Lazy clause generation reengineered

Thibaut Feydy and Peter J. Stuckey

National ICT Australia, Victoria Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
{tfeydy,pjs}@csse.unimelb.edu.au

Abstract. Lazy clause generation is a powerful hybrid approach to combinatorial optimization that combines features from SAT solving and finite domain (FD) propagation. In lazy clause generation finite domain propagators are considered as clause generators that create a SAT description of their behaviour for a SAT solver. The ability of the SAT solver to explain and record failure and perform conflict directed backjumping are then applicable to FD problems. The original implementation of lazy clause generation was constructed as a cut down finite domain propagation engine inside a SAT solver. In this paper we show how to engineer a lazy clause generation solver by embedding a SAT solver inside an FD solver. The resulting solver is flexible, efficient and easy to use. We give experiments illustrating the effect of different design choices in engineering the solver.

1 Introduction

Lazy clause generation [1] is a hybrid of finite domain (FD) propagation solving and SAT solving that combines some of the strengths of both. Essentially in lazy clause generation, a FD propagator is treated as a clause generator, that feeds a SAT solver with a growing clausal description of the problem. The advantages of the hybrid are: it retains the concise modelling of the problem of an FD system, but it gains the SAT abilities to record nogoods and backjump, as well as use activities to drive search. The result is a powerful hybrid that is able to solve some problems much faster than either SAT or FD solvers.

The original lazy clause generation solver was implemented as a summer student project, where a limited finite domain propagation engine was constructed inside a SAT solver.

In this paper we discuss how we built a robust generic lazy clause generation solver in the G12 system. The crucial difference of the reengineered solver is that the SAT solver is treated as a propagator in an FD solver (hence reversing the treatment of which solver is master). This approach is far more flexible than the original design, more efficient, and available as a backend to the Zinc compiler.

We discuss the design decisions that go into building a robust lazy clause generation solver, and present experiments showing the effect of these decisions. The new lazy clause generation solver is a powerful solver with the following features:

– Powerful modelling: any Zinc (or MiniZinc) model executable by the G12 FD solver can be run using the lazy clause generation solver.
– Excellent default search: if no search strategy is specified then the default VSIDS search is usually very good.
– Programmed search with nogoods: on examples with substantial search the solver usually requires orders of magnitude less search than the FD solver using the same search strategy.
– Flexible global constraints: since decomposed globals are highly effective we can easily experiment with different decompositions.

The resulting system is a powerful combination of easy modelling and highly efficient search. It competes against the best FD solutions, often on much simpler models, and against translation to SAT. In many cases using the lazy clause generation solver with default settings to solve a simple FD statement of the problem gives very good results

2 Background

2.1 Propagation-based constraint solvers

Propagation-based constraint solving models constraints c as propagators, that map the set of possible values of variables (a domain) to a smaller domain by removing values that cannot take part in any solution. The key advantage of this approach is that propagation is "composable", propagators for each constraint can be constructed independently, and used in conjunction.

More formally. A *domain* D is a mapping from a fixed (finite) set of variables $\mathcal{V}$ to finite sets of integers. A *false domain* D is a domain with $D(x) = \emptyset$ for some $x \in \mathcal{V}$. A domain D_1 is *stronger* than a domain D_2, written $D_1 \sqsubseteq D_2$, if $D_1(x) \subseteq D_2(x)$ for all $x \in \mathcal{V}$. A range is a contiguous set of integers, we use *range* notation $[\,l \mathinner{..} u\,]$ to denote the range $\{d \in \mathcal{Z} \mid l \leqslant d \leqslant u\}$ when l and u are integers. We shall be interested in the notion of a *starting domain*, which we denote D_{init}. The starting domain gives the initial values possible for each variable. It allows us to restrict attention to domains D such that $D \sqsubseteq D_{init}$.

An *integer valuation* θ is a mapping of variables to integer values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. We extend the valuation θ to map expressions and constraints involving the variables in the natural way.

Let *vars* be the function that returns the set of variables appearing in a valuation. We define a valuation θ to be an element of a domain D, written $\theta \in D$, if $\theta(x_i) \in D(x_i)$ for all $x_i \in vars(\theta)$.

A *constraint* c over variables $x_1, \ldots, x_n$ is a set of valuations θ such that $vars(\theta) = \{x_1, \ldots, x_n\}$. We also define $vars(c) = \{x_1, \ldots, x_n\}$. We will *implement* a constraint c by a set of propagators that map domains to domains. A *propagator* f is a monotonically decreasing function from domains to domains: $f(D) \sqsubseteq D$, and $f(D_1) \sqsubseteq f(D_2)$ whenever $D_1 \sqsubseteq D_2$. A propagator f is *correct* for a constraint c iff for all domains D

$$\{\theta \mid \theta \in D\} \cap c = \{\theta \mid \theta \in f(D)\} \cap c$$

This is a very weak restriction, for example the identity propagator is correct for all constraints c.

Example 1. For the constraint $c \equiv x_0 \Leftrightarrow x_1 \leqslant x_2$ the function f defined by

$$
\begin{aligned}
f(D)(x_0) &= D(x_0) \cap (\{0 \mid \max D(x_1) > \min D(x_2)\} \cup \{1 \mid \min D(x_1) \leqslant \max D(x_2)\}) \\
f(D)(x_1) &= \begin{cases} D(x_1), & \{0,1\} \subseteq D(x_0) \\ \{d \in D(x_1) \mid d \leqslant \max D(x_2)\}, & D(x_0) = \{1\} \\ \{d \in D(x_1) \mid d > \min D(x_2)\}, & D(x_0) = \{0\} \end{cases} \\
f(D)(x_2) &= \begin{cases} D(x_2), & \{0,1\} \subseteq D(x_0) \\ \{d \in D(x_2) \mid d \geqslant \min D(x_1)\}, & D(x_0) = \{1\} \\ \{d \in D(x_1) \mid d < \max D(x_2)\}, & D(x_0) = \{0\} \end{cases}
\end{aligned}
$$

is a correct propagator for c. Let $D(x_0) = [\,0\,..\,1\,]$, $D(x_1) = [\,7\,..\,9\,]$, $D(x_2) = [\,-3\,..\,5\,]$ then $f(D)(x_0) = \{0\}$.

