to determine which of the learned clauses get deleted. The clauses with a lower LBD score get priority over the ones with high LBD scores.

In conclusion we can see that all of the SAT solvers mentioned above, offered new techniques in the areas of unit propagation, branching heuristics and learned clause database reduction, many of which are still used today in modern SAT solvers. One characteristic that all of these solvers share, is that they only work with clauses. In this work we will try to incorporate the techniques mentioned above, in a new SAT solver architecture that uses different constraint types.

3.2 SAT Solvers with Different Constraint Types

This subsection focuses on related work in the area of SAT solvers that support several constraint types.

The paper "To Encode or to Propagate? The Best Choice for Each Constraint in SAT" [1] by Abio et al. examines in which situations it is better to use a CNF encoding of constraints instead of implementing the native support directly in the SAT solver. They specifically talk about the work in the paper "Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)" [17], which features a solver that natively supports cardinality constraints. It also has the ability to resolve cardinality constraint conflicts by learning clauses. It can also adapt and change to strict CNF encoding if certain conditions are met. The results show that this adaptive strategy, which changes between native constraints and encodings, can have advantages in certain benchmarks. It can also be a hindrance because of the large amount of clauses that it can generate in order to resolve conflicts.

Another SAT solver that is capable of natively solving cardinality constraints and pseudo-boolean constraints was presented in the paper "The Sat4j Library 2.2, System Description" [12]. SAT4j contains two modules that each are capable of natively solving cardinality constraints. The module Sat4j PB Res uses simple conflict resolution by treating each of the constraints as simple clauses. This makes the resolution process easier. The module Sat4j PB CP uses the cutting planes method, which is more complex and therefore more computationally expensive. In most benchmarks the resolution based module performs better, but the cutting planes method is more effective in specific problems like the pigeon hole problem.

4 Analysis

Our solver supports two kinds of additional constraints: at-most-one and DNF formulas. However, due to space limitations, we will only describe the DNF extension in detail. It is the more novel and interesting one.

4.1 Basic SAT Solver Structure

This work is concerned with building a novel SAT solver architecture that is capable of solving basic CNF formulas like other modern SAT solvers, while also being capable of handling other types of constraints. The first important step is therefore the decision on what techniques our architecture is going to use in order to solve basic CNF formulas.

The DPLL algorithm is one of the most prominent algorithms for SAT solving. It is a depth-first search based algorithm and was developed in order to combat the high space requirements of the DP algorithm [6]. Because the DPLL algorithm uses a search based approach, the time that is needed to solve a formula, can be really long, depending on how large the formula is. This is the case because the DPLL algorithm just tries every possible combination of variable assignments, until a satisfying assignment is found. There are techniques that allows for a faster search like unit propagation, phase saving, branching heuristics and also restarts [6], but many modern SAT solvers still don't use a pure DPLL-based approach.

Instead the current best SAT solvers like Kissat [5] and CaDiCaL [4] make use of the CDCL algorithm. This allows for non-chronological backtracking and the learned clauses make sure that non satisfiable areas of the search space are never reached. This can reduce the solving time of boolean formulas drastically.

For this work the integration of different constraint types into these modern algorithms is the most important aspect. The usage of the DPLL algorithm with different constraint types would only allow us to explore optimization options in the area of unit propagations, branching heuristics and restarts, because there is no real conflict resolution. By using the CDCL algorithm it is possible to explore how the constraints interact directly with each other via conflicts, and how this can be leveraged by learning new constraints. Both approaches are interesting and because it is difficult to assess which of these algorithms offers the best performance when the formula contains different constraint types, we decided to implement both algorithms and evaluate their performance.

4.2 Constraint Database Reduction

We will use the LBD score based clause reduction algorithm [2]. The problem with implementing this type of clause deletion scheme in our novel SAT solver are the different types of constraints that it uses. The LBD score was specifically defined to measure the quality of clauses, but our SAT solver uses AMO and DNF constraints in addition to clauses. So in order to make us of this scheme we first have to define the LBD score for general constraints: Given a partial assignment of variables the LBD score of a constraint is the number of distinct decision levels **n** that can be counted in the constraint.

