Approximately Propagation Complete and Conflict Propagating Constraint Encodings

Rüdiger Ehlers and Francisco Palau Romero

University of Bremen

Abstract. The effective use of satisfiability (SAT) solvers requires problem encodings that make good use of the reasoning techniques employed in such solvers, such as unit propagation and clause learning. Propagation completeness has been proposed as a useful property for constraint encodings as it maximizes the utility of unit propagation. Experimental results on using encodings with this property in the context of satisfiability modulo theory (SMT) solving have however remained inconclusive, as such encodings are typically very large, which increases the bookkeeping work of solvers.

In this paper, we introduce approximate propagation completeness and approximate conflict propagation as novel SAT encoding property notions. While approximate propagation completeness is a generalization of classical propagation completeness, (approximate) conflict propagation is a new concept for reasoning about how early conflicts can be detected by a SAT solver. Both notions together span a hierarchy of encoding quality choices, with classical propagation completeness as a special case. We show how to compute approximately propagation complete and conflict propagating constraint encodings with a minimal number of clauses using a reduction to MaxSAT. To evaluate the effect of such encodings, we give results on applying them in a case study.

1 Introduction

Satisfiability (SAT) solvers have become an important tool for the solution of NP-hard practical problems. In order to utilize them, the practical problem to be solved needs to be encoded as a satisfiability problem instance, which is then passed to an off-the-shelf SAT solver. The way in which this encoding is performed has a huge influence on the solver computation times. Hence, the effective use of SAT solvers requires encodings that keep the workload of the solvers as small as possible. Capturing how an encoding needs to look like to have this property is however not a simple task. While it is commonly agreed on that problem-specific knowledge should be made use of, only few general guidelines for efficient encodings are known [7].

A good encoding should keep the numbers of variables *and* the number of clauses as small as possible, while allowing the solver to make most use of clause learning and unit propagation, which are reasoning steps that are (part of) the foundation of modern *CDCL-style* SAT solving [14]. While the effect of clause learning depends on the variable selection scheme for branching employed by the solver, and hence is hard to predict, how an encoding makes most use of unit propagation is better studied. For instance, the class of *cardinality constraints* has many known encodings [1,3], and it is frequently suggested that encodings that are *propagation complete* should be preferred [7,13]. An encoding of a constraint is propagation complete (also known under the name *generalized arc-consistent* [13]) if every literal implied by some partial valuation and the encoded constraint is detected by the unit propagation mechanism of the SAT solver. A constraint encoding with this property reduces the number of times in which costly backtracking is performed until a satisfying assignment is found or the SAT instance is found to be unsatisfiable.

Propagation completeness is of interest for all types of constraints that appear in practically relevant SAT problems, so having an automated way to make the encoding of a constraint type that appears as a *building block* in real-world problems propagation complete is likely to be useful. Brain et al. [9] presented an approach to rewrite SAT encodings to make them propagation complete. They apply their approach to multiple building blocks commonly found in problems from the field of formal verification and modify the SAT-based satisfiability modulo theory (SMT) solver `CVC4` to use the computed encodings for bitvector arithmetic operations. Their experiments show that the change increased solver performance somewhat, but made limited overall difference.

This result is surprising. If propagation complete encodings enable a SAT solver to make most use of unit propagation and reduce the number of conflicts during the solution process, then SAT solving times should decrease when using such encodings. A contributing factor to this lack of substantial improvement is that propagation complete encodings are often much larger than minimally-sized encodings. As an example, a three-bit multiplier (with five output bits) can be encoded with 45 clauses, but a propagation complete encoding needs at least 304 clauses. As a consequence, the bookkeeping effort of the solver is higher for propagation complete encodings, which reduces solving efficiency [17]. This observation gives rise to the question if there is a way to balance encoding size and the *propagation quality* of an encoding to get some of the benefits of propagation complete encodings but still keep the burden to the solver by the additional clauses low.

In this paper, we present such an approach to balance the propagation quality of a constraint encoding into conjunctive normal form (CNF) and its size. We define the novel notions of *approximate propagation completeness* and *approximate conflict propagation*. The former is a generalization of propagation completeness, and we say that a CNF formula ψ is approximately propagation complete for a quality level of $c \in \mathbb{N}$ if for every partial valuation to the variables in ψ that can be completed to a satisfying assignment and that implies at least c other variable values, one of them need to be derivable from ψ by unit propagation. Approximate conflict propagation is concerned with how early conflicts are detected. We say that a CNF constraint encoding ψ is approximately conflict propagating with a quality level of $c \in \mathbb{N}$ if every partial valuation that cannot be completed to one that satisfies ψ and for which the values of at most c variables are not set in the partial valuation leads to unit propagation (or induces a conflict) in ψ.

Approximate propagation completeness and approximate conflict propagation both target making the most use of the unit propagation capabilities of solvers. While approximate propagation completeness deals with *satisfiable* partial valuations, i.e., those that can be extended to satisfying assignments, approximate conflict propagation deals with *unsatisfiable* partial valuations. Together these concepts allow to reason about the *propagation quality* of CNF encodings in a relatively fine-grained way.

To evaluate the two new concepts, we present an approach to compute approximately propagation complete and approximately conflict propagating encodings with a minimal number of clauses. The approach starts from a representation of the constraint to be encoded as a binary decision diagram (BDD) and enumerates all shortest clauses implied by the BDD. Every minimal CNF encoding consists of a subset of these clauses. We then compute clause selection requirements for the solution based on the desired propagation quality levels. The resulting requirement set is then processed by a (partial) MaxSAT solver [20] to find a smallest encoding. The approach supports finding minimal propagation complete encodings and minimal arbitrary CNF encodings as special cases.

We apply the approach to a wide variety of constraints, including the ones already used by Brain et al. [9]. We show that their approach can sometimes produce encodings with a clause cardinality that is higher than necessary, and that for a good number of constraints, the various propagation quality level combinations for our new propagation quality notions give rise to many different (minimal) encodings with vastly different sizes. Our approach is also very competitive in terms of computation time when using the MaxSAT solver LMHS [25] in combination with the integer linear programming (ILP) solver CPLEX as backend. To gain some intuition on how efficient the SAT solving process with the new encodings is, we compare them on some *integer factoring* benchmarks.

1.1 Related Work

Brain et al. [9] introduced abstract satisfaction as a theoretical foundation to reason about propagation strength of encodings. They use a modified SAT solver to generate propagation complete encodings and then minimize their sizes by removing redundant clauses using a procedure proposed by Bordeaux and Marques-Silva [8]. As we show in Section 5, this approach does not guarantee a minimal number of clauses (but guarantees that no clause from the encoding can be removed without making it propagation incomplete or incorrect), whereas the new algorithm given in Section 4 does. Brain et al. also give a variant of their approach in which auxiliary variables are added to the SAT instance, which can substantially reduce the encoding size. This makes the encoding propagation incomplete, however, except when assuming that the SAT solver never branches on the auxiliary variables.

Inala et al. [17] used *syntax-guided program synthesis* to automatically compute propagation complete encodings that improve the efficiency of SMT-based verification. In contrast to the work by Brain et al., their approach synthesizes code to generate encodings rather than computing the encodings directly. The code can then be used for multiple concretizations of the constraint type (e.g., for varying bit vector widths when encoding an addition operation).