A *propagation solver* $solv(F, D)$ for a set of propagators F and a domain D finds the greatest mutual fixpoint of all the propagators $f \in F$.

In practice the propagation solver $solv(F, D)$ is carefully engineered to take into account which propagators must be at fixed point and do not need to be reconsidered. It also will include priorities on propagators so that cheap propagators are executed before expensive ones. See [2] for more details.

2.2 SAT solvers

Propagation based SAT solvers [3] are specialized propagation solvers with only Booleans variables, a built-in conflict based search and clausal constraints of the form $l_1 \vee l_2 \vee \cdots \vee l_n$ where l_i is a *literal* (a Boolean variable or its negation).

Unit propagation consists of detecting a conflict or fixing a literal once all other literals in a clause have been fixed to false. SAT solvers can perform unit propagation very efficiently using watch literals.

Conflict analysis is triggered each time a conflict is detected. By traversing a reverse implication graph (ie. remembering which clause fixed a literal), SAT solvers build a *nogood*, or conflict clause, which is added to the constraint store.

Conflict analysis allows SAT solvers to find the last satisfiable decision level, to which they can backjump, i.e. backtrack to a point before the last choicepoint.

SAT solvers maintain *activities* of the variables seen during conflict analysis. The heuristic used prioritizes variables that are the most involved in recent conflicts. This allow them to use a *conflict driven* or *activity based* search [3].

2.3 Original lazy clause generation

The original lazy clause generation hybrid solver [1] works as follows. Propagators are considered as clause generators for the SAT solver. Instead of applying propagator f to domain D to obtain $f(D)$, whenever $f(D) \neq D$ we build a clause that encodes the change in domains. In order to do so we must link the integer variables of the finite domain problem to a Boolean representation.

We represent an integer variable x with domain $D_{init}(x) = [\,l\,..\,u\,]$ using the Boolean variables $[\![x = l]\!], \ldots, [\![x = u]\!]$ and $[\![x \leqslant l]\!], \ldots, [\![x \leqslant u-1]\!]$. The variable $[\![x = d]\!]$ is true if x takes the value d, and false if x takes a value different from d. Similarly the variable $[\![x \leqslant d]\!]$ is true if x takes a value less than or equal to d and false if x takes a value greater than d. For integer variables with $D_{init}(x) = [\,0\,..\,1\,]$ we simply treat x as a Boolean variable.

Not every assignment of Boolean variables is consistent with the integer variable x, for example $\{[\![x = 5]\!], [\![x \leqslant 1]\!]\}$ requires that x is both 5 and $\leqslant 1$. In order to ensure that assignments represent a consistent set of possibilities for the integer variable x we add to the SAT solver clauses $DOM(x)$ that encode $[\![x \leqslant d]\!] \rightarrow [\![x \leqslant d+1]\!]$ and $[\![x = d]\!] \leftrightarrow ([\![x \leqslant d]\!] \wedge \neg[\![x \leqslant d-1]\!])$. We let $DOM = \cup\{DOM(v) \mid v \in \mathcal{V}\}$.

Any set of literals A on these Boolean variables can be converted to a domain: $domain(A)(x) = \{d \in D_{init}(x) \mid \forall [\![c]\!] \in A.vars(l) = \{x\} \Rightarrow x = d \models c\}$, that is the domain of all values for x that are consistent with all the Boolean variables related to x. Note that it may be a false domain.

Example 2. For example the assignment $A = \{[\![x_1 \leqslant 8]\!], \neg[\![x_1 \leqslant 2]\!], \neg[\![x_1 = 4]\!], \neg[\![x_1 = 5]\!], \neg[\![x_1 = 7]\!], [\![x_2 \leqslant 6]\!], \neg[\![x_2 \leqslant -1]\!], [\![x_3 \leqslant 4]\!], \neg[\![x_3 \leqslant -2]\!]\}$ is consistent with $x_1 = 3, x_1 = 6$ and $x_1 = 8$. hence $domain(A)(x_1) = \{3, 6, 8\}$. For the remaining variables $domain(A)(x_2) = [\,0\,..\,6\,]$ and $domain(A)(x_3) = [\,-1\,..\,4\,]$. $\square$

In the lazy clause generation solver, search is controlled by the SAT engine. After making a decision, unit propagation is performed to reach a unit propagation fixpoint with assignment A. Every fixed literal is then translated into a domain change, creating a new domain $D = domain(A)$, and the appropriate propagators are woken up. When we find a propagator f where $f(D) \neq D$ the propagator does not directly modify the domain D but instead generates a set of clauses C which explain the domain changes. Each clause is added to the SAT solver, starting a new round of unit propagation. This continues until fixpoint when the next SAT decision is made. See Figure 1(a). Adding an explanation of failure will force the SAT solver to fail and begin its process of nogood construction. It then backjumps to where the nogood would first propagate, and on untrailing the domain D must be reset back to its previous state.