4.3 DNF Constraint

Our special constraint type is the "disjunctive normal form" (DNF) constraint.

$$DNF(x_1, ..., x_n) = \{x_1 \vee ... \vee x_n\}$$

$$x_i = y_1 \wedge ... \wedge y_m$$

The DNF is a disjunction of conjunctions. It is satisfied if at least a single conjunction is satisfied. This is the case if every literal in that conjunction is true. These conjunctions are called terms.

4.3.1 Unit Propagation

A unit propagation in a DNF can occur in one of two cases:

1. If there is only one non false term left, then all of its literals are unit literals
2. If all non false terms share a literal, then this literal is a unit literal.

The first case is trivial because at least one term needs to be satisfied in order for the DNF to be satisfied. Because there is only a single conjunction left, that can fulfill this condition, every literal of that term needs to be true.

For the second case consider a DNF with at least two terms that are currently undecided. The rest of the terms are false. If those terms share a literal x then both of them can only be true if and only if x is true. Because at least one of them needs to be true, x must be a unit literal.

Now consider a situation where those constraints don't share any literal. In this situation there are several possibilities on how the DNF can be satisfied and therefore there can't be a unit propagation yet.

4.3.2 Efficient Literal Propagation for DNF Constraints

The literal propagation process for DNF constraints is a difficult task, because the unit propagation starts when all non false terms are intersecting on at least one literal. So during the literal propagation process, the solver needs to keep track of these intersections, otherwise it could miss a unit propagation.

Generally the propagation needs to be efficient, because this process gets repeated many times during the solving procedure. The two-watched-literal algorithm for clauses achieves this by minimizing the amount of time where the solver visits clauses. So the first idea is to create an algorithm that achieves a similar property for DNF constraints. Because clauses are just a special case of DNF constraints, one could consider watching only two non false terms at a time. While this would minimize the time, that is spent by visiting the clauses, it is possible to create examples where this algorithm misses necessary unit propagations.

Consider the following constraints:

$$DNF((a,b),(a,c),(b,c,x))$$

$$DNF((a,b,c),(a,c,d),(a,b,d),(b,c,d,x))$$

This example shows a situation where the Two-Watched-Term algorithm misses a unit propagation. In the first DNF constraint there are three terms that are all pairwise intersecting. If the algorithm only watches two of these terms, then it is possible to construct a partial assignment where a unit propagation is missed. For this consider the situation that $\{(a,b),(a,c)\}$ are watched. If x turns false, then the DNF constraint isn't visited. The algorithm now misses the needed unit propagation of a. If a now turns false, then the DNF constraint isn't satisfied. In this situation the algorithm would have to watch all three of the terms. In a similar manner it is possible to construct a partial assignment, that leads to a missed unit propagation in the second DNF constraint. In this example the algorithm would have to watch four terms in order to not make a mistake. This can be extended for a variable amount of terms, so in order to only watch a subset of terms in every DNF constraint, the algorithm has to calculate how many terms it needs to watch to still be propagation complete.

A simple solution to this problem, is to just watch all terms in every DNF constraint. On every visit the algorithm can then just select the shortest non false term in the DNF constraints and check for intersections with all other non false terms. If there exists such a literal then it needs to be propagated. In order to speed up the search for the shortest term, it is efficient to already sort all terms by their size. But this algorithm has the problem that it results in many unnecessary visits to the DNF constraints. If the DNF constraints are very large then these visits take too much time.

4.3.3 The Two-Watched-Termsets-Algorithm

This section describes the Two-Watched-Termsets-Algorithm which is a compromise between the two approaches explained above. The algorithm changes its behavior depending on how many non intersecting terms are left in the DNF constraint. In order to understand how this algorithm ensures propagation completeness, it is necessary to look at all possible cases.

Unit DNF constraints: In order to understand the cases below, it is necessary to establish the term "unit DNF constraint". A unit DNF constraint shows similar behavior to a unit clause. A unit clause consists of a single literal and therefore forces this literal to always be true. A unit dnf constraint has the same property. A literal x that is contained in every term of a DNF constraint is a unit literal and needs to be propagated.

Because such a unit literal is always satisfied, the algorithm doesn't consider them during the calculation of intersections and they are therefore also not considered in the cases below.

First case: After the creation of the DNF constraint, the algorithm can't find any non-false non-intersecting terms. If this is the case then the algorithm automatically watches every term of the DNF constraint.