Bordeaux and Marques-Silva [8] already solved the problem of generating propagation complete encodings earlier than the aforementioned works. They show that when starting from a CNF encoding that should be made propagation complete, by restricting the search for clauses to be added to so-called *empowering clauses*, the computation time can be substantially reduced. However, their approach requires the use of a quantified Boolean formula (QBF) solver, whereas the later approach by Brain et al. [9] only requires a modified SAT solver. Bordeaux and Marques-Silva also showed that there are constraint classes that require exponentially-sized propagation complete encodings, which further motivates the study of approximate versions of this notion in the present work.

Gwynne and Kullmann [16,15] define two hierarchies for specifying how easy a SAT solver can detect implied literals and conflicts, which we also do in this paper. Their notions are however based on how often a solver has to branch in the optimal case to detect implied literals or conflicts. In contrast, our notions base on how *many* variable values are implied (for approximate propagation completeness) and how many variables do not have values assigned in a partial valuation (for approximate conflict propagation). Hence, our encoded constraints do not rely on the solver to branch on the right variables. This also allows us to automate the process of finding encodings with a minimal number of clauses for a wide variety of constraints, while the experimental evaluation of their work [15] focussed on few cases for which encodings in the levels of their hierarchies were available.

Minimal propagation complete CNF encodings typically have more clauses than minimal arbitrary CNF encodings. Many of the additional clauses can also be found automatically by SAT solvers that perform *preprocessing* or *inprocessing* [18] through techniques such as variants of hyper resolution [5]. Due to the high computational cost incurred by them, they are typically only used to a limited extent. Some clauses that are important for approximate propagation completeness and conflict propagation can therefore not be found or are found very late by these techniques. Furthermore, our approach computes minimal encodings from scratch, which be structured in completely different ways that expert-made encodings.

Manthey et al. [22] proposed a technique which can be used for inprocessing or preprocessing and that is based on introducing auxiliary variables that can make the CNF encoding smaller (where they count the number of clauses plus number of variables). They show that SAT solving performance improves on many benchmarks with their approach. While the introduction of additional variables could in principle break the propagation completeness, their approach does not suffer from this problem as they only undo clause distribution in a way that one of the variable phases occurs in two-literal clauses only. Their positive experimental results therefore do not give insight into the practical importance of using propagation complete encodings.

Babka et al. [4] study the theoretical properties of the problem of making a CNF encoding propagation complete. They identify the complexity classes of the different variants of this problem.

Kučera et al. [19] give lower bounds on the minimal number of clauses needed to encode so-called *"at most one"* or *"exactly one"* constraints. In contrast to the work in this paper, their result generalizes to an arbitrary size, also holds for the case that auxiliary variables are used, but is restricted to this particular constraint type.

2 Preliminaries

Given a set of variables $\mathcal{V}$ and Boolean formula ψ over $\mathcal{V}$, the *satisfiability* (SAT) problem is to find a satisfying assignment to the formula if one exists, or to deduce that no such assignment exists, in which case the formula is called *unsatisfiable*. Boolean formulas to be checked for satisfiability are also called *SAT instances* and are assumed to be given in *conjunctive normal form* (CNF) in the following. Such instances are conjunctions of *clauses*, which are in turn disjunctions of *literals*, i.e., from $\mathcal{L}(\mathcal{V}) = \mathcal{V} \cup \{\neg v \mid v \in \mathcal{V}\}$.

A *search-based SAT solver* maintains and manipulates a *partial valuation* to $\mathcal{V}$. A partial valuation $p : \mathcal{V} \rightharpoonup \{\mathbf{false}, \mathbf{true}\}$ is a partial function from the variables to **false** and **true**. We say that p is consistent with some other partial valuation p' if $p(v) = p'(v)$ holds for all variables v in the domain of p'. A *completion* of p is a full assignment to $\mathcal{V}$ that is consistent with p. We say that p satisfies some literal l over a variable v if $p(v)$ is defined and p can be completed to a full valuation that satisfies l. Likewise, p *falsifies* some literal l if $p(v)$ is defined and no completion of p satisfies l. We say that p' *implies* a literal $l \in \mathcal{L}(\mathcal{V})$ if every completion of p' that satisfies ψ also satisfies l. With a slight abuse of terminology, for some fixed set of clauses, we say that a partial valuation is *satisfiable* if it can be extended to a satisfying assignment, and it is *unsatisfiable* otherwise. We say that a clause c subsumes another clause c' if the literals of c are a subset of the literals of c'. If a CNF formula has two clauses c and c' such that c subsumes c', then c' can be removed without changing the encoded constraint.

During the search process, partial valuations $p : \mathcal{V} \rightharpoonup \{\mathbf{false}, \mathbf{true}\}$ are extended by the solver (1) by performing *decisions*, where the domain of p is extended by one variable, and (2) by *unit propagation*, where for some clause $l_1 \vee \ldots \vee l_n$ in the formula, there exists an $i \in \{1, \ldots, n\}$ such that the variable of l_i is not in the domain of p, but p falsifies all literals l_j with $i \neq j$.

Given some (sub-)set of variables $\mathcal{V}'$, a *constraint* over $\mathcal{V}'$ is a subset of valuations to $\mathcal{V}'$ that models the satisfaction of the constraint. A CNF encoding ψ of such a constraint is a set of clauses over $\mathcal{V}'$ that are together satisfied by exactly the valuations in the subset to be encoded. A CNF encoding of a constraint over a variable set $\mathcal{V}'$ is *propagation complete* if for every partial valuation p to $\mathcal{V}'$ and every literal $l \in \mathcal{L}(\mathcal{V}')$, if p implies l, then there exists a clause $l_1 \vee \ldots \vee l_n$ in the encoding for which all literals in the clause except for at most one are falsified by p.

Example 1. As an example, we consider the following CNF encoding:

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (x_4 \vee x_2 \vee x_3)$$

This constraint encoding is not propagation complete, as the partial valuation $p = \{x_4 \mapsto \mathbf{false}\}$ does not give rise to unit propagation, but in every satisfying valuation, x_1 needs to have a **false** value if x_4 has a **false** value. Shortening the last clause to $(x_2 \vee x_4)$ does not change the set of satisfying valuations and makes it propagation complete. The new clause enables the SAT solver to extend the partial valuation p to $p' = \{x_4 \mapsto \mathbf{false}, x_2 \mapsto \mathbf{true}\}$ by unit propagation, from where unit propagation can then deduce that x_1 must have a **false** value. Adding the clause $(\neg v_1 \vee x_4)$ instead of changing the clause $(v_4 \vee x_2 \vee x_3)$ would also make the encoding propagation complete.

Partial Maximum Satisfiability (MaxSAT) solvers take a CNF formula in which some clauses are *soft*. The solver searches for a variable assignment that satisfies all the remaining *hard* clauses and maximizes the number of soft clauses that are satisfied by the assignment.

Binary decision diagrams (BDDs) [10] are compact representations of Boolean formulas over some set of Boolean variables. They are internally represented as directed acyclic graphs and every path through the BDD to a designated **true** node represents one or multiple satisfying assignments of the formula. We will not need their details in the following, and refer to [11] for a well-accessible introduction. BDDs support the usual Boolean operators such as disjuction, complementation, and universal or existential abstraction of a variable. For instance, given a BDD F and a variable v, computing $\exists v.F$ yields a BDD that maps all valuations to the variables to **true** for which the value for v can be set to either **false** or **true** such that the resulting valuation is a model of F.

