Modelling for Lazy Clause Generation

Olga Ohrimenko and Peter J. Stuckey
NICTA Victoria Research Lab, Department of Comp. Sci. and Soft. Eng.
University of Melbourne, Australia
{olgao,pjs}@csse.unimelb.edu.au

Abstract

Lazy clause generation is a hybrid SAT and finite domain propagation solver that tries to combine the advantages of both: succinct modelling using finite domains and powerful nogoods and backjumping search using SAT technology. It has been shown that it can solve hard scheduling problems significantly faster than SAT or standard finite domain propagation alone. This new hybrid opens up many choices in modelling problems because of its dual representation of problems as both finite domain and SAT variables. In this paper we investigate some of those choices. Arising out of the modelling choices comes a novel combination of bounds representation and domain propagation which creates a form of propagation of disjunctions. We show this novel modelling approach can outperform more standard approaches on some problems.

1 Introduction

We consider the problem of solving *Constraint Satisfaction Problems* (CSPs) defined in the sense of [7], which can be stated briefly as follows:

> We are given a set of variables, a domain of possible values for each variable, and a set (read as a conjunction) of constraints. Each constraint is a relation defined over a subset of the variables, limiting the combination of values that the variables in this subset can take. The goal is to find a *consistent* assignment of values to the variables so that all the constraints are satisfied simultaneously.

Finite domain propagation systems solve CSPs using elaborate search strategies working in tandem with propagation to reduce the search space by removing inconsistent assigments as early as possible. There has been a significant amount of research on how to solve CSPs by encoding them in a Boolean clausal representation and then using Boolean satisfiability (SAT) solver to find a solution. Although this approach is quite successful for some problem classes, on other problems it turns out that the brute-force translation of the problem is too big to be handled effectively.

Finite domain propagation solvers effectively represent the possible values of variables by a set of choices which can be naturally modelled as Boolean variables. Recently [11] we described 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. We showed 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 [10] solvers with difference logic. The resulting hybrid [11] 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.

In this paper we extend our previous work by exploiting the possibilities that the new system offers.

We show that this approach allows independence between the Boolean representation of integer variables and the propagators that act upon them. This representation independence leads to a new type of propagation: mixing bounds representation and domain propagators. The new propagator results in disjunctive propagation, where new information is created by propagation which is disjunctive in nature, even though the propagator was not a disjunctive at the start. Since the underlying SAT representation of propagation can represent disjunctive information efficiently, it allows us to create new "disjunctive propagators" from scratch.

The next section introduces notations and the lazy clause generation solving approach. We then explore modelling choices that arise in lazy clause generation solving, in particular we show that the choice of propagator can be independent of the choice of Boolean variable representation. In Section 4 we discuss the implementation of lazy clause generation and how it has to be extended to support new features of the modelling. We give experimental results in Section 5, and then conclude.

2 Lazy Clause Generation

2.1 Finite Domain Propagation

We consider a set of integer variables $\mathcal{V}$. A *domain* D is a complete mapping from $\mathcal{V}$ to finite sets of integers. We can understand a domain D as a formula $\wedge_{v \in \mathcal{V}}(v \in D(v))$ 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: $[\,l \,..\, u\,]$ denotes the set of integers $\{d \mid l \leqslant d \leqslant u, d \in \mathbb{Z}\}$. We assume an *initial domain* D_{init} such that all domains D that occur will be stronger i.e. $D \sqsubseteq D_{init}$.

A *valuation* θ is a mapping of variables to values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. 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 $v \in vars(\theta)$.

A constraint is a restriction placed on the allowable values for a set of variables. We define the *solutions* of a constraint c to be the set of valuations θ that make that constraint true, i.e. $solns(c) = \{\theta \mid (vars(\theta) = vars(c)) \ \wedge \ (\models \theta(c))\}$

We associate with every constraint c a set of *propagators*. A propagator f for c is a monotonically decreasing function on domains such that for all domains $D \sqsubseteq D_{init}$: $f(D) \sqsubseteq D$ and $\{\theta \in D \mid \theta \in solns(c)\} = \{\theta \in f(D) \mid \theta \in solns(c)\}$. This is a weak restriction since, for example, the identity mapping is a propagator for any constraint.

Example 1 A common propagator f_d for the constraint $x \neq y$ is

$$\begin{array}{rcll} f(D)(x) & = & D(x) - \{d\}, & \text{if } D(y) = \{d\} \\ f(D)(x) & = & D(x), & \text{otherwise} \\ f(D)(y) & = & D(y) - \{d\}, & \text{if } D(x) = \{d\} \\ f(D)(y) & = & D(y), & \text{otherwise} \\ f(D)(v) & = & D(v), & v \notin \{x, y\} \end{array}$$

Let $D_1(x) = \{3, 4, 5, 6\}$ and $D_1(y) = \{5\}$, then $f(D_1)(x_1) = \{3, 4, 6\}$ and $f(D_1)(y) = \{5\}$. $\square$

A *propagation solver* for a set of propagators F and current domain D, $solv(F, D)$, repeatedly applies all the propagators in F starting from domain D until there is no further change in resulting domain. $solv(F, D)$ is the weakest domain $D' \sqsubseteq D$ which is a fixpoint (i.e. $f(D') = D'$) for all $f \in F$. In other words, $solv(F, D)$ returns a new domain defined by

$$\begin{array}{rcl} solv(F, D) & = & \text{gfp}(\lambda d.iter(F, d))(D) \\ iter(F, D) & = & \sqcap_{f \in F} f(D). \end{array}$$

where gfp denotes the greatest fixpoint w.r.t $\sqsubseteq$ lifted to functions.

2.2 Atomic Constraints and Propagation Rules

In order to convert propagation to clauses we need to extract the "pointwise" behavior of a propagator. To do so we use atomic constraints and propagation rules.

An *atomic constraint* represents the basic changes in domain that occur during propagation. For integer variables, the atomic constraints represent the elimination of values from an integer domain, i.e. $x \leqslant d$, $x \geqslant d$, $x \neq d$ or $x = d$ where $x \in \mathcal{V}$ and d is an integer. Note these correspond to events in a propagation engine: upper bound change, lower bound change, domain change and fixing the variable. We also consider the atomic constraint *false* which indicates that unsatisfiability is the direct consequence of propagation.