Second case: The DNF constraint contains two non-false non-intersecting terms. As already explained, a unit propagation in a DNF constraint can only occur if there is only one non-false term left or if all non-false terms intersect on a literal. So as long as the algorithm watches two non-false non-intersecting terms, it can never miss a unit propagation. So in this case the algorithm has to only watch these two terms.

So after the initial creation of a DNF constraint the algorithm either watches all the terms or only two terms. If a propagation occurs in a DNF constraint where all terms are watched, the algorithm can just check for literal intersection in all non-false terms and then propagate these intersections as unit literals

If a propagation occurs in a DNF constraint where only two terms are watched it is more complicated. Because both watched terms don't intersect, only one of them can turn false because of a propagation. The algorithm then has to try to replace the false watched term. This replacement also can't intersect with the currently watched non-false term. If the algorithm can find a suitable replacement then there is no need for a unit propagation. If it can't find a replacement then it now needs to watch all terms that intersect with the current non-false watched term. The behavior after this point is then similar to when all terms are watched.

A big advantage of this algorithm is that there is no backtracking needed in order to ensure the propagation completeness. At every point the algorithm either watches two non-false non-intersecting terms, or all non-false terms that are currently intersecting.

4.3.4 DNF - DNF Conflict

This section describes how a conflict between two DNF constraints can be resolved. The notation DNF/x_i means that every term that contains the literal x_i gets removed from the DNF constraint. Consider two DNF constraints DNF_1 and DNF_2 and the set of variables $\{x_1, ..., x_n\}$ that the two constraints are complementary on. Then every conflict between these two constraints can be resolved by learning the following constraints

$$\bigwedge_{i=1}^{n} DNF_1/x_i \vee DNF_2/\neg x_i$$

The two DNF constraints are complementary on the set of variables $\{x_1, ..., x_n\}$. If a term in DNF_1 that contains a literal x_k turn true, then every term in DNF_2 that contains the literal $\neg x_k$ turns false. The same principle is true when the situation is reversed and term in DNF_2 with the literal $\neg x_k$ turns true. That means that in order for both constraints to be satisfied either DNF_1/x_k or $DNF_2/\neg x_k$ needs to be true. This rule is true for every variable $\{x_1, ..., x_n\}$. Therefore the conflict can be resolved by learning the constraints $\bigwedge_{i=1}^{n} DNF_1/x_i \vee DNF_2/\neg x_i$.

As an example consider the following two DNF constraints:

$$DNF((a, b), (c, d))$$

$$DNF((e, \neg b), (f, \neg d))$$

Then the conflict can be resolved by learning the following constraints

$$DNF((a, b), (e, \neg b)) \wedge DNF((c, d), (f, \neg d))$$

4.3.5 How to Apply the Rule

The rule as it is defined above, allows the algorithm to learn a DNF constraint for every literal that the conflicting DNF constraints are complementary on. This can cost computation time, memory and might not make sense in a practical situation. Consider the following example:

$$DNF((a, b), (c, d), (x, y))$$

$$DNF((\neg a, e), (\neg c, f), (x, y))$$

Let's assume that the variables c, f and x turn false in this order. This causes a conflict on the variable a because the first constraint needs a to be true, while the second constraint needs $\neg a$ to be true. The constraints are complementary on the variables a and c. According to the rule from above, the conflict can be resolved by learning the constraints $DNF((c, d), (\neg c, f), (x, y))$ and $DNF((a, b), (\neg a, e), (x, y))$. Now if the literals turn false again in the same order, then the first learned DNF constraint will start a unit propagation of x and y after c and f turn false. This will then satisfy both constraints of the example. As we can see, in this example the constraint $DNF((a, b), (\neg a, e), (x, y))$ wasn't even involved in resolving the conflict from the example. So in order to spare memory and computation time it might make more sense to just learn the amount of constraints that are needed to resolve the current conflict. That's why in practice the following rule should be used:

Consider two DNF constraints DNF_1 and DNF_2 and the variable x that the two constraints are complementary on. Then a conflict between these two constraints that is caused by the variable x can be resolved by learning the following constraint:

$$DNF_1/x \vee DNF_2/\neg x$$

4.3.6 Alternative DNF-DNF Conflict Resolution

Even though the algorithm only has to learn a single DNF constraint in order to resolve a conflict between two DNF constraints, the following example shows that even learning a single DNF constraint can be a disadvantage for the solver.