3 Approximate Propagation Completeness and Conflict Propagation

Propagation complete encodings enable search-based SAT solvers to deduce implied literals by unit propagation. By definition, *every* partial valuation that implies a literal over a variable that does not have a defined value in the partial valuation must give rise to unit propagation by the solver. We want to weaken this requirement in a way that enables SAT practitioners to better balance propagation quality and the encoding size of a constraint. This is done in two ways:

– We separate the consideration of satisfiable and unsatisfiable partial valuations, and

– we relax the requirement that every partial valuation that could give rise to unit propagation should do so, but rather that only partial valuations that enable the solver to make much progress need to so do.

These ideas are implemented in the following two *propagation quality notions* for CNF encodings:

Definition 1 (Approximate Propagation Completeness). *Given a CNF encoding ψ over some set of variables $\mathcal{V}$ and some constant $n \in \mathbb{N}$, we say that ψ is* approximately propagation complete *with a quality level of n if for all satisfiable partial valuations $p : \mathcal{V} \rightharpoonup \{\mathbf{false}, \mathbf{true}\}$ for which n different literals are implied by p and ψ and for which the variables for these literals are not in the domain of p, at least one of them can be derived from p and ψ by unit propagation.*

Definition 2 (Approximate Conflict Propagation). *Given a CNF encoding ψ over some set of variables $\mathcal{V}$ and some constant $n \in \mathbb{N}$, we say that ψ is* approximately conflict propagating *with a quality level of n if for all unsatisfiable partial valuations $p \in \mathcal{V} \rightharpoonup \{\mathbf{false}, \mathbf{true}\}$ for which p is defined on at least $|\mathcal{V}| - n$ variables, there exists a clause in ψ all of whose literals except for at most one are falsified by p.*

In both definitions, we only care about situations in which at least *one* clause should lead to unit propagation (or a conflict in case of approximate conflict propagation). The definition however induces stronger propagation quality requirements through repeated application. If, for instance, ψ is an approximately propagation complete encoding with a quality level of 2 and p is a partial valuation that implies four new literals, then at least one clause needs to lead to the derivation of an extended partial valuation p' by unit propagation. As p' then still implies three literals, by the fact that ψ has an approximate propagation completeness quality level of 2, another clause must give rise to unit propagation. By this line of reasoning, an *n-approximate propagation complete encoding* can never leave more than $n-1$ implied literals of a partial valuation undetected by unit propagation.

The quality level for approximate conflict propagation states how early conflicts induced by a constraint need to be detected by the unit propagation capabilities of the solver. In contrast to approximate propagation completeness, higher values are better as they mean that more variables can be unassigned in a partial valuation that already violates the constraint and where unit propagation should lead to an extension of the partial valuation or the detection of the conflict.

The requirement that p needs to be defined on at least $|\mathcal{V}| - n$ variables for a partial valuation to be of interest in the approximate conflict propagation definition could also be replaced by considering all partial valuations for which at *most* $|\mathcal{V}| - n$ variables are undefined. This definition would also make sense and requires that unsatisfiable partial valuations in which few variables have values assigned need to give rise to unit propagation until at most n variables are left unassigned. However, this definition would not ensure that conflicts are actually detected by the solver in such a case, while our definition does, which we find more natural. To see that our definition indeed ensures this, note that if a partial valuation p meets the requirements given in the definition, then either a conflict is detected or unit propagation should be able to extend p to a partial valuation p' that is defined on more variables and that still cannot be extended to a satisfying assignment. Hence, the definition can be applied again (repeatedly), and eventually a conflict is found by the solver. Thus, the "at least" in Definition 2 is reasonable, even though in this way, higher quality levels for approximate conflict propagation are better, whereas for approximate propagation completeness, lower numbers are better.

While these definitions consider satisfiable and unsatisfiable partial valuations separately, there is still a connection between them:

Proposition 1. *If a CNF encoding ψ over some set of variables $\mathcal{V}$ is approximately propagation complete with a quality level of* 1*, then it is (approximately) conflict propagating with a quality level of* $|\mathcal{V}|$.

Proof. Let p be an unsatisfiable partial valuation. We can transform p to an unsatisfiable partial valuation p' to which p is consistent and for which removing any variable of the domain of p' would make it satisfiable. This transformation only requires removing variables from the domain of p until no more variables can be removed without making it satisfiable. Let us now remove an arbitrary variable v' from the domain of p' and let the resulting valuation be called p''. We have that p'' implies the literal $l = \neg v'$ if $p(v') = \mathbf{true}$ or $l = v'$ if $p(v') = \mathbf{false}$. Since ψ is approximately propagation complete with a quality level of 1, l needs to be deduced by unit propagation from p''. The last clause used in the propagation has, by the definition of unit propagation, all literals instead of l falsified for the partial valuation p''. Since l is in conflict with p' by construction, we have $p''(v) = p'(v)$ for all variables v in the domain of p'', and we have that p is an extension of p', we obtain that p falsifies a clause of ψ. □

A corollary of this proposition is that approximately propagation complete encodings with a quality level of 1 are exactly the same as propagation complete encodings. While the two definitions differ for unsatisfiable partial valuations, they both imply that the unsatisfiability of a partial valuation needs to be detectable by unit propagation. For the approximate notion, this follows from the proposition above. For the older non-approximate propagation completeness definition, this follows from the fact that such partial valuations imply all literals in $\mathcal{L}(\mathcal{V})$, which in turn need to lead to at least one clause in the CNF encoding to be falsified.

4 Computing Minimal Approximately Optimal Encodings

In this section, we present an approach to compute approximately propagation complete and conflict propagating encodings of minimal size. Both of these concepts are parameterized by quality levels, so our procedure will read a *propagation quality level tuple* (q_p, q_c), where q_p denotes the quality level for approximate propagation completeness, and q_c is the quality level for approximate conflict propagation. Applying the approach with the quality level tuple $(1, |\mathcal{V}|)$ thus yields the smallest propagation complete encoding, whereas applying the approach with $(|\mathcal{V}|, 1)$ gives the overall smallest possible CNF encoding for a constraint (as every encoding is automatically approximately conflict propagating with a quality level of 1 by the definition of this propagation quality notion). To avoid the occurrence of the set of variables in propagation quality tuples in the following, we use the ∞ symbol to denote all numbers $\geq |\mathcal{V}|$, as the propagation quality level definitions do not lead to differences for values greater than or equal to the number of variables in a constraint.

The main idea of our approach is that after enumerating all clauses that *could* occur in a minimal CNF encoding with the specified quality level, we can compute requirements on the selection of a subset of clauses that ensure (1) the completeness of the encoding (i.e., that it accepts the correct set of full variable valuations), (2) the desired quality level for approximate propagation completeness, and (3) the desired quality level for approximate conflict propagation. These requirements are then encoded into a (partial) MaxSAT instance whose optimal solution represents a minimally-sized CNF encoding.