Define a *propagation rule* as $C \rightarrowtail c$ where C is a conjunction of *atomic constraints*, and c is a single atomic constraint such that $\not\models C \rightarrow c$. A propagation rule $C \rightarrowtail c$ defines a propagator (for which we use the same notation) in the obvious way

$$(C \rightarrowtail c)(D)(v) = \begin{cases} \{\theta(v) \mid \theta \in D \cap solns(c)\} \\ \quad vars(c) = \{v\} \wedge \models D \rightarrow C \\ D(v) \quad \text{otherwise.} \end{cases}$$

In another words, $C \rightarrowtail c$ defines a propagator that removes values from D based on c only when D implies C.

A propagator f *implements* a propagation rule $C \rightarrowtail c$ iff $\models D \rightarrow C$ implies $\models f(D) \rightarrow c$ for all $D \sqsubseteq D_{init}$.

Example 2 The propagator f_d of Example 1 implements the following propagation rules (among many others) for $D_{init}(x) = D_{init}(y) = [\,l \,..\, u\,]$.

$$\begin{array}{rcll} x = d & \rightarrowtail & y \neq d, & l \leqslant d \leqslant u \\ y = d & \rightarrowtail & x \neq d, & l \leqslant d \leqslant u \end{array}$$

$\square$

A set of propagation rules $F \subseteq rules(f)$ *implements* f iff $solv(F, D) = f(D)$, for all $D \sqsubseteq D_{init}$.

In order to translate a propagator f to Boolean clauses we want to have a concise representation in terms of propagation rules, $rep(f)$, such that $rep(f)$ implements f.

Example 3 Consider the reified difference inequality $c \equiv b \Leftrightarrow x + c \leqslant y$ where $D_{init}(b) = \{0, 1\}$, $D_{init}(x) = [\,l \,..\, u\,]$, $D_{init}(y) = [\,l \,..\, u\,]$. Then a set of propagation rules $rep(f)$ implementing the domain propagator f for c is

$$\begin{array}{rcl} b \geqslant 1 \wedge x \geqslant d & \rightarrowtail & y \geqslant d + c \\ b \geqslant 1 \wedge y \leqslant d & \rightarrowtail & x \leqslant d - c \\ b \leqslant 0 \wedge x \leqslant d & \rightarrowtail & y \leqslant d + c - 1 \\ b \leqslant 0 \wedge y \geqslant d & \rightarrowtail & x \geqslant d - c + 1 \\ x \geqslant d - c + 1 \wedge y \leqslant d & \rightarrowtail & b \leqslant 0 \\ x \leqslant d \wedge y \geqslant d + c & \rightarrowtail & b \geqslant 1 \end{array}$$

where $l \leqslant d \leqslant u$, except for the last two where $l - c \leqslant d \leqslant u + c$. $\square$

A *bound propagation rule* only makes use of atomic constraints of the form $x \leqslant d$, $x \geqslant d$ and *false*. We can classify a propagator f as a *bounds propagator* if it has a representation $rep(f)$ which only makes use of bounds propagation rules.

Example 4 The propagator in Example 3 is clearly a bounds propagator. A bounds propagator f_b for the constraint $x \neq y$ is defined by the propagation rules for $D_{init}(x) = D_{init}(y) = [\, l \,..\, u \,]$ where $l \leqslant d \leqslant u$:

$$\begin{array}{lcl} x \leqslant d \wedge x \geqslant d \wedge y \leqslant d & \rightarrowtail & y \leqslant d-1 \\ x \leqslant d \wedge x \geqslant d \wedge y \geqslant d & \rightarrowtail & y \geqslant d+1 \\ y \leqslant d \wedge y \geqslant d \wedge x \leqslant d & \rightarrowtail & x \leqslant d-1 \\ y \leqslant d \wedge y \geqslant d \wedge x \geqslant d & \rightarrowtail & x \geqslant d+1. \end{array}$$

□

2.3 SAT and Unit Propagation

A *proposition* p is a Boolean variable from a universe of Boolean variables, $\mathcal{P}$. A *literal* l is either: a proposition p, its negation $\neg p$, the false literal $\bot$, or the true literal $\top$. The *complement* of a literal l, $\neg l$ is $\neg p$ if $l = p$ or p if $l = \neg p$, while $\neg\bot = \top$ and $\neg\top = \bot$. A *clause* C is a disjunction of literals. An *assignment* is either a set of literals A excluding $\bot$ such that $\forall p \in \mathcal{P}.\{p, \neg p\} \not\subseteq A$, or the failed assignment $\{\!\{\bot\}\!\}$. We define $A \sqsubseteq \{\!\{\bot\}\!\}$, and $A \sqcup A' = A \cup A$ unless the union contains $\bot$ or $\{p, \neg p\}$ for some literal p in which case $A \sqcup A' = \{\!\{\bot\}\!\}$.

An assignment A *satisfies* a clause C if one of the literals in C appears in A. A *theory* T is a set of clauses. An assignment is a *solution* to theory T if it satisfies each $C \in T$.

A SAT solver takes a theory T and determines if it has a solution. Complete SAT solvers typically involve some form of the DPLL algorithm which combines search and propagation by recursively fixing the value of a proposition to either $\top$ (true) or $\bot$ (false) and using unit propagation to determine the logical consequences of each decision made so far. The unit propagation algorithm finds all unit resolutions of an assignment A with the theory T. It can be defined as follows where C denotes a clause:

$$\begin{array}{rcl} up(A, C) & = & \left\{ \begin{array}{ll} \{\!\{\bot\}\!\} & \forall l \in C. \neg l \in A \\ A \sqcup \{l\} & \exists l \in C, , \neg l \notin A, \\ & \forall l' \in (C \setminus \{l\}). \neg l' \in A \\ A & \text{otherwise} \end{array} \right. \\ UP(A, T) & = & \text{lfp}.(\lambda a. \bigsqcup_{C \in T} up(a, C))(A) \end{array}$$