A conflict occurs between these two constraints because d and f turn false.

$$(a \wedge b) \vee (d \wedge e)$$

$$(\neg a \wedge c) \vee (f \wedge g)$$

This conflict can be resolved by learning the following DNF constraint

$$(d \wedge e) \vee (f \wedge g)$$

The example shows a simple DNF-DNF conflict that can be resolved by the resolution rule that we defined. Now consider the following conversion of the learned DNF constraint as a CNF

$$(d \vee f) \wedge (d \vee g) \wedge (e \vee f) \wedge (e \vee g)$$

So by learning the new DNF constraint the algorithm effectively learns four new clauses. This number could grow exponentially with the size of the DNF constraint. An alternative to learning the DNF constraints would be to just learn the clause $(d \vee f)$. This clause is effective in avoiding the conflict from above. The conflict occurred because d and f turned false. This clause ensures that at least one of them is true. The conflict between these two DNF constraint can then still occur if for example both $e \vee g$ turn false. The algorithm can then just learn the clause $(e \vee g)$. By using this gradual approach we can ensure that the algorithm only learns a minimal amount constraints which makes the propagation easier later on. So the new conflict resolution rule can be defined the following way.

Consider two DNF constraints DNF_1 and DNF_2 and the variable x that the two constraints are complementary on. Let $\{y_1, ..., y_n\}$ be the literals of DNF_1 that turned false and therefore forced a unit propagation of the variable x. Let $\{z_1, ..., z_m\}$ be the literals of DNF_2 that turned false and therefore caused the propagation of the variable x. Then this specific conflict can be resolved by learning the following clause:

$$y_1 \vee ... \vee y_n \vee z_1 \vee ... \vee z_m$$

5 Evaluation

This Section will focus on the performance evaluation of our new SAT solver. First we will talk about how the different benchmarks for this evaluation were created. Then we will look at how our SAT solver performs in comparison to other modern state of the art SAT solvers. Since our solver is implemented in Java, we will compare it against the only available state-of-the-art SAT solver in Java – Sat4j [12].

5.1 Translating DNF and AMO Benchmarks

Our comparison is focused on how fast our solver can solve benchmarks that contain additional constraints, compared to how fast other modern SAT solvers can solve the pure CNF equivalents. So in order to make this comparison possible, we need to convert our benchmarks with DNF and AMO constraints to equisatisfiable CNF formulas. In order to encode the formulas with DNF constraints into a CNF equivalent, the solver encodes them with the Tseitin encoding [19]. It is used because it allows the solver to generate an equisatisfiable CNF that is linear in size of the original formula by introducing a new set of variables [6]. To convert the AMO constraints we use the trivial paiwise encoding for constraints with six or less literals. For larger AMO constraints we use the ladder encoding.

5.2 Phase Transitions of Random Benchmarks

There is a Phase Transition [10] for random 3-sat instances at a clause to variable ratio of about 4.3. At this ratio about 50% of the randomized 3-SAT instances are satisfiable. Phase transition radnom formulas also tend to be the hardest with respect to the number of variables. We decided to examine whether there exists a phase transition point for formulas with DNF constraints and if it also corresponds to the point with the hardest problems.

In each iteration our solver solves 1000 randomly generated formulas and checks the percentage of satisfiable formulas. With each iteration the constraint to variable ratio gets higher, because more constraints are added. The formulas are generated by iteratively adding more randomly generated constraints.

The clauses are generated in the same way as the original experiment, by drawing k distinct literals from a uniform distribution over n variables [10]. If the algorithm draws a variable twice for the same clause, then this clause is discarded.

For the DNF constraints the algorithm draws k distinct variables for each term. If a variable is drawn twice for a term, then the DNF constraint is thrown out. This process gets repeated until the desired amount of DNF constraints is generated.