Example 3. Suppose the SAT solver decides to set $[\![y \leqslant 1]\!]$ and unit propagation determines that $\neg[\![x_1 \leqslant 6]\!]$. Assuming the current domain $D(x_0) = [\,0\,..\,1\,]$, $D(x_1) = [\,1\,..\,9\,]$, $D(x_2) = [\,-3\,..\,5\,]$ then the domain changes to $D'(x_1) = [\,7\,..\,9\,]$ and propagators dependent on the lower bound of x_1 are scheduled, including for example the propagator f for $x_0 \Leftrightarrow x_1 \leqslant x_2$ from Example 1. When applied to domain D' it obtains $f(D')(x_0) = \{0\}$. The clausal explanation of the change in domain of x_1 is $\neg[\![x_1 \leqslant 6]\!] \wedge [\![x_2 \leqslant 5]\!] \rightarrow \neg x_0$. This becomes the clause $[\![x_1 \leqslant 6]\!] \vee \neg[\![x_2 \leqslant 5]\!] \vee \neg x_0$. This is added to the SAT solver. Unit propagation sets the literal $\neg x_0$. This creates domain $D''(x_0) = \{0\}$ which causes the propagator f to be re-examined but no further propagation occurs.

Assuming $domain(A) \sqsubseteq D$, then when clauses C that explain the propagation of f are added to the SAT database containing DOM and unit propagation is

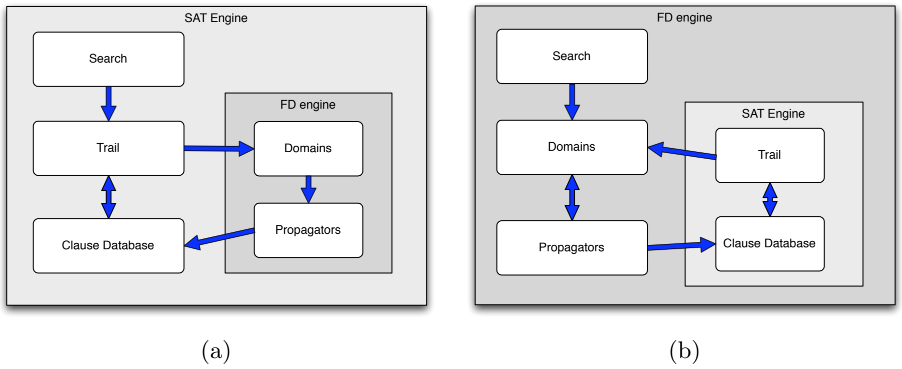

Fig. 1. (a) The original architecture for lazy clause generation, and (b) the new architecture.

performed, then the resulting assignment A' will be such that $domain(A') \sqsubseteq f(D)$. Using lazy clause generation we can show that the SAT solver maintains an assignment which is at least as strong the domains of an FD solver [1].

The advantages over a normal FD solver are that we automatically have the nogood recording and backjumping ability of the SAT solver applied to our FD problem, as well as its activity based search.

3 Lazy Clause Generation as a Finite Domain solver

The original lazy clause generation solver used a SAT solver as a master solver and had a cut down finite domain propagation engine inside. This approach meant that the search was not programmable, but built into the SAT solver and minimization was available only as dichotomic search, on top of SAT search.

3.1 The new solver architecture

The new lazy clause generation solver is designed as an extension of the existing G12 finite domain solver. It is a backend for the Zinc compiler which can be used wherever the finite domain solver is used. The new solver architecture is illustrated in Figure 1(b).

Search is controlled by the finite domain solver. When a variables domain changes propagators are woken as usual, and placed in priority queue. The SAT solver unit propagation engine, acts as a global propagator. Whenever a literal is set or clauses are posted to the SAT solver, then this propagator is scheduled for execution at the highest priority.

When a propagator f is executed that updates a variable domain ($f(D) \neq D$) or causes failure ($f(D)$ is a false domain) it posts an *explanation* clause to the SAT solver that explains the domain reduction or failure. This will schedule the SAT propagator for execution.

The SAT propagator when executed computes a unit fixpoint. Then each of the literals fixed by unit propagation causes the corresponding domain changes to be made in the domain, which may wake up other propagators. The cycle of propagation continues until a fixpoint is reached. Note that other work (e.g. [4]) has suggested using a SAT solver to implement a global propagator inside a CP solver.

3.2 Encoding of finite domain variables

In the new architecture integer variables are implemented as usual with a representation of bounds, and domains with holes, and queues of events for bounds changes, fixing a variable and removing an interior value. Concrete variables are restricted to be *zero based*, that is have initial domains that range over values $[\,0\,..\,n\,]$, views [5] are used to encode non-zero based integer variables.

The lazy clause generation solver associates each integer variable with a set of Boolean variables. Changes in these Boolean variables will be reflected in the domains of the integer variables. There are two possible ways of doing this:

The array encoding The array encoding of integer variables is an encoding with two arrays :

- An array of inequality literals $[\![x \leqslant d]\!]$, $d \in [\,0\,..\,n-1\,]$
- An array of equality literals $[\![x = d]\!]$, $d \in [\,0\,..\,n\,]$

inequality literals are generated eagerly whereas equality literals are generated lazily. When a literal $[\![x = d]\!]$ has to be generated we post the *domain clauses*: $[\![x = d]\!] \rightarrow [\![x \leqslant d]\!]$, $[\![x = d]\!] \rightarrow \neg[\![x \leqslant d-1]\!]$, and $\neg[\![x \leqslant d-1]\!] \wedge [\![x \leqslant d]\!] \rightarrow [\![x = d]\!]$. The array encoding is linear in the size of the initial integer domain, while bound updates are linear in the size of the domain reduction.

The list encoding The list encoding generates inequality and equality literals lazily when they are required for propagation or explanation. As such the size of the encoding is linear in the number of generated literals, and a bound update is linear in the number of generated literals that will be fixed by the update.

When a literal $[\![x \leqslant d]\!]$ has to be generated :

- we determine the closest existing bounds: $l = \max\{d' \mid [\![x \leqslant d']\!] \text{ exists}, d' < d\}$, $u = \min\{d' \mid [\![x \leqslant d']\!] \text{ exists}, d < d'\}$
- we post the new *domain clauses* : $[\![x \leqslant l]\!] \rightarrow [\![x \leqslant d]\!], [\![x \leqslant d]\!] \rightarrow [\![x \leqslant u]\!]$

When a literal $[\![x = d]\!]$ has to be generated we first generate the literals $[\![x \leqslant d]\!]$ and $[\![x \leqslant d-1]\!]$ if required, then proceed as for the array encoding.

