Lazy Clause Generation: Combining the power of SAT and CP (and MIP?) solving

Peter J. Stuckey

NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
University of Melbourne, 3010 Australia. pjs@cs.mu.oz.au

Abstract. Finite domain propagation solving, the basis of constraint programming (CP) solvers, allows building very high-level models of problems, and using highly specific inference encapsulated in complex global constraints, as well as programming the search for solutions to take into account problem structure. Boolean satisfiability (SAT) solving allows the construction of a graph of inferences made in order to determine and record effective nogoods which prevent the searching of similar parts of the problem, as well as the determination of those variables which form a tightly connected hard part of the problem, thus allowing highly effective automatic search strategies concentrating on these hard parts. Lazy clause generation is a hybrid of CP and SAT solving that combines the strengths of the two approaches. It provides state-of-the-art solutions for a number of hard combinatorial optimization and satisfaction problems. In this invited talk we explain lazy clause generation, and explore some of the many design choices in building such a hybrid system, we also discuss how to further incorporate mixed integer programming (MIP) solving to see if we can also inherit its advantages in combinatorial optimization.

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 trade-offs between the speed of inference of information versus the strength of the information inferred. Good propagators represent a good trade-off at least for some problem classes. The success of finite domain propagation in solving hard combinatorial problems arises from these good trade-offs, and programmable search, and has defined the success of constraint programming (CP).

Boolean Satisfiability (SAT) solvers have recently become remarkably powerful principally through the combination of: efficient engineering techniques for implementing inference (unit propagation) using watched literals, effective methods for generating and recording nogoods which prevent making a set of decisions which has already proven to be unhelpful (in particular 1UIP nogoods), and efficient search heuristics which concentrate on the hard parts of the problem combined with restarting to escape from early commitment to choices. These changes, all effectively captured in Chaff [1], have made SAT solvers able to solve problems orders of magnitude larger than previously possible.

Can we combine these two techniques in a way that inherits the strengths of each, and avoids their weaknesses. *Lazy clause generation* [2, 3] is a hybridization of the two approaches that attempts to do this. The core of lazy clause generation is simple enough, we examine a propagation based solver and understand its actions as applying to an underlying set of Boolean variables representing the integer (and set of integer) variables of the CP model.

In this invited talk we will first introduce the basic theoretical concepts that underlie lazy clause generation. We discuss the relationship of lazy clause generation to SAT modulo theories [4]. We then explore the difficulties that arise in the simple theoretical hybrid, and examine design choices that ameliorate some of these difficulties. We discuss how complex global constraints interact with lazy clause generation. We then examine some of the remaining challenges for lazy clause generation: incorporating the advantages of mixed integer programming (MIP) solving, and building hybrid adaptive search strategies. The remainder of this short paper will simply introduce the basic concepts of lazy clause generation.

2 Lazy Clause Generation by Example

The core of lazy clause generation is fairly straightforward to explain. An integer variable x with initial domain $[\,l\,..\,u\,]$ is represented by two sets of Boolean variables $[\![x \leq d]\!], l \leq d < u$ and $[\![x = d]\!], l \leq d \leq u$. The meaning of each Boolean variable $[\![c]\!]$ is just the condition c. In order to prevent meaningless assignments to these Boolean variables we add clauses that define the conditions that relate them.

$$\begin{aligned}
[\![x \leq d]\!] &\rightarrow [\![x \leq d+1]\!], l \leq d \leq u-2 \\
[\![x = d]\!] &\rightarrow [\![x \leq d]\!], l \leq d \leq u-1 \\
[\![x = d]\!] &\rightarrow \neg[\![x \leq d-1]\!], l < d \leq u \\
[\![x \leq d]\!] \wedge \neg[\![x \leq d-1]\!] &\rightarrow [\![x = d]\!], l < d \leq u-1 \\
[\![x \leq l]\!] &\rightarrow [\![x = l]\!] \\
\neg[\![x \leq u-1]\!] &\rightarrow [\![x = u]\!]
\end{aligned}$$

Each Boolean variable encodes a domain change on the variable x. Setting $[\![x = d]\!]$ true sets variable x to d. Setting $[\![x = d]\!]$ false excludes the value d from the domain of x. Setting $[\![x \leq d]\!]$ true creates an upper bound d on the variable x. Setting $[\![x \leq d]\!]$ false creates a lower bound $d+1$ on the variable x. We can hence mimic all domain changes using the Boolean variables. More importantly we can record the *behaviour* of a finite domain propagator using clauses over these variables.