For DNF constraints the amount of terms and the term size can be varied. We therefore decided to conduct two experiments. In the first experiment the DNF constraints all have a constant amount of terms and only the term size is varied. In the second experiment the term

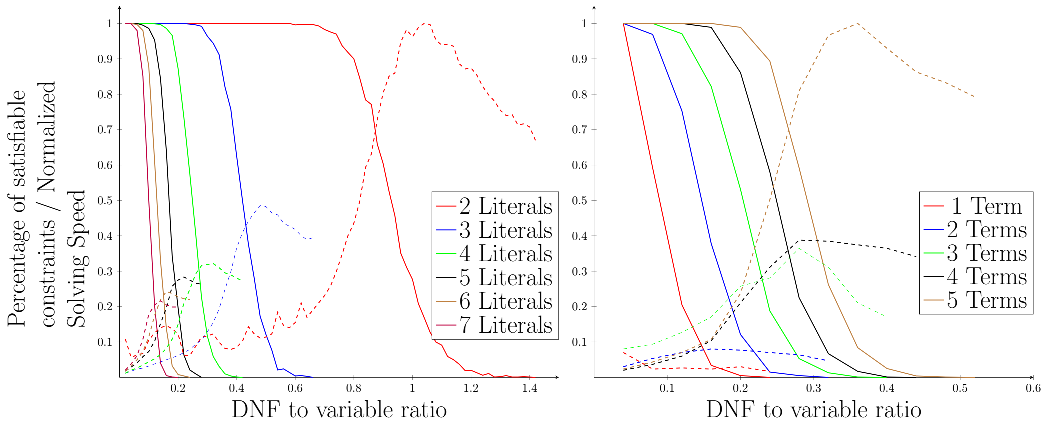

Figure 1: Phase transition and solving speed of DNF constraints with (Left) a constant term count of 3, 50 variables, and a DNF increment of 1 per iteration and (Right) with a constant term length of 5, 25 variables and a DNF increment of 1 per iteration

size is constant and the amount of terms is varied. In these two experiments formulas are conjunctions of only DNF constraints without any clauses.

The left graph in Figure 1 shows the relation between the DNF constraint to variable ratio and the percentage of satisfiable formulas, as well as the normalized solving time of 1000 instances. The dotted lines show the normalized solving time, while the continuous lines show the percentage of satisfiable constraints. Here we used DNF constraints with a constant term count of 3, 50 variables and a DNF increment of 1 per iteration. We then conducted the experiment for the term sizes 2-7. By adding more literals to the terms of the DNF constraints, the space of satisfiable constraints gets restricted, which is the likely explanation for why the phase transition occurs later the more literals are added to the terms.

In the next experiment we left the number of terms per DNF constraint constant while steadily increasing the amount of literals per term. The right graph in Figure 1 shows the relation between the DNF constraint to variable ratio and the percentage of the satisfiable constraints, as well as the normalized solving speed of 1000 instances. We use DNF constraints with a constant term length of five, 25 variables and a DNF increment of one per iteration. For the experiment we used DNF constraints with one to five terms. The dotted lines show the normalized solving time and the continuous lines show the percentage of satisfiable constraints. Here we see the opposite behavior of before. The more terms are added to the constraints, the later the phase transition occurs. A possible explanation for this, is that a higher amount of terms actually widens the space of possible solutions and therefore shift the phase transition point to a later point.

In the final phase transition experiment we generated formulas with both DNF constraints and regular clauses. The graph 2 shows the relation between the DNF constraint to variable ratio and the percentage of satisfiable formulas, as well as the normalized solving speed of 1000 instances. We used DNF constraints with 3 terms of length 3, 50 variables, DNF increments of 1 per iteration, clause increments of 50 per iteration and clauses of length 3. We can see that the phase transition of the DNF constraints shifts to an earlier point the more clauses are added.

In all three cases we can observe that the hardest instances seem to occur at around the phase transition point.

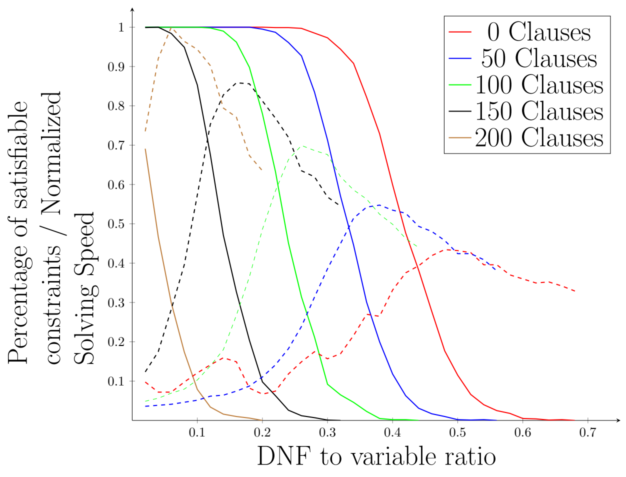

Figure 2: Phase transition of DNF constraint with 3 terms of length 3, 50 variables, DNF increments of 1, clauses of size 3 and clause increments of 50