Example 5 Given the theory $T = \{\, \neg p_1 \vee p_2 \vee p_3 \vee \neg p_4 \vee \neg p_5, p_1 \vee p_2, p_4 \vee \neg p_5\}$ and the assignment $A_1 = \{\neg p_2, p_5\}$ unit propagation on $p_1 \vee p_2$ adds p_1, and on $p_4 \vee \neg p_5$ adds p_4, then unit propagation with the first clause adds p_3. Hence $UP(A_1, T) = \{p_1, \neg p_2, p_3, p_4, p_5\}$. □

2.4 Lazy Clause Generation

The lazy clause generation hybrid solver defined in [11] works as follows. We execute a SAT solver using a Boolean representation of the integer variables of the problem. When the SAT solver reaches an assignment A on these Boolean variables we calculate a corresponding domain D to A, and execute the propagators $f \in F$ on D. Any propagation rule r in $rep(f)$ that creates new information (that is $r(D) \not\equiv D$) is converted to a clause and added to the SAT solver. Unit propagation on this new clause will cause the assignment A to change to agree with $r(D)$.

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.

Not every assignment of Boolean variables is consistent with the integer variable x, for example $\{[\![x = 3]\!], [\![x \leqslant 2]\!]\}$ requires that x is both 3 and $\leqslant 2$. In order to ensure that assignments represent a consistent set of possibilities for the integer variable x we add the clauses $DOM(x)$ to the SAT solver

$$\begin{array}{rl} \neg[\![x \leqslant d]\!] \vee [\![x \leqslant d+1]\!] & l \leqslant d < u-1 \\ \neg[\![x = d]\!] \vee [\![x \leqslant d]\!] & l \leqslant d < u \\ \neg[\![x = d]\!] \vee \neg[\![x \leqslant d-1]\!] & l < d \leqslant u \\ [\![x = l]\!] \vee \neg[\![x \leqslant l]\!] & \\ [\![x = d]\!] \vee \neg[\![x \leqslant d]\!] \vee [\![x \leqslant d-1]\!] & l < d < u \\ [\![x = u]\!] \vee [\![x \leqslant u-1]\!] & \end{array}$$

These clauses 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 unit fixpoint A of $DOM(x)$ can be converted to a domain for variable x:

$$\begin{array}{rl} domain(A)(x) = \{ & d \in D_{init}(x) \mid \forall [\![c]\!] \in A. \\ & vars(l) = \{x\} \Rightarrow x = d \models c\} \end{array}$$

that is the domain of all values for x that are consistent with all the Boolean variables related to x.

Example 6 For example the assignment $A = \{[\![x_1 \leqslant 10]\!], \neg[\![x_1 \leqslant 5]\!], \neg[\![x_1 = 7]\!], \neg[\![x_1 = 8]\!], [\![x_2 \leqslant 11]\!], \neg[\![x_2 \leqslant 5]\!], [\![x_3 \leqslant 10]\!], \neg[\![x_3 \leqslant -2]\!]\}$ is consistent with $x_1 = 6, x_1 = 9$ and $x_1 = 10$. hence $domain(A)(x_1) = \{6, 9, 10\}$. For the remaining variables $domain(A)(x_2) = [\, 6 \,..\, 11 \,]$ and $domain(A)(x_3) = [\, -1 \,..\, 10 \,]$. Note that for brevity A is not a fixpoint of $DOM(x_1)$ since we are missing many implied literals such as $\neg[\![x_1 = 5]\!]$, $\neg[\![x_1 = 12]\!]$, etc. □

The propagators F are run on the created domain, and each propagation rule that creates new information is converted to a Boolean clause. This is straightforward since we can map atomic constraints to Boolean literals. The mapping *lit* is defined as: (where $D_{init}(x) = [\,l\,..\,u\,]$)

$$\begin{array}{rcl}
lit(false) & = & \bot \\
lit(x = d) & = & \left\{ \begin{array}{ll} [\![x = d]\!] & l \leqslant d \leqslant u \\ \bot & \text{otherwise} \end{array} \right. \\
lit(x \neq d) & = & \left\{ \begin{array}{ll} \neg[\![x = d]\!] & l \leqslant d \leqslant u \\ \top & \text{otherwise} \end{array} \right. \\
lit(x \leqslant d) & = & \left\{ \begin{array}{ll} \top & d \geqslant u \\ \bot & d < l \\ [\![x \leqslant d]\!] & \text{otherwise} \end{array} \right. \\
lit(x \geqslant d) & = & \left\{ \begin{array}{ll} \top & d \leqslant l \\ \bot & d > u \\ \neg[\![x \leqslant d-1]\!] & \text{otherwise} \end{array} \right.
\end{array}$$

We can transform a propagation rule r to a clause $cl(r)$ by:

$$cl(C \rightarrowtail c) = (\bigvee_{c' \in C} \neg lit(c')) \vee lit(c)$$

Example 7 Given the domain D corresponding to assignment A from Example 6, imagine a propagator f fires the propagation rule

$$x_1 \leqslant 10 \wedge x_2 \geqslant 6 \rightarrowtail x_3 \leqslant 1$$

This is transformed into the clause

$$\neg[\![x_1 \leqslant 10]\!] \vee [\![x_2 \leqslant 5]\!] \vee [\![x_3 \leqslant 1]\!]$$

This clause is added to the SAT solver. Unit propagation using the assignment A and the clause above adds the new information $[\![x_3 \leqslant 1]\!]$ to get assignment A'. □

Just as we can convert an assignment A to a domain D, we can convert a domain D to an assignment

$$\begin{array}{rcl}
assign(D, x) & = & \{lit(c) \mid x \in D(x) \models c, \\
 & & \quad x \in vars(x)\} \\
assign(D) & = & \left\{ \begin{array}{ll} \{\!\{\bot\}\!\} & \exists v \in \mathcal{V}.D(v) = \varnothing \\ \bigcup_{v \in \mathcal{V}} assign(D, v) & \text{otherwise} \end{array} \right.
\end{array}$$

Using the lazy clause generation we can show that the SAT solver maintains an assignment which is equivalent to the domains. In particular if we have clauses representing all the propagators F then unit propagation is guaranteed to be at least as strong as finite domain propagation.