The following proposition gives rise to a procedure to efficiently enumerate the set of clauses that *can* occur in a minimal CNF encoding.

Proposition 2. *Let ψ be a CNF encoding of a constraint with propagation quality level tuple (q_p, q_c). If a literal can be removed from a clause in ψ without changing the set of satisfying variable valuations, then removing the literal does not degrade the propagation quality levels of ψ.*

Proof. If a partial valuation p leads to unit propagation for ψ, then it still does so after removing a literal from a clause *except* if the removed literal would be propagated. This can only happen for unsatisfiable partial valuations. For both propagation quality notions, such a case cannot reduce the quality level. $\square$

It follows that without loss of generality, we can assume smallest encodings to only use clauses that are as short as possible (in the sense that removing a literal from the clause would lead to a clause that some allowed variable valuation of the constraint to be encoded violates). Note that Bordeaux and Marques-Silva [8] already proved that when enumerating clauses for a constraint ordered by length, a longer clause can never make a shorter one redundant. Their result is not applicable here, as we later only select a subset of the enumerated *candidate clauses* to be contained in the computed encoding, while the application of their result requires that a clause remains a part of the CNF encoding once it has been found.

Enumerating all shortest clauses can be done in multiple ways. In our implementation for which we report experimental results in the next section, we start with a binary decision diagram description of the constraint and encode the search for a clause into a SAT instance by letting the SAT solver guess a partial valuation and which nodes in the BDD are reachable for the partial valuation. The **true** node of the BDD must not be reachable so that the valuation represents a possible clause in an encoding. We use a cardinality constraint with a ladder encoding [24] to count how many variables are set in the partial valuation, and iteratively search for partial valuations with the smallest possible domain size. Whenever one is found, a clause is added to the SAT instance that excludes all extensions of the partial valuation, and the search continues in an incremental manner.

After a *candidate* set of clauses $C = \{c_1, \ldots, c_m\}$ for the constraint to be encoded is computed, we employ a MaxSAT solver to find a minimal encoding with the desired propagation quality level tuple (q_p, q_c). The MaxSAT instance ϕ_M has the variables $x_1, \ldots, x_m$, i.e., one variable per clause in C, and we compute clauses for ϕ_M that ensure that the selected subset of C is (1) complete enough to encode the correct constraint, (2) approximately propagation complete with a quality level of q_p, and (3) approximately conflict propagating with a quality level of q_c. In the remainder of this section, we describe how to compute these clauses for ϕ_M.

4.1 Ensuring Encoding Correctness

All clauses in C can be part of a correct encoding of the constraint. The final set of selected clauses needs to be large enough not to allow spurious satisfying assignments, however. To achieve this, we recursively enumerate all (partial) assignments p while keeping track of the set of clauses C' not (yet) satisfied by the partial assignment. Whenever a complete assignment is reached and $C' \subseteq C$ is the set of clauses not satisfied by p, the (hard) MaxSAT clause $\bigvee_{c_i \in C'} x_i$ is added to ϕ_M. As optimizations for this process, a recursion step is aborted whenever C' becomes empty, and p is never extended by values for variables that do not occur in C'.

It is possible that a MaxSAT clause found late in this process subsumes a clause found earlier. We use a simple clause database structure that sorts clauses by length and removes subsumed clauses to avoid generating unnecessarily large MaxSAT instances. Storing all clauses with ordered literals increases the efficiency of the approach.

4.2 Encoding Approximate Propagation Completeness

By the definition of approximate propagation completeness, we need to ensure that for every partial valuation that implies at least q_p literals for variables that do not have values in the partial valuation, the final encoded CNF formula includes one clause that gives rise to unit propagation for the partial valuation.

We start by building a BDD that represents all partial valuations for which at least q_p new literals are implied. Algorithm 1 shows how this is done. In order to encode partial valuations, we use the auxiliary variable set $\mathcal{D} = \{d_v \mid v \in \mathcal{V}\}$ to encode which variables have *defined* values in a partial valuation. The algorithm iterates over all variables $v \in \mathcal{V}$ (line 3) and finds the set of partial valuations

Algorithm 1 Procedure to compute the satisfiable partial valuations that imply at least q_p many literals

```
 1: $F \leftarrow$ satisfying assignments of the constraint to be encoded over $\mathcal{V}$
 2: $X[0] \leftarrow F$
 3: **for** $v \in \mathcal{V}$ **do**
 4:     $I = \neg d_v \wedge (\exists v.F) \wedge (\neg \forall v.F)$
 5:     **for** $v' \in \mathcal{V} \setminus \{v\}$ **do**
 6:         $I = I \wedge ((\forall v'.I) \vee d_{v'})$
 7:     $Y \leftarrow X$
 8:     $X[0] \leftarrow Y[0] \wedge \neg I$
 9:     **for** $i \in \{1, \dots, |Y|\}$ **do**
10:         $X[i] \leftarrow (Y[i] \wedge \neg I) \vee (Y[i-1] \wedge I)$
11:     $X[|Y|] \leftarrow Y[|Y|-1] \wedge I$
12: **return** $\bigvee_{i=q_p}^{|\mathcal{V}|} X[i]$
```

that imply v or $\neg v$ (line 4). We only consider partial valuations that can be extended to satisfying assignments, so only one of them can be implied. If some other variable v' does not have a defined value in a partial valuation, then the valuation can only induce a literal over v if the value of v' does not matter for implying v (lines 5 to 6). The resulting partial valuation set is stored into the variable I in the algorithm.

After I is computed, the algorithm updates a partitioning of the partial valuations by how many literals over the variables already considered in the outer loop are induced by the respective partial valuation (lines 7 to 11). Finally, the algorithm returns the partial valuations that induce at least q_p literals.

For every partial valuation in the resulting BDD, we compute the subset $C' \subseteq C$ of clauses in C that give rise to unit propagation over the valuation, as for the final CNF encoding of the constraint to be approximately propagation complete with a quality level of q_p, one of them needs to be contained. We then add $\bigvee_{c_i \in C'} x_i$ as a hard clause to the MaxSAT instance ϕ_M and update the BDD with the remaining partial valuations to be considered by taking its conjunction with $\neg \bigwedge_{c_i \in C'} \bigvee_{l \in c_i} \bigwedge_{l' \in c_i, l \neq l'} m(l)$, where $m(\neg v) = d_v \wedge v$ and $m(v) = d_v \wedge \neg v$ for every variable $v \in \mathcal{V}$. In this way, all partial valuations that are guaranteed to lead to unit propagation whenever at least one of the clauses in C' is contained in the CNF encoding are removed from the BDD. We do the same with all clause subsets C' found in the procedure given in the previous subsection, as this further reduces the number of partial valuations to be considered. As before, we use a special clause database for ϕ_M to remove subsumed clauses.