Table 1: Composition of the industrial benchmark sets with the number of Formulas (F), the average number of variables (V), the average number of DNF constraints (DNF), the average number of terms per DNF (TPD), average number of literals per term (LPT), the average number of clauses (C), the average number of literals per clause (LPC), the average number of AMO constraints (AMO) and the average number of literals per AMO constraint

Name	F	V	DNF	TPD	LPT	C	LPC	AMO	LPA
Ind_{large}	20	27815.40	2645.55	7.50	32.38	114661.60	11.63	917.30	16.25
Ind_{medium}	21	13795.14	546.52	3.98	9.67	18504.38	5.87	393.00	12.40
Ind_{small}	26	6949.23	155.96	2.72	5.42	8626.42	4.02	134.19	16.00
$(CNF)Ind_{medium}$	21	19329.38	0.00	0.00	0.00	53697.86	3.71	0.00	0.00
$(CNF)Ind_{large}$	20	49742.00	0.00	0.00	0.00	797924.50	4.18	0.00	0.00
$(CNF)Ind_{small}$	26	9057.96	0.00	0.00	0.00	16935.81	3.15	0.00	0.00

5.3 Industrial Benchmarks

In order to assess how the SAT solver performs in real world scenarios, we used benchmarks from the SAT-based product configurator Merlin by CAS Software AG. These benchmarks are based on real industrial products and contain clauses, DNF constraints and AMO constraints. We converted these benchmarks into an equisatisfiable CNF by using the encodings already described above to compare the performance of our solver to Sat4j. It also allows us to examine whether our own architecture can leverage the advantages of the DNF and AMO constraints, or if the encoding is still the better alternative. The table 1 shows the composition of the Ind_{small}, Ind_{medium} and Ind_{large} industrial benchmark sets, as well as their equisatisfiable CNF counterparts.

The table 2 contains the performance evaluation for the three groups of industrial benchmark sets. We used a timeout of two minutes. First it is necessary to understand the naming of the different solver configurations. This naming scheme is consistent throughout this chapter. The configurations that start with "DPLL" make use of the DPLL algorithm instead of the CDCL algorithm. They therefore don't resolve conflicts by learning clauses. The "CDCL"

Table 2: Performance evaluation of the industrial benchmarks with the solving time (ST), number of solved instances (SI), number of timeouts (TO), average number of branching decisions (D(A)), average number of unit propagations (P(A)) and the average number of conflicts (C(A))

Solver	ST	SI	TO	D(A)	P(A)	C(A)
Large Industrial Benchmark Set						
$DPLL_{NR\_F}$	12.144	20	0	26318.40	9797.55	510.10
$DPLL_{R\_F}$	10.668	20	0	35914.35	15574.10	656.30
$CDCL_{R\_F}$	42.942	20	0	671542.00	945384.80	9085.85
$CDCL_{R\_F\_CNF}$	99.89	20	0	1030202.95	4973969.20	2583.40
SAT4J	18.867	20	0	82588.10	726588.50	48.25
Medium Industrial Benchmark Set						
$DPLL_{NR\_F}$	2.558	21	0	11403.43	2391.71	0.00
$DPLL_{R\_F}$	1.849	21	0	11403.43	2391.71	0.00
$CDCL_{R\_F}$	1.576	21	0	11403.43	2391.71	0.00
$CDCL_{R\_F\_CNF}$	2.086	21	0	10888.57	10895.29	1.76
SAT4J	1.579	21	0	6822.62	16236.81	0.00
Small Industrial Benchmark Set						
$DPLL_{NR\_F}$	1.799	26	0	5712.42	1245.27	2.15
$DPLL_{R\_F}$	0.876	26	0	5712.42	1245.27	2.15
$CDCL_{R\_F}$	1.011	26	0	6012.42	1895.88	4.08
$CDCL_{R\_F\_CNF}$	0.958	26	0	5485.58	6132.15	3.15
SAT4J	0.914	26	0	3900.38	8410.46	0.85

configurations learn clauses at every conflict. The configurations that contain "NR" don't use any restarts, while the configurations that contain an "R" make use of the Luby restarts. Then there is also the letter "F" which indicates that the first value, that a branching variable gets assigned, is always false. If a configuration starts with "(CNF)", then that means that the solver solved the equisatisfiable CNF version of this benchmark.