Consider the usual bounds propagator for the constraint $x = y \times z$ (see e.g. [5]). Suppose the domain of x is $[\,-10\,..\,10\,]$, y is $[\,2\,..\,10\,]$ and z is $[\,3\,..\,10\,]$. The bounds propagator determines that the lower bound of x should be 6. In doing so it only made use of the lower bounds of y and z. We can record this as a clause c_1

$$(c_1) : \neg[\![y \leq 1]\!] \wedge \neg[\![z \leq 2]\!] \rightarrow \neg[\![x \leq 5]\!]$$

It also determines the upper bound of z is 5 using the upper bound of x and the lower bound of y, and similarly the upper bound of y is 3. These can be recorded as

$$(c_2) : [\![x \leq 10]\!] \wedge \neg[\![y \leq 1]\!] \rightarrow [\![z \leq 5]\!]$$
$$(c_3) : [\![x \leq 10]\!] \wedge \neg[\![z \leq 2]\!] \rightarrow [\![y \leq 3]\!]$$

Similarly if the domain of x is $[\,-10\,..\,10\,]$, y is $[\,-2\,..\,3\,]$ and z is $[\,-3\,..\,3\,]$, the bounds propagator determines that the upper bound of x is 9. In doing so it made use of both the upper and lower bounds of y and z. We can record this as a clause

$$\neg[\![y \leq -3]\!] \wedge [\![y \leq 3]\!] \wedge \neg[\![z \leq -4]\!] \wedge [\![z \leq 3]\!] \rightarrow [\![x \leq 9]\!]$$

In fact we could strengthen this explanation since the upper bound of x will remain 4 even if the lower bound of z was -4, or if the lower bound of y were -3. So we could validly record a stronger explanation of the propagation as

$$\neg[\![y \leq -3]\!] \wedge [\![y \leq 3]\!] \wedge \neg[\![z \leq -5]\!] \wedge [\![z \leq 3]\!] \rightarrow [\![x \leq 9]\!]$$

or

$$\neg[\![y \leq -4]\!] \wedge [\![y \leq 3]\!] \wedge \neg[\![z \leq -4]\!] \wedge [\![z \leq 3]\!] \rightarrow [\![x \leq 9]\!]$$

In a lazy clause generation system every time a propagator determines a domain change of a variable it records a clause that *explains* the domain change. We can understand this process as *lazily* creating a clausal representation of the information encapsulated in the propagator. Recording the clausal reasons for domain changes creates an implication graph of domain changes. When conflict is detected (an unsatisfiable constraint) we can construct a reason for the conflict, just as in a SAT (or SMT solver).

Suppose the domain of x is $[\,6\,..\,20\,]$, domain of y is $[\,2\,..\,20\,]$, z is $[\,3\,..\,10\,]$ and t is $[\,0\,..\,20\,]$ and we have constraints $x \leq t$, $x = y \times z$ and $y \geq 4 \vee z \geq 7$. Suppose search adds the new constraint $t \leq 10$ (represented by $[\![t \leq 10]\!]$). The inequality changes the upper bounds of x to 10 with explanation $(c_4) : [\![t \leq 10]\!] \rightarrow [\![x \leq 10]\!]$. The multiplication changes the upper bounds of z to 5 ($[\![z \leq 5]\!]$), and y to 3 ($[\![y \leq 3]\!]$) with the explanations c_2 and c_3 above, and the disjunctive constraint (which is equivalent to $(c_5) : \neg[\![y \leq 3]\!] \vee \neg[\![z \leq 6]\!]$) makes $\neg[\![z \leq 6]\!]$ true which by the domain constraints makes $(c_6) : [\![z \leq 5]\!] \rightarrow [\![z \leq 6]\!]$ unsatisfiable. The implication graph is illustrated in Figure 1

We can explain the conflict by any cut that separates the conflict node from the earlier parts of the graph. The first unique implication point (1UIP) cut chooses the closest literal to the conflict where all paths from the last decision