Theorem 1 ([11]) *Let $rep(f)$ be a set of propagation rules implementing propagator f. Let $A = UP(assign(D), DOM \cup \bigcup\{cl(r) \mid f \in F, r \in rep(f)\})$. Then $A = \{\!\{\bot\}\!\}$ or $A \supseteq assign(solv(F, D))$.* □

3 Modelling Choices

Lazy clause generation proved to be a powerful approach to tackling finite domain problems with large amounts of search. In [11] we show that it can solve hard open shop scheduling problems more efficiently than pure SAT approaches and other finite domain solvers using the same model (Laborie [6] shows how to tackle hard scheduling problems using finite domains solvers by using complex resource constraints and specialized searching methods).

In this paper we explore some of the modelling possibilities that the novel solving technology of lazy clause generation allows.

3.1 Laziness and Eagerness

An important choice in the lazy clause generation approach is whether to implement a propagator lazily (which is the default) or eagerly. The eager representation of a propagator f simply adds the clauses $cl(r)$ for all $r \in rep(f)$ into the SAT solver before beginning the search. This clearly can improve search, since more information is known apriori, but the size of the clausal representation may make it inefficient.

Example 8 The representation of the domain propagator for disequality $x \neq y$ where $D_{init}(x) = D_{init}(y) = [\,l\,..\,u\,]$ requires $2(u - l + 1)$ binary clauses. Hence it is possible to model eagerly.

The representation of the bounds propagator for $x_1 + \cdots + x_n \leqslant k$ where $D_{init}(x_1) = \cdots = D_{init}(x_n) = [\,0\,..\,1\,]$ has ${}^nC_k = n!/((n-k)!k!)$ propagation rules. Clearly it is impossible to represent this eagerly for large n and k. □

In practice eager representation is useful for constraints that have very small representations.

3.2 Variable representation

The lazy clause generation approach represents variables domains of possible values in dual manner: a Boolean assignment and a domain D on integer variables. There are a number of choices of how we can represent integer variables in terms of Boolean variables. The default choice (full integer representation) is described in the previous section and was used in [11]. We present new choices below.

3.2.1 Non-continuous variables

We can represent an integer variable where $D_{init}(x) = \{d_1, \ldots, d_n\}$ where $d_i < d_{i+1}, 1 \leqslant i \leqslant n$, and the values are noncontinuous. This requires fewer Boolean variables, and fewer domain constraints then representing the domain $[\,d_1\,..\,d_n\,]$. The Boolean representation uses variables $[\![x = d_i]\!], 1 \leqslant i \leqslant n$ and $[\![x \leqslant d_i]\!], 1 \leqslant i < n$.

The clauses $DOM(x)$ required to maintain consistency of the Boolean assignment are:

$$\begin{array}{rl}
\neg[\![x \leqslant d_i]\!] \vee [\![x \leqslant d_{i+1}]\!] & 1 \leqslant i < n-1 \\
\neg[\![x = d_i]\!] \vee [\![x \leqslant d_i]\!] & 1 \leqslant i < n \\
\neg[\![x = d_i]\!] \vee \neg[\![x \leqslant d_{i-1}]\!] & 1 < i \leqslant n \\
[\![x = d_1]\!] \vee \neg[\![x \leqslant d_1]\!] & \\
[\![x = d_i]\!] \vee \neg[\![x \leqslant d_i]\!] \vee [\![x \leqslant d_{i-1}]\!] & 1 < i < n \\
[\![x = d_n]\!] \vee [\![x \leqslant d_{n-1}]\!] &
\end{array}$$

3.2.2 Bounds variables

We can represent an integer variable only using the bounds variables $[\![x \leqslant d]\!], l \leqslant d < u$ where $D_{init}(x) = [\,l\,..\,u\,]$. While this means we cannot represent all possible subset of $[\,l\,..\,u\,]$, it has the advantage of requiring fewer Boolean variables, and the domain representation requires only the clauses:

$$\neg[\![x \leqslant d]\!] \vee [\![x \leqslant d+1]\!] \qquad l \leqslant d < u-1$$

3.2.3 Non-continuous bounds variables

We can clearly restrict the representation of non-continuous variables to bounds only analogously, just using the Boolean variables $[\![x \leqslant d_i]\!]$

3.3 Propagator and variable representation independence

In a usual finite domain solver we are restricted so that if we use bounds variables, they must be restricted to only occur in bounds propagators. Indeed in [11] we use this observation to avoid using full integer variables for variables that only occur in bounds propagators. In the lazy clause generation solver we can separate the variable representation from the propagator type. To do so we make use of the more flexible clausal representation of propagators of the lazy clause generation solver.

With this separation the propagation engine can work without knowing whether integer variable x is a full integer, non-continuous, or bounds variable, since the translation of assignments to domains, and from propagation rules to clauses, completely captures the relationship between the Boolean representation and the integer variable.

Because of this separation we can independently choose which propagator we will use to represent a problem, without considering the variable representation. Hence for an individual constraint we can choose any of the propagators for that constraint.

3.3.1 Non-continuous variables

We extend the translation of atomic constraints *lit* to map atomic constraints involving non-continuous variable x where $D_{init}(x) = \{d_1, \ldots, d_n\}$ as follows:

$$
\begin{aligned}
lit(x = d) &= \begin{cases} \bot & d \notin \{d_1,\ldots,d_n\} \\ [\![x = d_i]\!] & d = d_i \end{cases} \\
lit(x \neq d) &= \begin{cases} \top & d \notin \{d_1,\ldots,d_n\} \\ \neg[\![x = d_i]\!] & d = d_i \end{cases} \\
lit(x \leqslant d) &= \begin{cases} \top & d >= d_n \\ \bot & d < d_1 \\ [\![x \leqslant d_i]\!] & d_i < d \leqslant d_{i+1} \end{cases} \\
lit(x \geqslant d) &= \begin{cases} \top & d \leqslant d_1 \\ \bot & d > d_n \\ \neg[\![x \leqslant d_i]\!] & d_i < d \leqslant d_{i+1} \end{cases}
\end{aligned}
$$

Note that each atomic constraint is translated as a single literal.