In the Large set both DPLL configurations of our solver show the fastest solving time, with the DPLL configuration, that uses restarts, taking the first place. Both CDCL configurations perform worse than SAT4J. So in this benchmark it seems, that natively solving the DNF and AMO constraints yields an advantage with the DPLL algorithm. An interesting observation is, that the average number of branching decisions, propagations and conflicts is way higher for both CDCL configurations compared to SAT4J. The DPLL configurations actually have the least amount of average branching decisions and propagations, but have more conflicts.

For the medium and small benchmark sets, the solving times for every configuration of our solver, as well as SAT4J, are so low, that it is difficult to draw conclusions on which solver actually performs better. The solving times are very similar for every solver and the small differences could also be attributed to the parsing time. What can be observed, is that SAT4J has the lowest amount of branching decisions out of every solver. A probable explanation for this, is that SAT4J uses a better branching heuristic, especially when combined with clause learning. Our solver seems to have the largest amount of branching decisions, when it uses clause learning. This indicates that our VSIDS implementation makes worse decisions, especially when new clauses are learned.

Table 3: DNF constraints with 15 terms each with 3 literals, performance evaluation with the solving time (ST), number of solved instances (SI), number of timeouts (TO), average number of branching decisions (D(A)), average number of unit propagations (P(A)) and the average number of conflicts (C(A))

Solver	ST	SI	TO	D(A)	P(A)	C(A)
Satisfiable Benchmark Set						
$DPLL_{NR\_F}$	169.861	99	0	29979.13	83177.88	29969.36
$DPLL_{R\_F}$	1495.094	99	0	265148.13	730542.03	263769.10
$CDCL_{R\_F}$	6019.026	70	29	348181.73	718620.80	270602.02
$CDCL_{R\_F\_CNF}$	10873.713	18	81	400048.77	46125526.05	119606.17
SAT4J	355.249	99	0	24897.67	6970444.00	22712.35
Unsatisfiable Benchmark Set						
$DPLL_{NR\_F}$	371.622	101	0	69360.25	192030.95	69361.25
$DPLL_{R\_F}$	5170.768	101	0	951018.38	2620153.44	946667.61
$CDCL_{R\_F}$	12120.204	0	101	691474.96	1439292.20	536616.08
$CDCL_{R\_F\_CNF}$	12120.63	0	101	479496.85	55210042.55	143715.06
SAT4J	802.786	101	0	51116.41	14393933.36	47212.17

5.4 Random Benchmark Runtimes

In addition to the provided industrial benchmark sets, we also used the knowledge from the phase transition experiments to generate a large amount of hard satisfiable and unsatisfiable random benchmark sets. First we will investigate how the solvers perform on formulas with DNF constraints that have a large amount of small terms.

The Table 3 shows the results of the performance evaluation. The results are similar for both satisfiable and unsatisfiable benchmarks. Our solver with the DPLL algorithm and no restarts is the clear winner. SAT4J takes the second place. What is also interesting to observe, is that the average amount of unit propagations of the configurations, that natively solve the DNF constraints, is now significantly lower compared to SAT4J This is probably because the encoding of the DNF constraints with a large amount of terms, results in a large amount of clauses. This in turn causes the amount of unit propagations to rise. Because our solver uses our Two-Watched-Termsets algorithm, it can take full advantage of the small terms, and therefore only has to watch a very small amount of literals, which is probably the reason for the small amount of unit propagations.

Most real world scenarios include several types of constraints, which is why we also wanted to examine how our SAT solver performs, when we create random benchmarks that consist of several constraint types. This section contains the performance evaluation on a randomized benchmark sets, where each formula contains DNF constraints and clauses.