Caching the positions of the largest $[\![x \leqslant d]\!]$ which is *false* (lower bound $d+1$) and smallest $[\![x \leqslant d]\!]$ which is *true* (upper bound d) allows access performances similar to the array encoding for the following reasons. An inequality literal is required either to explain another variable update or to reduce the domain of the current variable. In the first case, the inequality literal is most likely the one corresponding to the current bound, in which case it is cached. In the latter case, where we reduce a variable bound by an amount δ, a sequence of δ clauses will have to be propagated in the array encoding.

When a new literal is generated, previous sequence literals becomes redundant. When a literal $[\![x \leqslant d]\!]$ is inserted where $l < d < u$ then the binary clause $[\![x \leqslant l]\!] \rightarrow [\![x \leqslant u]\!]$ becomes redundant. At most n redundant constraints exists after n literals have been generated. However these redundant constraints alter the propagation order and can have a negative impact on the nogoods generated during conflict analysis.

By default the solver uses array encoding for variables with "small" (< 10000) initial domains and list encoding for larger domains.

Use of views We use views [5] to avoid creating additional variables. A *view* is a monotonic, injective function from a variable to a domain. In practice, we use affine views, and each variable is an affine view over a zero-based variable.

Given a variable x where $D_{init}(x) = [\, l \,..\, u \,]$ where $l \neq 0$ we represent x as a view $x = x_c + l$ where $D_{init}(x_c) = [\, 0 \,..\, u - l \,]$. Given a variable x and a new variable y defined as $y = ax + b$, let x_c be the concrete variable of x, ie. $\exists a', \exists b'$ such that $x = a'x_c + b'$ then y is defined as $y = a'ax_c + (a'b + b')$.

Using views rather than creating fresh variables has the following advantages over creating new concrete variables :

– *space savings.* This is especially true with the array encoding which is always linear in the domain size.
– *literal reuse.* Reusing literals means stronger and shorter nogoods.

3.3 Propagator implementation

For use in a lazy clause generation each propagator should be extended to explain its propagations and failures. Note that the new lazy clause generation system can handle propagators that do not explain themselves by treating their propagations as decisions, but this significantly weakens the benefits of lazy clause generation.

When a propagator is run, then all its propagations are reflected by changing the domains of integer variables, as well as adding *explanation clauses* to the SAT solver that explains the propagation made. Note that it must also explain failure.

Once we are using lazy clause generation we need to reassess the best possible way to implement each constraint.

Linear constraints Linear constraints $\sum_{i \in 1 \ldots n} a_i x_i \leqslant a_0$ and $\sum_{i \in 1 \ldots n} a_i x_i = a_0$ are among the most common constraints. But long linear constraints do not generate very reusable explanations, since they may involve many variables. It is worth considering breaking up a long linear constraint into smaller units. So e.g. $\sum_{i \in 1 \ldots n} a_i x_i = a_0$ becomes $s_1 = a_1 x_1 + a_2 x_2$, ... $s_{i+1} = s_i + a_{i+1} x_{i+1}$, ..., $s_n = s_{n-1} + a_n x_n$, $a_0 = s_n$. Note that for finite domain propagation alone this is guaranteed to result in the same domains (on the original variables) [6]. The decomposition adds many new variables and slows propagation considerably, but means explanations are more likely to be reused. We shall see that the *size* of the intermediate sums s_i will be crucial in determining the worth of this translation.

Reified constraints The lazy clause generation solver does not have regular reified constraints but only implications constraints of the form $l \Rightarrow c$ where l is a literal and c is a constraint. This has the advantage that the events of this implication constraint are the events of the non reified version plus an event on l being asserted. Similarly the explanations for $l \Rightarrow c$ are the same as for c with $\neg l$ disjoined. The main advantage is that often we do not need both directions.

Example 4. Consider the constraint $x_1 + 2 \leqslant x_2 \vee x_2 + 5 \leqslant x_1$. The usual decomposition is $b_1 \Leftrightarrow x_1 + 2 \leqslant x_2$, $b_2 \Leftrightarrow x_2 + 5 \leqslant x_1$, $(b_1 \vee b_2)$. A better decomposition is $b_1 \Rightarrow x_1 + 2 \leqslant x_2$, $b_2 \Rightarrow x_2 + 5 \leqslant x_1$, $(b_1 \vee b_2)$ since the only propagation possible if the falsity of one of the inequalities forcing a Boolean to be false, which forces the other Boolean to be true and the other inequality to hold. Note that e.g. $x_0 \Leftrightarrow x_1 \leqslant x_2$ is implemented as $x_0 \Rightarrow x_1 \leqslant x_2$, $\neg x_0 \Rightarrow x_1 > x_2$ illustrating the need for the lhs to be a literal rather than a variable.

3.4 Global propagators

Rather than create complex explanations for global constraints it is usually easier to build decompositions. Learning for decomposed globals is stronger, and can regain the benefits of the global view that are lost by decomposition. If we are using decomposition to define global propagators then we can easily experiment with different definitions. It is certainly worth reconsidering which decomposition to use in particular for lazy clause generation.

Element constraints An element constraint `element`(x, a, y) which enforces that $y = a[x]$ where a is a fixed array of integers indexed on the range $[\,0\,..\,n\,]$ can be implemented simply as the binary clauses $\wedge_{k=0}^{n} [\![x = k]\!] \rightarrow [\![y = a[k]]\!]$ which enforces domain consistency.