Example 9 Consider the translation of the propagation rules $x = 3 \rightarrowtail y \neq 3$ and $x \neq 3 \rightarrowtail y = 3$, where $D_{init}(x) = \{0,3,5\}$ and $D_{init}(y) = \{1,2,4\}$. The resulting clauses are $\neg[\![x = 3]\!] \vee \top$ or $\top$ (the always true clause) and $[\![x = 3]\!] \vee \bot$ or equivalently $[\![x = 3]\!]$. □

3.3.2 Bounds variables

We extend the translation of atomic constraints *lit* to map atomic constraints involving bounds variable x where $D_{init}(x) = [\,l\,..\,u\,]$ as follows:

$$
lit(x = d) = \begin{cases} [\![x \leqslant d]\!] & d = l \\ [\![x \leqslant d]\!] \wedge \neg[\![x \leqslant d-1]\!], & l < d < u \\ \neg[\![x \leqslant u-1]\!] & d = u \\ \bot & \text{otherwise} \end{cases}
$$

$$
lit(x \neq d) = \begin{cases} \neg[\![x \leqslant d]\!] & d = l \\ \neg[\![x \leqslant d]\!] \vee [\![x \leqslant d-1]\!], & l < d < u \\ [\![x \leqslant u-1]\!] & d = u \\ \top & \text{otherwise} \end{cases}
$$

The translations of $x \leqslant d$ and $x \geqslant d$ are as for full integer variables. Note that these translations now no longer guarantee to return a single literal.

Clearly "Boolean integer" variables x where $D_{init}(x) = [\,0\,..\,1\,]$ can be represented as bounds only variables without loss of expressiveness since $x \leqslant 0 \leftrightarrow x = 0 \leftrightarrow \neg(x = 1)$.

We can translate any propagation rule to a conjunction of clauses by simply applying *lit* as before. This creates (a possibly non-clausal) Boolean formulae which can be transformed to conjunctive normal form.

Example 10 Consider the translation of the propagation rule $x = 3 \rightarrowtail y \neq 3$, where x and y are bounds only variables. The resulting formula is $\neg[\![x \leqslant 3]\!] \vee [\![x \leqslant 2]\!] \vee [\![y \leqslant 2]\!] \vee \neg[\![y \leqslant 3]\!]$, which is a clause already.

Consider the translation of the propagation rule $x \neq 3 \rightarrowtail y = 3$. The resulting formula is $\neg([\![x \leqslant 2]\!] \vee \neg[\![x \leqslant 3]\!]) \vee ([\![y \leqslant 3]\!] \wedge \neg[\![y \leqslant 2]\!])$. The conjunctive normal form is

$$
\begin{array}{l}
\neg[\![x \leqslant 2]\!] \vee [\![y \leqslant 3]\!] \\
[\![x \leqslant 3]\!] \vee [\![y \leqslant 3]\!] \\
\neg[\![x \leqslant 2]\!] \vee \neg[\![y \leqslant 2]\!] \\
[\![x \leqslant 3]\!] \vee \neg[\![y \leqslant 2]\!]
\end{array}
$$

It would appear that the conversion of propagation rules including bounds variables could lead to an exponential explosion in the number of clauses required to represent them. By restricting the conversion of the rules to clauses which may actually be able to cause unit propagation, in fact we can represent them with at most 2 clauses.

Lemma 1 *If domain $D = domain(A)$ is such that $D(x) \models x \neq d$ where x is a bounds only variable, then $D(x) \models x \geqslant d+1$ or $D(x) \models x \leqslant d-1$.*

Proof: Now A can only include literals $[\![x \leqslant d']\!]$ or $\neg[\![x \leqslant d']\!]$ for some d'. Hence $domain(A)(x)$ is a range domain. If $D(x) \models x \neq d$ then either $D(x) \models x \geqslant d+1$ or $D(x) \models x \leqslant d-1$. □

Define the bounds simplification $bs(r)$ of a propagation rule $r \equiv C \rightarrowtail c$, for domain $D = domain(A)$ for some assignment A which fires the rule, as follows. Replace each atomic constraint $x \neq d$ appearing in C where x is a bounds only variable by either $x \leqslant d-1$ or $x \geqslant d+1$, whichever holds in D. The resulting propagation rule can create at most 2 clauses.

Theorem 2 *The conjunctive normal form of the clausal representation of $bs(r)$ involves at most 2 clauses.*

Proof: Each atomic constraint appearing in the left hand side of $bs(r)$ is translated as a single Boolean literal. The only conjunction that can occur in the translation is if the right hand side is an atomic constraint $x = d$ and x is a bounds variable. The resulting CNF has two clauses. □

Example 11 Consider the translation of the propagation rule $r \equiv x \neq 3 \rightarrowtail y = 3$ where x and y are bounds variables ranging over $[\,0\,..\,10\,]$. Suppose the domain that causes it to fire is $D = domain(A)$ where $A = \{[\![x \leqslant 1]\!]\}$. Then $D(x) = [\,0\,..\,1\,]$ and $D(x) \models x \leqslant 2$ and $bs(r) \equiv x \leqslant 2 \rightarrowtail y = 3$. The translation to Booleans is the formula $\neg[\![x \leqslant 2]\!] \vee ([\![y \leqslant 3]\!] \wedge \neg[\![y \leqslant 2]\!])$, which in CNF is $(\neg[\![x \leqslant 2]\!] \vee [\![y \leqslant 3]\!]) \wedge (\neg[\![x \leqslant 2]\!] \vee \neg[\![y \leqslant 2]\!])$. Note that the two clauses from Example 10 that are missing could not fire in A. □

There is an important new behaviour that arises when we consider using domain propagators on bounds variables. The result of propagation is always a clause of a form

$$cl(C \rightarrowtail c) = \vee_{c' \in C}(\neg\, lit(c')) \vee lit(c),$$

where $\neg\, lit(c')$ are all false in the current assignment and $lit(c)$ is either undefined or false in the current assignment. Previously $lit(c)$ was always a single literal, hence we could guarantee unit propagation would apply, and set $lit(c)$ to true. Now there is a possibility that $lit(c)$ is itself a disjunction and unit propagation will not apply.