The table 4 shows the results for these benchmarks. Our SAT solver has the fastest solving time with about 377.423 seconds, while using the DPLL algorithm with no restarts. SAT4J shows the second best performance with 633.809 seconds. All other configurations are significantly slower and both CDCL configurations weren't able to solve all formulas before the timeout. What can be observed again, is that the average amount of unit propagations is significantly higher for SAT4J than our solver with the DPLL configuration and no restarts. The DNF constraints are probably the cause of this, because we chose a combination of clauses and DNF constraints, where the DNF constraints have a large amount of small terms, like already explained above. If we chose a combination of clauses and small DNF constraints, then SAT4J

Table 4: DNF constraints with 15 terms with 3 literals each and additional 3-clauses, performance evaluation with the solving time (ST), number of solved instances (SI), number of timeouts (TO), average number of branching decisions (D(A)), average number of unit propagations (P(A)) and the average number of conflicts (C(A))

Satisfiable Benchmark Set						
Solver	ST	SI	TO	D(A)	P(A)	C(A)
$DPLL_{NR\_F}$	377.423	100	0	75034.69	259320.14	75024.31
$DPLL_{R\_F}$	4874.963	85	15	965176.94	3331950.47	960203.61
$CDCL_{R\_F}$	9325.511	36	64	542065.71	1333230.91	411110.29
$CDCL_{R\_F\_CNF}$	11267.317	12	88	458510.29	52227427.61	142576.51
SAT4J	633.809	100	0	45855.74	12760527.31	42233.01
Unsatisfiable Benchmark Set						
$DPLL_{NR\_F}$	953.813	100	0	196002.04	672922.24	196003.04
$DPLL_{R\_F}$	11994.235	5	95	2432477.67	8329660.62	2420712.23
$CDCL_{R\_F}$	12000.195	0	100	702954.86	1729444.16	532706.77
$CDCL_{R\_F\_CNF}$	12000.56	0	100	578466.55	65768664.11	180233.79
SAT4J	1386.435	100	0	96550.89	26889449.38	89580.13

would probably have outperformed our solver.

6 Conclusion

In this work we examined how modern SAT solving algorithms, like the DPLL and CDCL algorithm, that normally deal with pure CNF formulas, can be extended in order to also process DNF and AMO constraints. We then performed phase transition experiments on different combinations of the constraints, in order to find out, if there even exists a phase transition and if there is a correlation between the phase transition point and difficult randomized benchmarks. The experiments showed that there exists a phase transition for every combination of constraints. We then used these results to generate hard random benchmarks. For the performance evaluation we used those random benchmarks and also industrial benchmarks coming from the commercial SAT-based product configurator Merlin. We compared several different configurations of our solver with the Java based SAT solver "SAT4J". We observed, that for DNF constraints with a large amount of small terms, our solver was able to outperform SAT4J. This is both true for the industrial CAS benchmarks, as well as the randomized benchmarks. All in all we can conclude that there are benchmarks, where a native support for DNF constraints can be beneficial. to the solving time, and in some situations the new solver was able to outperform even SAT4J.

As for future work we would like to speed up the solving process of CNF formulas in general, which would then also lead to a higher solving speed, when solving the other constraints natively. Another area of improvement is the CDCL algorithm that works with the new constraints. The DPLL configuration of our solver was consistently able to outperform the CDCL configurations. Here the branching heuristic could also be the reason for this performance difference. Another idea is to examine if there are more conflicts on average, if a certain constraint type occurs in a benchmark. Then it might be necessary to use a more aggressive clause database reduction algorithm.

References

[1] Ignasi Abio, Robert Nieuwenhuis, Albert Oliveras, Enric Rodriguez-Carbonell, and Peter J Stuckey. To encode or to propagate? the best choice for each constraint in sat. In *International Conference on Principles and Practice of Constraint Programming*, pages 97–106. Springer, 2013.

[2] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Twenty-first International Joint Conference on Artificial Intelligence*, 2009.

[3] Armin Biere. Adaptive restart strategies for conflict driven sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 28–33. Springer, 2008.

[4] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.

[5] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[6] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[7] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

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

[9] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[10] Ian P Gent and Toby Walsh. The sat phase transition. In *ECAI*, volume 94, pages 105–109. PITMAN, 1994.

[11] Mikoláš Janota. *SAT solving in interactive configuration*. PhD thesis, University College Dublin, 2010.

[12] Daniel Le Berre and Anne Parrain. The sat4j library 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation To appear*, 2010.

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

[14] Joao P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[15] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[16] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[17] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[18] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.

[19] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.