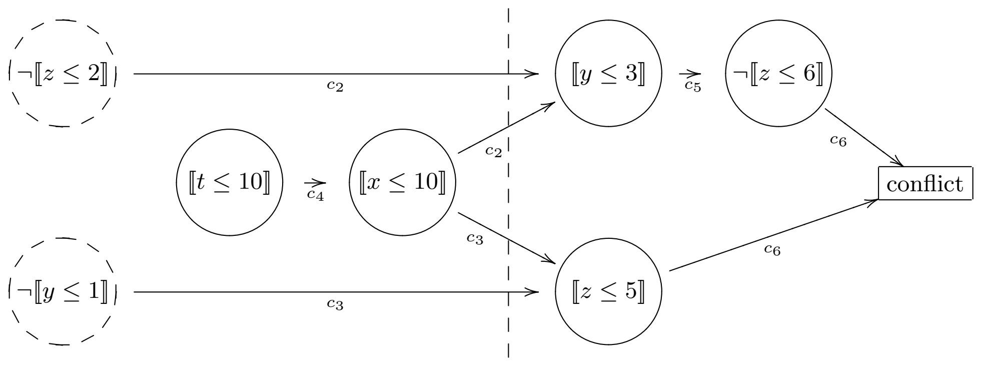

Fig. 1. Implication graph of propagation

to the conflict flow through that literal, and draws the cut just after this literal. The 1UIP cut for Figure 1 is shown as the dashed line. The resulting nogood is

$$\neg[\![z \leq 2]\!] \wedge \neg[\![y \leq 1]\!] \wedge [\![x \leq 10]\!] \rightarrow false$$

Since we are explaining conflicts completely analogously to a SAT (or SMT) solver we can attach activities to the Boolean variables representing the integer original variables. Each Boolean variable examined during the creation of the explanation (including those appearing in the final nogood) has their activity bumped. Every once in a while all activities counts are decreased, so that more recent activity counts for more. This allows us to implement activity based VSIDS search heuristic for the hybrid solver. We can also attach activity counters to clauses, which are bumped when they are involved in the explanation process.

Since all of the clauses generated are redundant information we can at any stage remove any of the generated clauses. This gives us the opportunity to control the size of the clausal representation of the problem. Just as in a SAT solver we can use clausal activities to decide which generated clauses are most worthwhile retaining.

3 Concluding Remarks

The simple description of lazy clause generation in the previous section does not lead to an efficient lazy clause generation solver, except for some simple kinds of examples. In practice we need to also lazily generate the Boolean variables required to represent the original integer (and set of integer) variables. We may also choose to either eagerly generate the explanation clauses as we execute forward propagation, or lazily generate explanations on demand during the process of explaining a conflict. For each propagator we have to determine how to efficiently determine explanations of each propagation, and which form the explanation should take. In particular for global constraints many choices arise. Lazy clause generation also seems to reduce the need for global constraints, since in some cases decomposition of the global constraint, together with conflict learning, seems to recapture the additional propagation that the global constraint has over its decomposition. Decompositions of global constraints may also be more incremental that the global, and learn more reusable nogoods. In short, lazy clause generation requires us to revisit much of the perceived wisdom for creating finite domain propagation solvers, and indeed leads to many open questions on the right design for a lazy clause generation solver. Experiments have shown that for some classes of problem, such as resource constrained project scheduling problems [6] and set constraint solving [7], lazy clause generation provides state-of-the-art solutions.

References

1. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of 38th Conference on Design Automation (DAC'01). (2001) 530–535
2. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints **14**(3) (2009) 357–391
3. Feydy, T., Stuckey, P.: Lazy clause generation reengineered. In Gent, I., ed.: Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. Volume 5732 of LNCS., Springer-Verlag (2009) 352–366
4. Niewenhuis, R., Oliveras, A., Tinelli, C.: Abstract DPLL and abstract DPLL modulo theories. In: Proceedings of the 11th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04). Volume 3452 of LNAI. (2004) 36–50
5. Marriott, K., Stuckey, P.: Programming with Constraints: an Introduction. MIT Press (1998)
6. Schutt, A., Feydy, T., Stuckey, P., Wallace, M.: Why cumulative decomposition is not as bad as it sounds. In Gent, I., ed.: Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. Volume 5732 of LNCS., Springer-Verlag (2009) 746–761
7. Gange, G., Stuckey, P., Lagoon, V.: Fast set bounds propagation using a BDD-SAT hybrid. Journal of Artificial Intelligence Research (2010) to appear