Example 12 Consider the execution of the domain propagation for $x \neq y$ (Example 1) where x and y are bounds variables on the assignment $A = \{[\![x \leqslant 3]\!], \neg[\![x \leqslant 2]\!]\}$. Then in the corresponding $domain(A)(x) = \{3\}$ and the propagation rule $x = 3 \rightarrowtail y \neq 3$ fires. The resulting clause is $\neg[\![x \leqslant 3]\!] \vee [\![x \leqslant 2]\!] \vee \neg[\![y \leqslant 3]\!] \vee [\![y \leqslant 2]\!]$. No unit propagation is possible using A and this new clause.

In fact the domain propagator for $x \neq y$ applied to bounds variables x and y generates exactly the same clauses as the bounds propagator, but it generates them earlier! □

3.3.3 Disjunctive propagators

The discussion of the end of the last subsection motivates examining a new possibility. Propagation rules are designed so that the result of the propagation is a single atomic constraint, which can then be represented immediately as a change in domain. Given that we will convert the propagation rules to clauses in any case we can extend them to allow disjunction on the right hand side. A *disjunctive propagation rule* has the form $c_1 \wedge \cdots \wedge c_n \rightarrowtail c_{n+1} \vee \cdots \vee c_m$. The translation to clauses is clear $cl(c_1 \wedge \cdots \wedge c_n \rightarrowtail c_{n+1} \vee \cdots \vee c_m) = \neg\, lit(c_1) \vee \cdots \vee \neg\, lit(c_n) \vee lit(c_{n+1}) \vee \cdots \vee lit(c_{n+m})$. Presently we restrict our implementation to only support disjunctive propagation rules with at most two literals on the right hand side.

Example 13 Consider the constraint $|x - y| \geqslant k$ for constant $k > 0$. The bounds propagator for this constraint has representation given by the propagation rules: (where $l + k > u - k$)

$$\begin{array}{rcl}
x \geqslant l \wedge x \leqslant u \wedge y \leqslant l + k - 1 & \rightarrowtail & y \leqslant u - k \\
x \geqslant l \wedge x \leqslant u \wedge y \geqslant u - k + 1 & \rightarrowtail & y \geqslant l + k \\
y \geqslant l \wedge y \leqslant u \wedge x \leqslant l + k - 1 & \rightarrowtail & x \leqslant u - k \\
y \geqslant l \wedge y \leqslant u \wedge x \geqslant u - k + 1 & \rightarrowtail & x \geqslant l + k
\end{array}$$

A more eager version of this propagator fires when the range on one variable is small enough to guarantee some (disjunctive) constraints on the other variable. It is defined by the disjunctive propagation rules: (where $l + k > u - k$)

$$\begin{array}{rcl}
x \geqslant l \wedge x \leqslant u & \rightarrowtail & y \geqslant l + k \vee y \leqslant u - k \\
y \geqslant l \wedge y \leqslant u & \rightarrowtail & x \geqslant l + k \vee x \leqslant u - k
\end{array}$$

Disjunctive propagators can be seen as a more eager form of lazy clause generation.

4 Implementation

The creation of a practical lazy clause generation solver involves many more considerations than were addressed in Section 2.4. To build the system we add a cut down propagation engine into a SAT solver and modify it as a lazy clause generator. We first describe this process as defined in [11] and then describe the extensions required.

The SAT solver applies unit propagation, and when it reaches a fixpoint it calls the propagation engine. The new literals set by the SAT solver are converted into domain changes in the propagation solver, and these "events" queue up propagators for execution.

The first propagator in the queue is then executed. If it causes propagation, then the clausal representation of the first propagation rule that fires is added to the SAT solver and unit propagation is applied. When the SAT solver finishes we rexecute the same propagator (which is still at the head of the queue) to search for another firing propagation rule. When there are no more firing rules the propagator is removed from the queue and the next propagator considered. The reason we add clauses as soon as possible is to detect failure as soon as possible. Unit propagations may schedule (or re-schedule) propagators. The process continues until the propagation queue is empty and unit propagation is at fixpoint. At this point the SAT solver makes a decision about a literal to set *true* and search continues.

On failure the propagation queue is cleared, and the SAT solver backtracks up the trail of decided and inferred literals. For each canceled literal we undo the domain change on the corresponding integer variable in the propagation engine.

A subtle point we have not addressed is why we do not worry about a propagator creating duplicates of clauses corresponding to its propagation rules, particularly since we can execute the propagator repeatedly simply to create all the propagation rules that fire for one domain. The reason is that since a propagator f is only run at domain $D = \mathit{domain}(A)$ for an assignment A which is a unit propagation fixpoint, then if $\mathit{cl}(r)$ is already in the SAT solver then r cannot fire on domain D (it has no new information).

Example 14 Consider the propagation of the constraint $x = y$ with $D_{init}(x) = D_{init}(y) = [\,0\,..\,4\,]$. After the SAT solver sets $\neg[\![x = 2]\!]$ and $\neg[\![y = 3]\!]$ the first propagation rule that fires is $x \neq 2 \rightarrowtail y \neq 2$. This is added as the clause $[\![x = 2]\!] \vee \neg[\![y = 2]\!]$ and propagated to set $\neg[\![y = 2]\!]$. Returning to the propagation engine the, the propagator for $x = y$ is still head of the queue. The original propagation rule no longer fires since $y \neq 2$ is not new information. Hence the next propagation rule $y \neq 3 \rightarrowtail x \neq 3$ is considered. $\square$

The extensions of lazy clause generation we consider in this paper require modifications to the implementation. The reason is that using domain propagators on bounds variables, or more generally disjunctive propagators means that we can not be sure that a newly added clause does not already exist (has not previously been added) since it may not cause unit propagation with the current assignment.

This requires two modifications to the approach. First disjunctive propagators at the head of the queue must store an index of propagation rule processed last, and clear this index every time the propagator queue is cleared. This is to avoid them regenerating the same propagation rule when they are still the head of the queue. Secondly, before adding a clause corresponding to a disjunctive propagation rule we need to check that it is not already in the SAT solver.