4.3 Encoding Approximate Conflict Propagation

Next, we take care of partial valuations that cannot be completed to satisfying assignments and for which at most q_c variables in $\mathcal{V}$ do not have an assigned value. Algorithm 2 describes how to compute this set of partial valuations. After

Algorithm 2 Procedure to compute the unsatisfiable partial valuations for which at least q_c variables do not have a assigned values.

```
1: $X[0] \leftarrow \neg$(satisfying assignments of the constraint to be encoded over $\mathcal{V}$)
2: for $v \in \mathcal{V}$ do
3:     $Y \leftarrow X$
4:     for $i \in \{1, \ldots, |X|\}$ do
5:         $T \leftarrow (\forall v.Y[i]) \wedge \neg d_v$
6:         if $i > 1$ then
7:             $T \leftarrow T \vee (Y[i-1] \wedge d_v)$
8:         $X[i] \leftarrow T$
9:     $X[|Y|+1] \leftarrow Y[|Y|] \wedge d_v$
10: return $\bigvee_{i=|\mathcal{V}|-q_c}^{|V|} X[i]$
```

it has been computed, the process is exactly the same as in the previous subsection: for every partial valuation that the BDD maps to **true**, the subset of clauses C' that lead to a conflict or unit propagation for this partial valuation is computed, and the (hard) MaxSAT clause $\bigvee_{c_i \in C'} x_i$ is added to the MaxSAT instance. The BDD for the remaining partial valuations is also updated in the same way and the clauses in the MaxSAT clause database generated by the procedures in the preceding two subsections are used to reduce the number of partial valuations to be considered before enumerating them.

4.4 MaxSAT solving

All constraints in ϕ_M so far are *hard* constraints, i.e., need to be fulfilled by all satisfying variable valuations of the MaxSAT instance. To request the solver to minimize the number of clauses in the CNF encoding, we add the *soft clauses* $\neg x_1, \ldots, \neg x_m$. Maximizing the number of satisfied soft clauses then exactly corresponds to minimizing the number of clauses in the final constraint encoding.

5 Experiments

We implemented the approximate propagation complete and conflict propagating CNF encoding procedure in `C++` using the BDD library `CuDD` [26]. Unlike the tool `GenPCE` for computing propagation complete encodings with the approach by Brain et al. [9], our new `optic` tool[1] does not start with a constraint encoding in CNF, but supports arbitrary Boolean formulas as constraints, which makes their specification easier. For the following experiments, we use `lingeling bbc 9230380` [6] as SAT solver for enumerating the candidate clauses for the encoding, and apply the MaxSAT solver `LMHS` [25] in the version from the end of March 2018, using `minisat` [12] as backend solver. `LMHS` performs calls to an integer linear programming solver, for which we employ IBM `CPLEX V12.8.0`. We use the following benchmark sets:

[1] Available at `https://github.com/progirep/optic`

1. Full adders with and without carry bit output
2. Multipliers of various input and output bit widths
3. Square and square root functions
4. Unsigned less-than-or-equal comparison functions
5. All-different constraints, including Cook's encoding [23]
6. Bipartite matching problems, where for some set of nodes $V_1 = V_2 = \{1, \ldots, n\}$ for $n \in \mathbb{N}$ and (random) $E \subseteq V_1 \times V_2$, we have one Boolean variable v_e for every $e \in E$ and allow all edge subset choices encoded into $\{v_e\}_{e \in E}$ for which every node in V_1 is predecessor node for exactly one chosen edge and every node in V_2 is the successor node of exactly one chosen edge
7. All other benchmarks from the work of Brain et al. [9] not contained in the benchmark families above

We apply our implementation with various propagation quality tuples and compare the resulting encoding sizes and computation times with the propagation complete encodings generated by the GenPCE tool and the time it took that tool to compute them. As baseline, we also compare against a simple BDD-based tool that enumerates some shortest possible clauses in a CNF encoding until the encoding is complete, without guaranteeing any propagation quality. All benchmarks were executed on a computer with AMD Opteron 2220 processors with 2.8 GHz clock rate, running an x64 version of Linux. The memory limit for every execution was 8 GB, all executions where single-threaded, and we imposed a time limit of 30 minutes.

Table 1 shows an excerpt of the results. We tested the propagation quality of all generated encodings using an additional tester tool, and report the results in the table as well. In many cases, `optic` generated encodings of higher propagation quality levels than requested when this was possible without increasing the number of clauses in the encoding. A few other observations can be made:

1. The `GenPCE` tool did not always compute propagation complete encodings with a minimal number of clauses (e.g, all-different constraints for 3 objects, some bipartite matching problems), but did so quite often.
2. There are a many cases (e.g., addition with a large number of bits, multiplication, all-different constraints) for which the encodings for different propagation quality tuples have different sizes.
3. Encodings that are fully conflict propagating but for which approximate propagation completeness was not requested are typically small.
4. The `GenPCE` tool is often slower than `optic` for computing propagation complete encodings, but the scalability of `optic` and `GenPCE` are quite similar. However, `optic` also times out in a few cases in which `GenPCE` does not. Computing minimally-sized encodings (as `optic` does but not `GenPCE`) appears to be much harder in these cases.

We can see that using BDDs for enumerating candidate clauses and generating MaxSAT clauses is not a major bottleneck, as `optic` frequently outperforms `GenPCE` on propagation-complete encodings. We also tested how long computing good encodings for the four-object all-different constraints takes. For the $(1, \infty)$ and $(2, \infty)$ propagation quality levels, it was determined after 18.1 and 28.4 hours that 200 and 156 clauses are needed, respectively, of which 11.2 and 621.3 minutes were spent solving the MaxSAT problems. In the $(3, \infty)$ case, the overall computation took 141.7 hours (120 clauses).

5.1 Case study: Integer Factoring

To evaluate the effect of the encodings on propagation quality, we apply them in an integer factoring case study. We generated 5 numbers that are products of two primes each. For each number c and each $2 \leq n_1 \leq \lceil \log_2(\sqrt{c+1}) \rceil$, we then computed SAT instances for finding a factoring for which the first number has n_1 bits with the most significant bit set. We compose the SAT instance of encodings for full adders with 1, 2, 3, or 4 input bits, and multipliers with 1, 2, or 3 input bits. We use the propagation quality tuples $(3, 3)$, (∞, ∞),$(\infty, 1)$, and $(1, \infty)$, for which the latter three refer to best possible conflict propagation, minimal encoding size, and classical propagation completeness. For the $(3, 3)$ case, we were unable to generate a minimal encoding for the four-bit full adder and had to use a suboptimal (possibly too large) encoding that we obtained by using the MaxSAT solver `maxino-2015-k16` [2] that can output suboptimal solutions found early. We aborted its computation after 400 minutes. We also compare the solving performance against using the *greedy* encoding introduced as comparison baseline in Table 1.

Figure 1 shows the cactus plot for the computation times of the SAT solvers `lingeling bbc 9230380` [6] and `MapleSAT_LRB` [21] as representatives for modern solvers with and without advanced *inprocessing*, respectively. It can be seen that for shorter time limits, minimally sized encodings without propagation quality guarantees are inferior in overall performance. Above 600 seconds of computation time for every benchmark, `MapleSAT` however works quite well with minimally-sized encodings, but the difference between the encodings is quite small for high time limits anyway. `Lingeling` works best with encodings of higher propagation quality. Encodings that only enforce conflict propagation seem to be particularly well suited for easier benchmarks.

6 Conclusion

We presented two new propagation quality notions and described an approach to compute minimally-sized CNF encodings from constraint descriptions. Our approach reduces the problem to solving a single MaxSAT instance, and the experiments show that many constraints found in practice give rise to differently sized encodings for different propagation quality level combinations.