GCC We propose a new decomposition of the global cardinality constraint (and by specialisation also the alldifferent constraint) which exploits the property of our solver that maintaining the state of the literals $[\![x = k]\!]$ and $[\![x \leqslant k]\!]$ is cheap as it is part of the integer variable encoding. `gcc`$([x_1, \ldots, x_n], [c_1, \ldots, c_m])$ enforces that the value i occurs c_i times in $x_1, \ldots, x_n$. We introduce $m + 1$ sum variables $s_0, \ldots, s_m$ defined by $s_i = \sum_{j \in 1 \ldots n} [\![x_j \leqslant i]\!]$ and post the following constraints $s_m - s_0 = \sum_{i \in 1..m} c_i$ and $\forall i \in 1 \ldots m$, $s_i - s_{i-1} = c_i$. To generate holes in the domains we add the constraints $\forall i \in 1 \ldots m, c_i = \sum_{j \in 1 \ldots n} [\![x_j = i]\!]$.

3.5 Extending the SAT solver

SAT solvers need to be slightly extended to be usable with lazy clause generation.[1] The first extension is to communicate domain information back to the

[1] Although we manage this by building code outside the SAT solver code, leaving it untouched, but accessing its data structures.

propagation solver, e.g. when $[\![x \leqslant d]\!]$ is set true we remove from $D(x)$ the values greater than d, when is set false we remove values less than or equal to d, similarly for $[\![x = d]\!]$.

Lazy clause generation adds new clauses as search progresses of three kinds: domain clauses, explanation clauses, and nogood clauses. Usually a SAT solver only posts nogood clauses. On posting a nogood it immediately backjumps to the first place the nogood clause could unit propagate. We don't have such a luxury in lazy clause generation, since the SAT solver is not in charge of search, and indeed it may be unaware of choices that did not affect any of its variables.

When the SAT solver can backjump a great distance because a failure is found to not depend on the last choice, we have to mimic this. This is managed by checking the SAT solver first in each propagation loop, before applying any search decision. If unit propagation in SAT still detects failure, then we can immediately fail, and continue backtracking upward to the first satisfiable ancestor state.

A feature of the dual modelling inherent in lazy clause generation is that explanation clauses are redundant information, since they can be regenerated by the propagators whenever they could unit propagate.[2] Hence we can choose to delete these clauses from the SAT solver on backtracking. This reduces the number of clauses in the database, but means that more expensive propagators need to be called more often. We can select whether to delete explanations or not, by default they are deleted.

3.6 Search

Search is controlled by the FD solver, but we can make use of information from the SAT solver. We can perform:

VSIDS search The SAT solver search heuristic VSIDS [3], based on activity, can be used to drive the search. At each choice point we retrieve the highest activity literal from the SAT solver and try setting it true or false. This is the default search for the lazy solver. Because of lazy encodings, it may be necessary to interleave search with the generation of new literals for unfixed variables, as not all literals encoding the variable domain exist initially, and in the end we need to fix all the finite domain variables. As in SAT solvers, we restart the search from time to time.

Finite Domain Search One of the main advantages of the solver presented here compared to the solver presented in [1] is the ability to use programmed specialized finite domain searches if they are specified in the model.

Branch and bound Search We use incremental branch and bound rather than dichotomic branch and bounds with restart due to the incrementality of our SAT solver. This differs with other SAT solver based approach such as [7] and [1].

[2] Except in cases where that the clause is stronger than the propagator. (See [1]).

Hybrid Search We can of course build new hybrids of finite domain programmed search that make use of the activity values from the SAT solver as part of the search. We give an example in Section 4.3

4 Experiments

The experiments were run on Core 2 T8300 (2.40 GHz), except the experiments from 4.4 and 4.6 which were run respectively on a Pentium D 3.0 GHz and a Xeon 3.0 GHz for comparison with cited experiments. All experiments were run on one core. We use the following scheme for expressing variants of our approach: l = G12 lazy clause generation solver, f = G12 normal finite domain solver; v = VSIDS search, s = problem specific programmed search, h = hybrid search (see Section 4.3). When we turn off optimizations we place them after a minus: d = no deletion, a = list encoding (no arrays), r = normal reified constraints rather than single implication ones from Section 3.3, and w = no views.

4.1 Arithmetic Puzzles

The Grocery Puzzle [8] is a tiny problem but its intermediate variables have bounds up to 2^{38}. It cannot be solved using the array encoding. SEND-MORE-MONEY is another trivial problem, but here if we break the linear constraint (which has coefficients up to 9000) into ternary constraints the array encoding requires a second to solve because of the size of intermediate sum variables. Applying the list encoding on the decomposed problem, and either encoding on the original form require only a few milliseconds. These simple examples illustrate why the lazy list encoding is necessary for a lazy clause generation solver.

4.2 Constrained path covering problem

The constrained path covering problem is a problem which arises in transportation planning and consists of finding a covering of minimum cardinality of a directed network. Each node $n \in Nodes$ except the start and end nodes have a positive cost $cost[n]$, and the total cost of a path cannot exceed a fixed bound. A CP model for this problem associates predecessor ($prev[n]$)/successor($next[n]$) variables to each node, as well as a cumulative cost $cumul[n]$, related by the following constraints :

$$\forall n \in Nodes.cumul[n] - cost[n] = cumul[prev[n]]$$
$$\forall n \in Nodes.\forall p \in Nodes.prev[n] = p \Leftrightarrow next[p] = n$$

In Table 1, we compare the lazy clause generation solver with default search (lv) and a specialized finite domain search (ls), as well as the G12 FD solver (fs) with the same search. We also compare creating fresh variables (lv-w,ls-w) for the result of the element constraints generated above ($cumul[prev[n]]$), as opposed to using views. The benchmark CPCP-n-m has n nodes and m edges.