We could build a separate data structure to record which clauses have been sent to the solver. To avoid the complexity and space required to do this we re-use existing data structures in the SAT solver. Suppose a propagation rule $C \rightarrowtail c_1 \vee c_2$ already has its corresponding clause Cl in the SAT solver. All literals in the clause except $\mathit{lit}(c_1)$ and $\mathit{lit}(c_2)$ must be false in the current assignment, otherwise the propagation rule would not fire. The SAT solver keeps track of at least two literals in each clause which are not false, the so-called *watched literals*, in order to detect unit propagations. Hence $\mathit{lit}(c_1)$ and $\mathit{lit}(c_2)$ must be the watched literals for Cl. To check if Cl appears in the SAT solver already, we check all clauses where $\mathit{lit}(c_1)$ is a watched literal (the SAT solver provides this data structure), and see if one is identical to Cl.

This check is reasonably expensive, but much cheaper than looking at all clauses involving $\mathit{lit}(c_1)$ since it will be the watched literal in few of them.

5 Experimental results

All experiments are performed on a 3.4GHZ Intel Pentium D with 4Gb RAM running on Debian Linux 4. The lazy clause generation system is built using MiniSat [9] version 2.0 beta. We compare our results against a highly optimized propagation solver Gecode 1.3.1 [3]. Eager models are run on MiniSat version 2.0 beta.

5.1 `all_different` propagators

The disequality `all_different`$([x_1, \ldots, x_n])$ constraint requires that $\forall 1 \leqslant i < j \leqslant n, x_i \neq x_j$.

In the lazy clause generation solver we can represent the disequality constraint $x \neq y$ in a number of ways: (a) using the domain propagator f_d from Example 1, (b) using the bounds propagator f_b from Example 4, and (c) using (bounds) propagators F_r for the reified set of constraints $b_1 \vee b_2$, $b_1 \Leftrightarrow x{+}1 \leqslant y$, $b_2 \Leftrightarrow y{+}1 \leqslant x$. In fact the last two representation have exactly the same propagation behaviour

Lemma 2 *Let $D(b_1) = D(b_2) = [\,0\,..\,1\,]$, then $\mathit{solv}(F_r, D) =_{\{x,y\}} \mathit{solv}(\{f_b\}, D)$.*

Proof: Suppose a propagation rule for f_b fires in D. Assume it has the form $x \geqslant d \wedge x \leqslant d \wedge y \geqslant d \rightarrowtail y \geqslant d + 1$, reasoning for the other rules is analogous. Then the propagation rule $y \geqslant d \wedge x \leqslant d \rightarrowtail b_2 \leqslant 0$ from $b_2 \Leftrightarrow y + 1 \leqslant x$ fires. Hence the propagation rule $b_2 \leqslant 0 \rightarrowtail b_1 \geqslant 1$ from $b_1 \vee b_2$ fires, and hence the rule $b_1 \geqslant 1 \wedge x \geqslant d \rightarrowtail y \geqslant d{+}1$ from $b_1 \Leftrightarrow x + 1 \leqslant y$ fires.

In the reverse direction suppose a propagation rule for F_r fires in D modifying x or y. Assume it is of the form $b_1 \geqslant 1 \wedge x \geqslant d \rightarrowtail y \geqslant d + 1$, reasoning for other rules is analogous. Then since $b_1 \geqslant 1$ is true, and is not true in D, either a rule $x \leqslant d' \wedge y \geqslant d' + 1 \rightarrowtail b_1 \geqslant 1$ fires or $b_2 \leqslant 0 \rightarrowtail b_1 \geqslant 1$ fires.

Suppose a rule of the first kind fired. Now $d' \geqslant d$ since $x \geqslant d$ and $x \leqslant d'$ both hold and $d + 1 > d' + 1$ otherwise $y \geqslant d + 1$ is not new information. This is a contradiction

Hence the second rule must fire. Since $b_2 \leqslant 0$ is now true, a rule of the form $y \geqslant d'' \wedge x \leqslant d'' \rightarrowtail b_2 \leqslant 0$ must have fired for some d''. Since the first rule creates new information $y \geqslant d + 1$ is stronger that $y \geqslant d''$ hence $d \geqslant d''$. But since D ensures both $x \geqslant d$ and $x \leqslant d''$ we have that $d \geqslant d'' \geqslant x \geqslant d$, so $d = d''$. Hence D ensure that $x \geqslant d$, $x \leqslant d$ and $y \geqslant d$ and hence f_b fires the rule $x \geqslant d \wedge x \leqslant d \wedge y \geqslant d \rightarrowtail y \geqslant d + 1$, causing $y \geqslant d + 1$. $\square$

There are more complex propagators for `all_different`$([x_1, \ldots, x_n])$ (see the survey [15]) that implement more complex rules based on Hall sets [4]. A *hall set* H is a subset of $\{x_1, \ldots x_n\}$

such that $|H| \geqslant |S|$ where $S = \cup_{v \in H} D(v)|$. If $|H| > |S|$ the propagation rule is

$$\wedge_{v \in H} \wedge_{d \in D_{init}(v) - S}\, v \neq d \rightarrowtail \mathit{false}$$

If $|H| = |S|$ the propagation rules are for each $v' \in \{x_1, \ldots, x_n\} - H$ and $d' \in S$

$$\wedge_{v \in H} \wedge_{d \in D_{init}(v) - S}\, v \neq d \rightarrowtail v' \neq d'$$

The domain propagation of Regin [12] implements all propagation rules for all possible Hall sets. Given there are exponentially many Hall sets, these stronger propagators do not necessarily lead to advantage in lazy clause generation.

5.2 Quasigroup Completion Problems

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 constraints

$$\begin{array}{ll} \texttt{alldifferent}([x_{i1}, \ldots, x_{in}], & 1 \leqslant i \leqslant n \\ \texttt{alldifferent}([x_{1j}, \ldots, x_{nj}], & 1 \leqslant j \leqslant n \end{array}$$

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 examples from the 2006 Constraint Satisfaction Solver Competition [2].

Table 1 compares the user time for finding the first solution of quasigroup completion problems of size 15×15 for various modelling possibilities. The choices are 3 letter codes: **e**ager or **l**azy modelling, **b**ounds or **f**ull integer representation, and **b**ounds (f_b), **d**omain (f_d) or **r**eified (F_r) propagators for representing disequality. Note that for the eager approach with bound variable representation the clauses for the bound and domain propagator are exactly the same, and thus we write **eb(bd)** to denote **ebb** and **ebd**. We also compare against Gecode [3]. For eager modelling the time for constructing the clausal representation is included, it is either 0.01 or 0.02 seconds. The benchmarks 0–9 are satisfiable while 10–14 are unsatisfiable. We omit **lbr** from the tables, since they are not competitive for these benchmarks.