In contrast to the work by Brain et al. [9], we based our experimental case study on problems encoded from scratch rather than modifying an existing SMT solver, as the techniques used in SMT solvers are highly tuned to work in concert, and hence replacing a single element has side-effects that cannot be tuned out in an experimental evaluation.

Benchmark	**Greedy**			(∞, ∞)			$(1, \infty)$			$(2, \infty)$			$(3, \infty)$			$(3, 3)$			$(\infty, 1)$			**GenPCE**		
All-Difference Cook 3 obj.	36	2.64	$(11, 10)$	timeout			timeout			timeout			timeout			timeout			17	20.74	$(11, 11)$	33	0.00	$(1, \infty)$
All-Difference 3 obj.	26	0.00	$(7, 6)$	15	1.15	$(4, \infty)$	24	0.24	$(1, \infty)$	20	0.35	$(2, \infty)$	15	0.39	$(3, \infty)$	15	0.34	$(3, \infty)$	12	0.10	$(4, 6)$	26	0.00	$(1, \infty)$
All-Difference Cook 4 obj.	125	6.33	$(\times, 23)$	timeout			timeout			timeout			timeout			timeout			timeout			memout		
All-Difference 4 obj.	86	0.10	$(13, 12)$	timeout			timeout			timeout			timeout			timeout			28	74.80	$(8, 12)$	203	0.00	$(1, \infty)$
cvc-add15to4.cnf	65534	1633.63	$(\times, \infty)$	timeout			timeout			timeout			timeout			timeout			timeout			timeout		
cvc-add3-bw3.cnf	824	1.74	$(6, 8)$	144	2.31	$(4, \infty)$	1536	1.64	$(1, \infty)$	808	8.30	$(2, \infty)$	512	3.64	$(3, \infty)$	500	47.51	$(3, 4)$	122	10.49	$(6, 8)$	1536	19.81	$(1, \infty)$
cvc-add3-carry2-gadget.cnf	62	0.07	$(2, \infty)$	36	0.23	$(6, \infty)$	76	0.10	$(1, \infty)$	62	0.16	$(2, \infty)$	51	15.18	$(3, \infty)$	51	6.90	$(3, \infty)$	32	0.41	$(6, 4)$	76	0.00	$(1, \infty)$
cvc-add3-opt-bw3.cnf	824	0.88	$(6, 8)$	144	1.46	$(4, \infty)$	1152	1.22	$(1, \infty)$	624	2.97	$(2, \infty)$	416	2.11	$(3, \infty)$	416	2.66	$(3, \infty)$	126	1.02	$(6, 8)$	1152	16.98	$(1, \infty)$
cvc-add7to3.cnf	254	0.23	$(1, \infty)$	timeout			254	2.72	$(1, \infty)$	254	16.03	$(1, \infty)$	254	32.34	$(3, \infty)$	254	17.04	$(3, \infty)$	158	20.98	$(7, 5)$	254	0.55	$(1, \infty)$
cvc-mult-const-const3-2n-2.cnf	12	0.22	$(5, \infty)$	8	0.25	$(3, \infty)$	11	0.31	$(1, \infty)$	9	0.25	$(2, \infty)$	8	0.30	$(3, \infty)$	8	0.32	$(3, \infty)$	8	0.30	$(3, \infty)$	11	0.00	$(1, \infty)$
cvc-mult-const-const5-2n-3.cnf	27	0.08	$(7, 6)$	16	0.31	$(7, \infty)$	24	0.36	$(1, \infty)$	22	0.38	$(2, \infty)$	19	0.38	$(3, \infty)$	19	0.73	$(3, 6)$	13	0.40	$(8, 7)$	24	0.00	$(1, \infty)$
cvc-mult-const-const7-2n-3.cnf	30	0.25	$(7, 6)$	15	1.70	$(8, \infty)$	32	1.25	$(1, \infty)$	24	2.41	$(2, \infty)$	20	3.03	$(3, \infty)$	20	3.58	$(3, \infty)$	14	3.07	$(8, 6)$	32	0.00	$(1, \infty)$
cvc-plus-3.cnf	48	0.08	$(6, 4)$	36	0.28	$(5, \infty)$	96	0.12	$(1, \infty)$	72	0.28	$(2, \infty)$	56	0.28	$(3, \infty)$	56	0.40	$(3, \infty)$	32	0.39	$(5, 6)$	96	0.09	$(1, \infty)$
cvc-plus-4.cnf	120	0.21	$(8, 7)$	60	1.39	$(6, \infty)$	336	1.42	$(1, \infty)$	243	1.64	$(2, \infty)$	158	3.08	$(3, \infty)$	158	5.16	$(3, \infty)$	54	6.35	$(7, 9)$	336	5.53	$(1, \infty)$
cvc-slt-gadget.cnf	6	0.08	$(1, \infty)$	6	0.13	$(1, \infty)$	6	0.09	$(1, \infty)$	6	0.08	$(1, \infty)$	6	0.09	$(1, \infty)$	6	0.13	$(1, \infty)$	6	0.11	$(1, \infty)$	6	0.00	$(1, \infty)$
cvc-ult-6.cnf	158	0.28	$(1, \infty)$	158	0.39	$(1, \infty)$	158	0.47	$(1, \infty)$	158	0.26	$(1, \infty)$	158	0.36	$(1, \infty)$	158	0.49	$(1, \infty)$	158	0.42	$(1, \infty)$	158	38.42	$(1, \infty)$
cvc-ult-gadget.cnf	6	0.06	$(1, \infty)$	6	0.07	$(1, \infty)$	6	0.07	$(1, \infty)$	6	0.07	$(1, \infty)$	6	0.07	$(1, \infty)$	6	0.07	$(1, \infty)$	6	0.06	$(1, \infty)$	6	0.00	$(1, \infty)$
Adder, 2 bits, carry output	23	0.04	$(3, \infty)$	18	0.06	$(5, \infty)$	30	0.05	$(1, \infty)$	27	0.07	$(2, \infty)$	22	0.06	$(3, \infty)$	21	0.06	$(3, 4)$	17	0.05	$(5, 4)$	30	0.00	$(1, \infty)$
Adder, 3 bits, carry output	61	0.00	$(6, 4)$	42	0.15	$(6, \infty)$	102	0.31	$(1, \infty)$	93	0.15	$(2, \infty)$	65	0.68	$(3, \infty)$	65	19.23	$(3, \infty)$	34	0.68	$(7, 7)$	102	0.20	$(1, \infty)$