	Times(sec)					Choicepoints				
	lv	ls	lv-w	lv-w	fs	lv	ls	lv-w	lv-w	fs
CPCP-17-89	0.40	0.17	0.76	0.27	**0.08**	572	**63**	563	**63**	1905
CPCP-23-181	9.02	**0.25**	36.74	0.38	0.28	31521	**449**	44423	1198	9149
CPCP-30-321	>600	**0.53**	>600	0.80	0.64	?	**804**	?	1595	9666
CPCP-37-261	>600	**1.59**	>600	2.70	>600	?	**1689**	?	2067	?
CPCP-37-495	>600	**0.99**	>600	1.44	>600	?	**1745**	?	3348	?
Average	>361.89	**0.71**	367.5	1.12	>240.2	?	**950**	1654	1231	?

Table 1. Constrained Path Covering Problem

The specialized finite domain search clearly outperforms VSIDS on these problems. Avoiding creating variables by using views improves search as well. This is especially true with VSIDS which can be explained by the addition of useless literals, which just confuse its discovery of the "hard parts" of the problem.

The lazy clause generation solver, while slower than the finite domain solver, scales a lot better due to huge search reductions and wins for all but the easiest instance.

4.3 Radiation

Radiation scheduling [9] builds a plan for delivering a specific pattern of radiation by multiple exposures. The best search for this problem first fixes the variables shared by subproblems then fixes the subproblem variables, for each subproblem independently. Then if any subproblem is unsatisfiable we can use cuts to backtrack directly to search again the shared variables. For these experiments since we are restricted to Zinc search which does not support cuts, we simply search first on the shared variables and then on the subproblem variables in turn.

We use square matrices of size 6 to 8 with maximum intensity ranging from 8 to 10 constructed as in [9]. We ran these instances using VSIDS (lv), as well as the specialized finite domain search (without cuts) (ls), as well as a hybrid search (lh) where we use the specialized search on the shared variables, and then VSIDS on the remaining variables. We also run the FD solver (fs) with specialized search.

Each instance was run with the original linear inequalities, as well as with a decomposition into ternary inequalities, introducing intermediate sums. These linear sums are short ($6-8$ Boolean variables) and have small coefficients ($1-10$).

In this case, introducing intermediate sums definitely improved nogood generation as the choice point count is systematically reduced. On average, the specialized search outperforms VSIDS search, although the difference is reduced by the constraint decomposition, which strengthens the reusability of the explanations and nogoods generated. The hybrid search outperforms both the finite domain search and VSIDS on most instances. The FD solver is not competitive on any but the smallest instances because of the lack of explanation.

	Time(sec)							Choicepoints(x1000)						
	Long linear				Ternary			Long linear				Ternary		
	fs	lv	ls	lh	lv	ls	lh	fs	lv	ls	lh	lv	ls	lh
6-08-1	2.25	**0.40**	0.61	**0.40**	0.60	0.78	0.58	72.8	1.46	1.40	**1.27**	1.29	1.33	1.31
6-08-2	**0.16**	0.37	0.53	0.41	0.54	0.72	0.53	**0.90**	1.40	1.07	2.00	1.35	1.06	1.03
6-08-3	1.12	**0.36**	0.80	0.48	0.61	1.06	0.65	48.9	1.60	2.25	1.59	**1.44**	2.09	1.56
6-09-1	2.11	0.35	0.39	**0.26**	0.52	0.46	0.36	39.2	1.19	0.61	0.46	1.43	0.52	**0.45**
6-09-2	6.68	**0.70**	1.46	0.74	0.86	1.66	1.00	271.7	3.02	4.56	2.99	**2.32**	3.98	2.78
6-09-3	432.4	0.84	1.62	**0.77**	0.96	1.86	1.06	10497	2.96	5.01	2.98	**2.66**	4.68	2.79
7-08-1	>1800	0.69	1.18	**0.56**	0.82	1.42	0.88	?	2.19	2.47	**1.11**	1.32	2.48	1.12
7-08-2	1378	**0.42**	0.89	0.54	0.76	1.09	0.78	60310	**0.78**	1.49	0.90	1.31	1.46	0.89
7-08-3	299.2	1.00	1.67	**0.82**	1.44	1.89	1.18	13767	4.49	3.95	2.26	3.49	3.57	**2.19**
7-09-1	>1800	1.17	1.63	**0.79**	1.49	2.05	1.20	?	3.48	3.59	**1.71**	3.02	3.53	1.79
7-09-2	>1800	4.05	8.62	**3.10**	3.82	9.14	4.08	?	12.4	27.9	10.7	**7.88**	23.5	9.30
7-09-3	5.60	1.11	2.04	**1.00**	1.44	2.30	1.40	199.2	4.27	4.69	**2.90**	3.47	4.22	2.58
8-09-1	950.3	3.42	4.36	**1.99**	2.94	5.11	3.14	27814	8.29	7.52	**3.99**	4.39	7.31	3.92
8-09-2	14.8	2.01	2.58	**1.50**	1.86	3.21	2.26	424.4	5.24	3.72	3.10	**2.72**	3.49	3.21
8-09-3	31.70	5.94	7.39	**3.02**	7.02	7.56	4.30	1345	14.9	18.3	10.9	**10.7**	15.3	8.23
8-10-1	1033	45.72	34.50	**18.76**	35.28	30.07	24.78	39494	51.7	64.0	41.5	**36.1**	40.1	38.1
8-10-2	>1800	26.16	21.47	**8.49**	11.41	20.74	12.47	?	39.4	47.1	18.9	**15.7**	33.0	18.6
8-10-3	>1800	93.68	37.11	**17.80**	54.41	31.11	20.62	?	88.1	96.8	52.8	55.8	63.5	**41.3**
Av.	>706	10.47	7.16	**3.41**	7.04	6.79	4.51	?	13.7	16.5	9.0	8.7	12.0	**7.8**

Fig. 2. Radiation problem : time and choice points

4.4 Open shop scheduling problem

An open shop scheduling problem n-m-k is defined by n jobs and m machines, where each job consist of m tasks each requiring a different machine. The objective is to find a minimal schedule such that each pair of tasks (i, j) from the same job or machine are not overlapping, which is represented by the constraint $s_i + d_i \leqslant s_j \vee s_j + d_j \leqslant s_i$, where s_i and s_j are the start time of the tasks and d_i and d_j are the (fixed) durations of the tasks. An open-shop problem of size $n \times m$ has nm variables and $(nm)(nm + 1)/2$ non-overlapping constraints.

