

Propagation = Lazy Clause Generation

Olga Ohrimenko¹, Peter J. Stuckey¹, and Michael Codish²

¹ NICTA Victoria Research Lab, Department of Comp. Sci. and Soft. Eng. University of Melbourne, Australia

² Department of Computer Science, Ben-Gurion University, Israel

Abstract. Finite domain propagation solvers effectively represent the possible values of variables by a set of choices which can be naturally modelled as Boolean variables. In this paper we describe how we can mimic a finite domain propagation engine, by mapping propagators into clauses in a SAT solver. This immediately results in strong nogoods for finite domain propagation. But a naive static translation is impractical except in limited cases. We show how we can convert propagators to lazy clause generators for a SAT solver. The resulting system can solve scheduling problems significantly faster than generating the clauses from scratch, or using Satisfiability Modulo Theories solvers with difference logic.

1 Introduction

Propagation is an essential aspect of finite domain constraint solving which tackles hard combinatorial problems by interleaving search and restriction of the possible values of variables (propagation). The propagators that make up the core of a finite domain propagation engine represent tradeoffs between the speed of inference of information versus the strength of the information inferred. Good propagators represent a good tradeoff at least for some problem classes. The success of finite domain propagation in solving hard combinatorial problems arises from these good tradeoffs, and programmable search.

Propositional satisfiability (SAT) solvers are becoming remarkably powerful and there is an increasing number of papers which propose encoding hard combinatorial (finite domain) problems in SAT. The success of modern SAT solvers is largely due to a combination of techniques including: watch literals, 1UIP nogoods and the VSIDS variable ordering heuristic [13].

In this paper we propose modelling combinatorial problems in SAT, not by modelling the constraints of the problem, but by modelling/mimicking the propagators used in a finite domain model of the problem. Variables are modelled in terms of the changes in domain that occur during the execution of propagation. We can then model the domain changing behaviour of propagators as clauses.

Encoding finite domain propagation uncovers an Achilles' heel of SAT solvers. While modern SAT solvers can often handle problems with millions of constraints and hundreds of thousands of variables, many problems are difficult to encode into SAT without breaking these implicit limits. We propose a hybrid approach.

Instead of introducing clauses representing propagators *a priori*, we execute the original (finite domain) propagators as lazy clause generators inside the SAT solver. Propagators introduce their propagation clauses precisely when they are able to trigger new unit propagation. The resulting hybrid combines the advantages of SAT solving, in particular powerful and efficient nogood learning and backjumping, with the advantages of finite domain propagation, simple and powerful modelling and specialized and efficient propagation of information.

This paper contributes a hybrid system for implementing propagation-based finite domain solving with a SAT solver and demonstrates its successful application to hard open-shop scheduling benchmarks. We compare the hybrid solver with a static approach that introduces the propagation clauses *a priori*, and with Satisfiability Modulo Theories (SMT) [14] solving using difference logic. Our prototype implementation can significantly improve on the carefully engineered SAT and SMT solvers.

In the remainder of the paper we first introduce terminology, and then propagation rules, a method of understanding propagator behaviour. We show how these can be expressed as CNF formulae, and introduce the lazy clause generation approach. After experiments we compare with related work and conclude.

2 Propagation-Based Constraint Solving

We consider a typed set of variables $\mathcal{V} = \mathcal{V}_I \cup \mathcal{V}_S$ made up of *integer* variables, \mathcal{V}_I , and *sets of integers* variables, \mathcal{V}_S . We use lower case letters such as x and y for integer variables and upper case letters such as S and T for sets of integers. A *domain* D is a complete mapping from \mathcal{V} to finite sets of integers (for the variables in \mathcal{V}_I) and to finite sets of finite sets of integers (for the variables in \mathcal{V}_S). We can understand a domain D as a formula $\bigwedge_{v \in \mathcal{V}} (v \in D(v))$ stating for each variable v that its value is in its domain.

Let D_1 and D_2 be domains and $V \subseteq \mathcal{V}$. We say that D_1 is *stronger* than D_2 , written $D_1 \sqsubseteq D_2$, if $D_1(v) \subseteq D_2(v)$ for all $v \in \mathcal{V}$ and that D_1 and D_2 are *equivalent modulo* V , written $D_1 =_V D_2$, if $D_1(v) = D_2(v)$ for all $v \in V$. The *intersection* of D_1 and D_2 , denoted $D_1 \sqcap D_2$, is defined by the domain $D_1(v) \cap D_2(v)$ for all $v \in \mathcal{V}$.

We use *range* notation: For integers l and u , $[l..u]$ denotes the set of integers $\{d \mid l \leq d \leq u\}$, while for sets of integers L and U , $[L..U]$ denotes the set of sets of integers $\{A \mid L \subseteq A \subseteq U\}$. A *convex* domain D is where $D(T)$ is a range for all $T \in \mathcal{V}_S$. We restrict attention to convex domains. We assume an *initial domain* D_{init} which is convex such that all domains D that occur will be stronger i.e. $D \sqsubseteq D_{init}$.

A *valuation* θ is a mapping of integer and set variables to correspondingly typed values, written $\{x_1 \mapsto d_1, \dots, x_n \mapsto d_n, S_1 \mapsto A_1, \dots, S_m \mapsto A_m\}$. We extend the valuation θ to map expressions or constraints involving the variables in the natural way. Let *vars* be the function that returns the set of variables appearing in an expression, constraint or valuation. In an abuse of notation, we define a valuation θ to be an element of a domain D , written $\theta \in D$, if $\theta(v) \in D(v)$ for all $v \in \text{vars}(\theta)$.