Adder, 4 bits, carry output	149	0.09	$(9, 8)$	66	2.11	$(8, \infty)$	342	2.16	$(1, \infty)$	303	2.48	$(2, \infty)$	188	3.00	$(3, \infty)$	timeout			56	42.89	$(9, 10)$	342	14.57	$(1, \infty)$
Adder, 4 bits	120	0.06	$(8, 7)$	60	1.30	$(6, \infty)$	336	1.33	$(1, \infty)$	243	1.37	$(2, \infty)$	158	3.28	$(3, \infty)$	158	273.06	$(3, \infty)$	54	6.62	$(7, 9)$	336	5.77	$(1, \infty)$
Adder, 5 bits, carry output	343	0.38	$(11, 11)$	90	76.98	$(9, \infty)$	1086	64.51	$(1, \infty)$	948	76.16	$(2, \infty)$	560	106.05	$(3, \infty)$	timeout			80	376.57	$(11, 13)$	1086	848.67	$(1, \infty)$
Adder, 5 bits	272	0.36	$(10, 10)$	84	24.39	$(8, \infty)$	1080	17.00	$(1, \infty)$	762	20.37	$(2, \infty)$	446	32.59	$(3, \infty)$	timeout			78	136.79	$(8, 12)$	1080	353.66	$(1, \infty)$
Multiplier, 2 to 2 bits	12	0.00	$(2, \infty)$	12	0.06	$(2, \infty)$	12	0.06	$(1, \infty)$	12	0.06	$(2, \infty)$	12	0.06	$(2, \infty)$	12	0.05	$(2, \infty)$	12	0.06	$(2, \infty)$	12	0.00	$(1, \infty)$
Multiplier, 2 to 4 bits	24	0.06	$(2, \infty)$	16	0.08	$(4, \infty)$	19	0.09	$(1, \infty)$	17	0.09	$(2, \infty)$	17	0.09	$(3, \infty)$	17	0.08	$(3, \infty)$	16	0.08	$(4, \infty)$	19	0.00	$(1, \infty)$
Multiplier, 3 to 3 bits	31	0.05	$(5, 4)$	29	0.06	$(5, \infty)$	71	0.07	$(1, \infty)$	41	0.13	$(2, \infty)$	39	0.06	$(3, \infty)$	39	0.05	$(3, \infty)$	29	0.09	$(6, 5)$	71	0.00	$(1, \infty)$
Multiplier, 3 to 5 bits	98	0.07	$(8, 7)$	timeout			304	3.63	$(1, \infty)$	165	3.95	$(2, \infty)$	124	10.12	$(3, \infty)$	123	6.01	$(3, 5)$	45	25.33	$(8, 7)$	305	2.10	$(1, \infty)$
Multiplier, 4 to 5 bits	212	0.05	$(9, 9)$	timeout			2274	112.48	$(1, \infty)$	1106	121.12	$(2, \infty)$	timeout			timeout			timeout			2276	40.92	$(1, \infty)$
Multiplier, 5 to 7 bits	1327	0.57	$(\times, \times)$	timeout			timeout			timeout			timeout			timeout			timeout			timeout		
Square root, 4 bits	10	0.04	$(3, \infty)$	8	0.11	$(2, \infty)$	11	0.08	$(1, \infty)$	8	0.08	$(2, \infty)$	8	0.08	$(2, \infty)$	8	0.08	$(2, \infty)$	8	0.10	$(2, \infty)$	11	0.00	$(1, \infty)$
Square root, 5 bits	14	0.30	$(3, \infty)$	14	0.39	$(3, \infty)$	18	0.30	$(1, \infty)$	15	0.41	$(2, \infty)$	14	0.36	$(3, \infty)$	14	0.37	$(3, \infty)$	14	0.41	$(3, \infty)$	18	0.00	$(1, \infty)$
Square root, 6 bits	28	0.65	$(5, 5)$	22	0.87	$(5, \infty)$	41	0.84	$(1, \infty)$	29	0.86	$(2, \infty)$	24	0.87	$(3, \infty)$	24	0.80	$(3, \infty)$	22	0.86	$(5, \infty)$	41	0.08	$(1, \infty)$
Square root, 7 bits	41	2.28	$(7, 7)$	35	2.64	$(7, \infty)$	71	2.83	$(1, \infty)$	51	2.78	$(2, \infty)$	45	2.79	$(3, \infty)$	45	2.92	$(3, \infty)$	35	2.77	$(7, 7)$	71	0.00	$(1, \infty)$
Square root, 8 bits	74	5.03	$(9, 8)$	54	7.23	$(8, \infty)$	209	6.63	$(1, \infty)$	124	6.73	$(2, \infty)$	103	8.13	$(3, \infty)$	103	8.61	$(3, \infty)$	52	6.03	$(9, 7)$	209	3.78	$(1, \infty)$
Square function, 4 bits	52	0.07	$(8, 6)$	28	1.01	$(9, \infty)$	56	0.62	$(1, \infty)$	44	0.65	$(2, \infty)$	35	0.93	$(3, \infty)$	35	0.69	$(3, \infty)$	26	0.32	$(9, 8)$	56	0.07	$(1, \infty)$
Square function, 5 bits	109	0.08	$(11, 10)$	timeout			273	254.83	$(1, \infty)$	184	752.77	$(2, \infty)$	timeout			135	1443.43	$(3, \infty)$	42	54.10	$(12, 10)$	273	3.37	$(1, \infty)$
Square function, 6 bits	228	0.25	$(\times, 13)$	timeout			timeout			timeout			timeout			timeout			timeout			1701	195.27	$(1, \infty)$
Square function, 7 bits	492	0.41	$(\times, \times)$	timeout			timeout			timeout			timeout			timeout			timeout			timeout		
Bipatite matching, 3 obj's, ex. 12	9	0.05	$(1, \infty)$	7	0.08	$(3, \infty)$	8	0.08	$(1, \infty)$	8	0.08	$(1, \infty)$	7	0.07	$(3, \infty)$	7	0.07	$(3, \infty)$	7	0.08	$(3, \infty)$	9	0.00	$(1, \infty)$
Bipatite matching, 4 obj's, ex. 59	16	0.07	$(6, 4)$	11	0.19	$(4, \infty)$	14	0.08	$(1, \infty)$	14	0.06	$(2, \infty)$	11	0.11	$(3, \infty)$	11	0.20	$(3, \infty)$	10	0.05	$(4, 6)$	16	0.00	$(1, \infty)$