The benchmarks used are from [7]. In Table 2(a) we compare: our default lazy clause generation solver (lv) and without deletion (lv-d); the solver presented in [1] (cutsat); and the static translation approach of [7] (csp2sat) using MiniSAT version 2.0. All solvers use VSIDS search. We do not compare against f for this and subsequent problems since they all use VSIDS search. The table shows that our approach, using branch and bound rather than dichotomic search and with a slightly different propagation, vastly outperforms cutsat which beats cps2sat. Deleting previously generated explanations also substantially improves the results.

In Table 2(b) we compare results on smaller instances for different variations of lv. We can see that the overhead of the list representation is substantial, while the use of one directional reification also has significant benefits, although this is lessened by deletion.

4.5 Hoist scheduling

We tested our lazy clause generation solver on the hoist scheduling problem presented in [10]. Example j-h-p has j jobs, h hoists and parallel tracks if $p = y$. We compare a simple Zinc model, run with default settings (lv) and without

tai	lv-d	lv	cutsat	csp2sat
20-20-1	55.27	**31.69**	283.1	1380.3
20-20-2	341.6	**47.54**	497.8	1520.1
20-20-3	56.63	**37.80**	270.7	1367.6
20-20-4	93.47	**38.14**	269.9	1361.3
20-20-5	50.74	**47.94**	278.8	1397.0
20-20-6	57.62	**35.26**	324.2	1405.6
20-20-7	79.20	**38.44**	455.3	1439.9
20-20-8	130.40	**41.42**	424.8	1420.8
20-20-9	44.54	**32.41**	246.1	1377.8
20-20-10	49.27	**38.84**	242.2	1346.8
Average	95.88	**39.96**	329.8	1401.7

(a)

tai	lv-d	lv	lv-ad	lfd-a	lv-dr	lv-r
15-15-1	18.63	11.73	37.47	45.28	22.94	**11.24**
15-15-2	12.68	**11.68**	70.77	76.50	24.80	14.11
15-15-3	18.87	**13.44**	49.78	96.58	15.27	15.10
15-15-4	17.61	**8.47**	55.06	54.12	14.83	9.21
15-15-5	21.02	12.34	77.30	74.45	35.36	**12.16**
15-15-6	32.44	**12.75**	26.12	39.85	20.67	16.97
15-15-7	22.66	**15.80**	33.25	45.25	23.27	16.54
15-15-8	17.12	**13.55**	29.84	21.54	12.97	13.87
15-15-9	29.68	23.23	99.44	71.24	24.51	**17.11**
15-15-10	15.1	**10.85**	124.21	41.79	47.61	17.32
Average	20.58	**13.38**	60.32	56.66	24.22	14.36

(b)

Table 2. Open shop scheduling: (a) comparing with previous approaches on hard instances, and (b) comparing the two variable representations on easier instances.

Example	fzntini	lv-d	lv	ic	iclin
4-1-n	1153.3	3.46	2.81	**2.18**	8.57
4-2-n	458.4	0.72	0.66	0.3	**0.1**
4-2-y	358.5	0.35	0.34	**0.3**	2.04
4-3-n	493.5	0.55	0.55	0.39	**0.09**
4-3-y	272.6	**0.32**	0.33	0.4	1.0
5-1-n	>1800	3.98	**3.68**	10.8	52.9
5-2-n	1090.2	0.40	0.77	0.5	**0.1**
5-2-y	594.0	**0.53**	0.57	8.55	50.2
5-3-n	983.9	0.40	0.42	0.5	**0.12**
5-3-y	484.3	0.44	0.72	0.7	**0.12**

Example	fzntini	lv-d	lv	ic	iclin
6-1-n	>1800	1.23	**1.18**	4.09	13.9
6-2-n	>1800	1.80	1.78	0.7	**0.15**
6-2-y	827.7	**1.01**	2.31	4.81	28.8
6-3-n	1524.8	0.63	0.68	0.6	**0.14**
6-3-y	780.0	0.86	0.72	**0.56**	4.43
7-1-n	>1800	**1.39**	1.46	3.65	9.59
7-2-n	>1800	6.02	4.82	0.70	**0.18**
7-2-y	927.9	19.5	18.0	**17.2**	100.1
7-3-n	>1800	0.53	0.78	0.80	**0.17**
7-3-y	912.8	**0.66**	0.92	0.80	4.71
Average	>1083	2.26	**2.17**	2.93	13.95

Table 3. Hoist scheduling results

deletion (lv-d) as well as by static translation to SAT using [11] (fzntini), all of these using VSIDS search, against the carefully crafted Eclipse model [10] using its specialized finite domain search, run either with the Eclipse finite domain solver ic or with a finite domain and linear programing hybrid iclin using COIN-OR [12] as the linear solver. The results in Table 3 shows that our approach using a simple model is competitive with the specialized models with hand-written search, especially on the hardest instances. We see that lazy clause generation is competitive whereas static translation (fzntini) struggles because of the size of the resulting SAT model (in results not shown). Clause deletion does not seem as advantageous as in the open-shop benchmarks.