Table 1. Number of computed clauses, computation times (in seconds) and measured propagation quality for a selection of benchmarks and a selection of target propagation quality tuples. A $\times$ entry in the measured propagation quality tuples means that the quality level could not be determined due to a timeout (set to 60 seconds) of the respective measuring tool. We compare against a greedy encoding into CNF (by enumerating shortest possible clauses using BDDs) and against the tool `GenPCE` that computes propagation complete encodings. Computation times are given in seconds. The output of the the greedy tool is used as input to `GenPCE` (as that tool requires the input to already be in CNF form, with possibly some variables that need to be existentially quantified away). The $(1, \infty)$ quality tuple refers to propagation completeness, which `GenPCE` also guarantees. The $(\infty, 1)$ quality tuple refers to plain minimally-sized encodings. Encodings of quality level (∞, ∞) are (fully) conflict propagating, but not necessarily propagation complete.

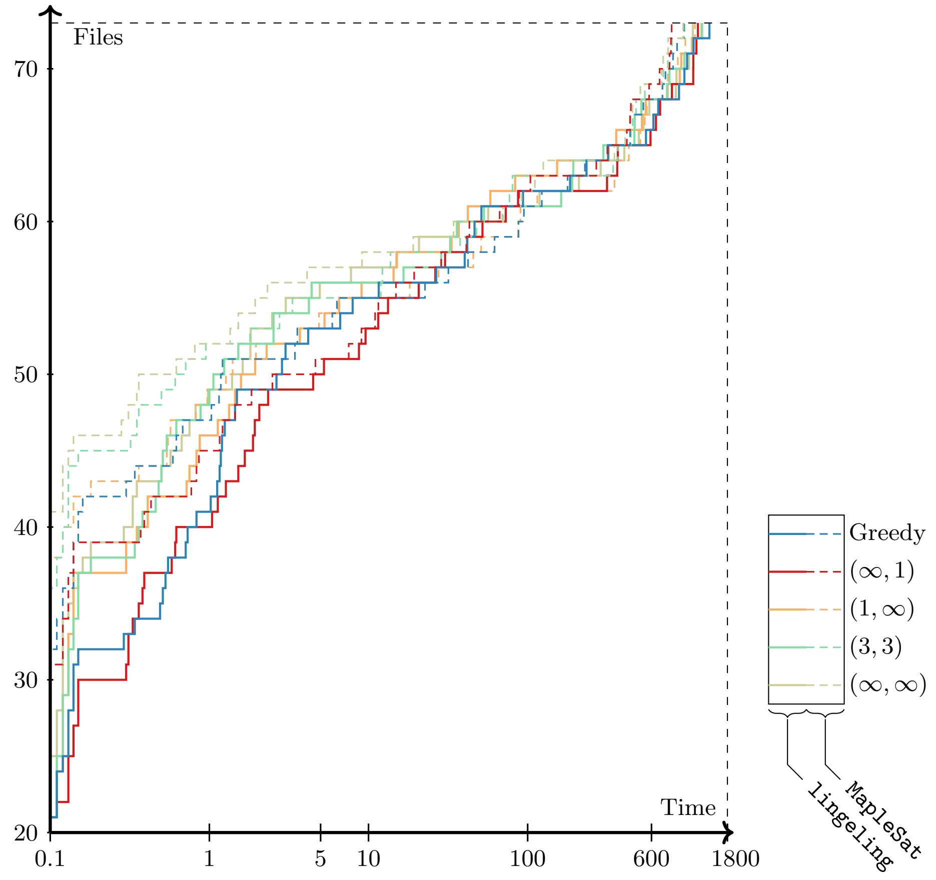

Fig. 1. Cactus plot for the integer factoring case study. Time is given in seconds, the overall number of files is 73. The legend describes the line styles for the studied combinations of solvers and propagation quality tuples.

Studying the precise effect of *inprocessing* [18] on what constraint encodings can be used most efficiently is left for future work. For instance, clauses that solvers with inprocessing could automatically derive could be left out, which may give rise to very different minimally-sized encodings.

Acknowledgements

This work was supported by DFG grant EH 481/1-1 and the Institutional Strategy of the University of Bremen, funded by the German Excellence Initiative. The authors want to thank Armin Biere for early feedback on the propagation quality notions defined in this work and Erika Abraham for proposing MaxSAT solvers as reasoning backend.

References

1. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: A parametric approach for smaller and better encodings of cardinality constraints. In: Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. pp. 80–96 (2013), http://dx.doi.org/10.1007/978-3-642-40627-0_9
2. Alviano, M., Dodaro, C., Ricca, F.: A MaxSAT algorithm using cardinality constraints of bounded size. In: Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 2677–2683 (2015), http://ijcai.org/Abstract/15/379
3. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks: a theoretical and empirical study. Constraints 16(2), 195–221 (2011), http://dx.doi.org/10.1007/s10601-010-9105-0
4. Babka, M., Balyo, T., Cepek, O., Gurský, S., Kucera, P., Vlcek, V.: Complexity issues related to propagation completeness. Artif. Intell. 203, 19–34 (2013), http://dx.doi.org/10.1016/j.artint.2013.07.006
5. Bacchus, F., Winter, J.: Effective preprocessing with hyper-resolution and equality reduction. In: Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. pp. 341–355 (2003), https://doi.org/10.1007/978-3-540-24605-3_26
6. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010. FMV Report Series Technical Report 10/1, Johannes Kepler University, Linz, Austria (2010)
7. Bjork, M.: Successful SAT encoding techniques. Journal on Satisfiability, Boolean Modeling and Computation Addendum Paper (2009)
8. Bordeaux, L., Marques-Silva, J.: Knowledge compilation with empowerment. In: SOFSEM 2012: Theory and Practice of Computer Science - 38th Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 21-27, 2012. pp. 612–624 (2012), https://doi.org/10.1007/978-3-642-27660-6_50
9. Brain, M., Hadarean, L., Kroening, D., Martins, R.: Automatic generation of propagation complete SAT encodings. In: Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. pp. 536–556 (2016), http://dx.doi.org/10.1007/978-3-662-49122-5_26
10. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Computers 35(8), 677–691 (1986), https://doi.org/10.1109/TC.1986.1676819
11. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001), http://books.google.de/books?id=Nmc4wEaLXFEC
12. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. pp. 502–518 (2003), https://doi.org/10.1007/978-3-540-24605-3_37
13. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. JSAT 2(1-4), 1–26 (2006), http://jsat.ewi.tudelft.nl/content/volume2/JSAT2_1_Een.pdf

14. Franco, J., Martin, J.: A history of satisfiability. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, chap. 1, pp. 3–74. IOS Press (2009)
15. Gwynne, M., Kullmann, O.: Towards a theory of good SAT representations. CoRR abs/1302.4421 (2013), http://arxiv.org/abs/1302.4421
16. Gwynne, M., Kullmann, O.: Generalising unit-refutation completeness and SLUR via nested input resolution. J. Autom. Reasoning 52(1), 31–65 (2014), https://doi.org/10.1007/s10817-013-9275-8
17. Inala, J.P., Singh, R., Solar-Lezama, A.: Synthesis of domain specific CNF encoders for bit-vector solvers. In: Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016. pp. 302–320 (2016), http://dx.doi.org/10.1007/978-3-319-40970-2_19
18. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. pp. 355–370 (2012), https://doi.org/10.1007/978-3-642-31365-3_28
19. Kucera, P., Savický, P., Vorel, V.: A lower bound on CNF encodings of the at-most-one constraint. In: Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017. pp. 412–428 (2017), https://doi.org/10.1007/978-3-319-66263-3_26
20. Li, C.M., Manyà, F.: MaxSAT, hard and soft constraints. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, chap. 19, pp. 613–631. IOS Press (2009)
21. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016. pp. 123–140 (2016), https://doi.org/10.1007/978-3-319-40970-2_9
22. Manthey, N., Heule, M., Biere, A.: Automated reencoding of Boolean formulas. In: Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers. pp. 102–117 (2012), http://dx.doi.org/10.1007/978-3-642-39611-3_14
23. Petke, J.: Bridging Constraint Satisfaction and Boolean Satisfiability. Artificial Intelligence: Foundations, Theory, and Algorithms, Springer (2015), http://dx.doi.org/10.1007/978-3-319-21810-6
24. Roussel, O., Manquinho, V.: Pseudo-boolean and cardinality constraints. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, chap. 22, pp. 695–733. IOS Press (2009)
25. Saikko, P., Berg, J., Järvisalo, M.: LMHS: A SAT-IP hybrid MaxSAT solver. In: Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016. pp. 539–546 (2016), https://doi.org/10.1007/978-3-319-40970-2_34
26. Somenzi, F.: CUDD: CU Decision Diagram package release 3.0.0 (2015)