4.6 Quasi-Group Completion

A $n \times n$ *latin square* is a square of values $x_{ij}, 1 \leqslant i, j \leqslant n$ where each number $[1..n]$ appears exactly once in each row and column. It is represented by

	Time (seconds)				Choicepoints			
	gcc	bnd	bnd+	diseq	gcc	bnd	bnd+	diseq
qcp-25-264-0-ext	**6.44**	1164.34	166.19	31.48	1759	15171	**1635**	77110
qcp-25-264-1-ext	**44.80**	>1800	1577.96	>1800	**13773**	?	17099	?
qcp-25-264-2-ext	**2.53**	730.13	116.25	68.74	**421**	10016	971	153128
qcp-25-264-3-ext	**157.58**	>1800	>1800	1473.21	**46063**	?	?	702897
qcp-25-264-4-ext	**22.30**	1334.34	712.71	>1800	**6697**	17322	7648	?
qcp-25-264-5-ext	**12.58**	1449.90	459.54	537.30	**3785**	18254	4679	380392
qcp-25-264-6-ext	**341.62**	>1800	>1800	170.99	**83871**	?	?	216433
qcp-25-264-7-ext	**6.08**	1265.34	159.93	178.15	1423	14289	**1342**	200123
qcp-25-264-8-ext	**3.01**	546.69	75.73	23.08	**553**	6051	586	76995
qcp-25-264-9-ext	**12.66**	>1800	638.59	36.18	**3303**	?	5484	94814
qcp-25-264-10-ext	**5.30**	914.16	121.65	123.26	981	11128	**979**	200110
qcp-25-264-11-ext	0.81	15.76	14.46	**0.35**	**0**	**0**	**0**	399
qcp-25-264-12-ext	0.80	37.53	14.65	**0.55**	**0**	590	**0**	3960
qcp-25-264-13-ext	**0.80**	337.77	14.62	1.03	**0**	4259	**0**	11408
qcp-25-264-14-ext	**4.77**	1106.12	146.86	347.84	**1183**	13412	1251	323175
Average	**41.47**	>1047.35	>522.13	>439.48	**10920.8**	?	?	?

Table 4. Comparison of all different decomposition on quasi group completion problems

constraints

$$\texttt{alldifferent}([x_{i1}, \ldots, x_{in}]),\ 1 \leqslant i \leqslant n$$
$$\texttt{alldifferent}([x_{1j}, \ldots, x_{nj}]),\ 1 \leqslant j \leqslant n$$

The quasigroup completion problem (QCP) is a latin square problem where some of the x_{ij} are given. These are challenging problems which exhibit phase transition behaviour. We use instances from the 2008 CSP Solver Competition [13].

We compare several decompositions of the `alldifferent` constraint all using our default solver lv. The diseq decomposition is the usual decomposition into disequalities $\neq$. The gcc decomposition explained in Section 3.4, strengthens propagation by doing some additional bounds propagation. The bnd decomposition is a decomposition that maintains bounds-consistency [14], while bnd+ is a modification of bnd where we replace each expression $[\![x_i \leqslant d]\!] \wedge \neg[\![x_i \leqslant d-1]\!]$ with $[\![x_i = d]\!]$ to obtain a decomposition which combines the propagation of bnd and diseq. The different variations only require changing the definition of `alldifferent` included in the Zinc model.

The results are shown in Table 4. While the bnd+ decomposition is the strongest its size is prohibitive. The gcc decomposition is comprehensively best hitting the right tradeoff of strength of propagation versus size of decomposition. Comparing with results from the CSP Solver competition 2008, only two solvers could solve more than 2 of these problems (using the diseq model) in 1800s (on a 3GHz Xeon): choco2_dwdeg, requiring an average $>$ 608.8s (2 timeouts), and choco2_impwdeg, requiring $>$ 776.8s (3 timeouts)

5 Conclusion

The reengineered lazy clause generation solver is highly flexible hybrid constraint programming solver that combines the modelling and search flexibility of finite domain solving with the learning and adaptive search capabilities of SAT solvers. It forces us to reconsider many design choices for finite domain propagation. The resulting solver is highly competitive and able to tackle problems that are beyond the scope of either finite domain or SAT solvers alone. It also illustrates that the combination of specialized finite domain search with nogoods can be extremely powerful.

Acknowledgments: NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

References

1. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation = lazy clause generation. In: Procs. of CP2007. (2007) 544–558
2. Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. ACM Trans. Program. Lang. Syst. **31**(1) (2008)
3. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Procs. DAC 2001. (2001) 530–535
4. Bacchus, F.: GAC via unit propagation. In: Procs. of CP2007. (2007) 133–147
5. Schulte, C., Tack, G.: Views and iterators for generic constraint implementations. In: Procs. of CP 2005. (2005) 817–821
6. Harvey, W., Stuckey, P.: Improving linear constraint propagation by changing constraint representation. Constraints **8**(2) (2003) 173–207
7. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. In: Proceedings CP2006. (2006) 590–603
8. Schulte, C., Smolka, G.: Finite Domain Constraint Programming in Oz. A Tutorial http://www.mozart-oz.org/documentation/fdt/.
9. Baatar, D., Boland, N., Brand, S., Stuckey, P.J.: Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches. In: Procs. of CPAIOR 2007. (2007) 1–15
10. Rodosek, R., Wallace, M.: A generic model and hybrid algorithm for hoist scheduling problems. In: Proceedings CP1998. (1998) 385–399
11. Huang, J.: Universal booleanization of constraint models. In: Procs. of CP2008. (2008) 144–158
12. Lougee-Heimer, R.: The Common Optimization INterface for operations research: Promoting open-source software in the operations research community. IBM Journal of Research and Development **47**(1) (2003) 57–66
13. : International CSP Solver Competition http://www.cril.univ-artois.fr/CPAI08/.
14. Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C.G., Walsh, T.: Decompositions of all different, global cardianlity and related constraints. In: IJCAI. (2009)