The eager approaches are best for these examples, while the **lfd** combination is the best lazy approach. This is interesting as the bounds representation is quite poor for the lazy approach, but better than the domain representation for the eager approach. The larger the search required the poorer Gecode performs in comparison to the SAT/hybrid approaches.

Table 2 shows the results on 25×25 QCP problems in order to see the trend for modelling choices as size increases. These problems are hard for Gecode, taking hours to complete. In 6 out of 15 instances **lfd** improves upon the eager approach **efd**, and overall it solves the whole suite faster. Even though QCP problems are small (the cost of eager clause generation is less than 0.10 seconds) the lazy approach avoids the overhead of examining many useless clauses, and hence starts outperforming the eager approach as the problem size grows. Interestingly **eb(bd)** is still better than the lazy approach **lfd** for these problems, even though the lazy bounds representations are poor. Examining the novel combination **lbd** we see that for 2 instances it gives the best results, and it suffers significantly greater overhead because it has to check for duplicate clauses. With a dedicated systems for duplicate clause checking it could be improved further.

Table 3 shows the search space for each approach. While **lfd** has the overhead of propagator execution compared to **efd** and **eb(bd)** it usually requires less search, since only the used claused are counted for the search heuristic. Clearly there is an overhead for the full integer representation. When **lbd** leads to around the same search space as **lfd** it is twice as fast.

5.3 Magic Squares Problems

A $n \times n$ *magic square* is a square of values $x_{ij}, 1 \leqslant i, j \leqslant n$ where each number in $[\,1\,..\,n^2\,]$ occurs exactly once and each row, column and major diagonal adds to the same number ($s = n(n^2+1)/2$). It is represented by one `alldifferent` constraint, and $2n + 2$ linear equations.

In Table 4 we compare various modelling choices for magic square problems, for finding the first solution (F) and all solutions (A)(for small problems). The $*$ entries arise since the eager approach **eb(bd)** could not search for all solutions (A) since this required modifying the SAT solver.

For these problems, the first fail search strategy of Gecode is clearly much better than the VSIDS search used by our hybrid. The eager modelling approach quickly fails since just the generation of the clauses for $\sum_{i=1}^{n} x_{ij} = s$ requires more than 400 seconds. The additional variables $[\![x = d]\!]$ in the full integer variable representation cause too much overhead for these example, the bounds representations are clearly superior. Of these the hybrid disjunctive propagator performs well. Interestingly **lbr** which has the same propagation strength as **lbb** is superior on the harder problems. This may be because nogoods can make use of the Boolean reification variables to record more pertinent information about failures.

5.4 CELAR Radio Link Frequency Assignment Problems

The CELAR Radio Link Frequency Assignment Problems [1] consist of a set of radio frequencies and a set of radio links to assign a frequency to each radio link. Some pairs of radio links must be an exact distance apart in frequency, while other should be at least some distance apart. We use the first 5 problems (where all constraints are mutually satisfiable) while minimizing the maximum frequency used. The set of possible frequencies F is non-continuous:

$$\begin{array}{c} \{2 + 14i | 1 \leqslant i \leqslant 11\} \cup \{2 + 14i | 18 \leqslant i \leqslant 28\} \\ \cup \{8 + 14i | 29 \leqslant i \leqslant 30\} \cup \{8 + 14i | 46 \leqslant i \leqslant 56\}, \end{array}$$

Table 1: QCP 15 × 15 instances: user time

Benchmark	**Time(sec)**					
	efd	**eb(bd)**	**lfd**	**lbd**	**lbb**	**gecode**
qcp-15-120-0_ext	0.05	0.02	0.03	0.14	0.64	0.02
qcp-15-120-1_ext	0.04	0.04	0.06	0.22	0.74	0.08
qcp-15-120-2_ext	0.08	0.02	0.05	0.16	0.74	454.53
qcp-15-120-3_ext	0.05	0.04	0.14	0.26	0.84	0.19
qcp-15-120-4_ext	0.20	0.02	0.02	0.33	0.65	5.50
qcp-15-120-5_ext	0.15	0.09	0.21	0.62	2.52	117.08
qcp-15-120-6_ext	0.04	0.02	0.01	0.17	1.01	38.01
qcp-15-120-7_ext	0.11	0.13	0.29	0.24	0.97	1.28
qcp-15-120-8_ext	0.05	0.10	0.04	0.18	0.76	6.70
qcp-15-120-9_ext	0.08	0.14	0.24	0.27	0.78	1685.44
qcp-15-120-10_ext	0.06	0.04	0.04	0.20	0.55	1044.80
qcp-15-120-11_ext	0.03	0.05	0.01	0.32	0.41	47.64
qcp-15-120-12_ext	0.03	0.01	0.02	0.04	0.41	862.29
qcp-15-120-13_ext	0.16	0.30	0.17	0.21	1.57	179.18
qcp-15-120-14_ext	0.02	0.01	0.01	0.01	0.62	2034.72
Arith mean	0.08	0.07	0.09	0.22	0.88	431.83
Geom mean	0.06	0.04	0.05	0.17	0.78	24.67

Table 2: QCP 25 × 25: user time

Benchmark	**Time(sec)**				
	efd	**eb(bd)**	**lfd**	**lbd**	**lbb**
qcp-25-264-0_ext	114.07	65.56	149.88	85.89	242.73
qcp-25-264-1_ext	832.31	108.37	99.84	374.77	1346.06
qcp-25-264-2_ext	15.40	44.40	12.25	47.34	144.92
qcp-25-264-3_ext	542.61	273.36	442.57	532.47	1655.22
qcp-25-264-4_ext	265.00	268.84	24.87	418.33	1136.17
qcp-25-264-5_ext	108.60	146.36	341.25	158.62	4810.77
qcp-25-264-6_ext	255.60	185.53	130.06	127.91	871.80
qcp-25-264-7_ext	35.36	1.52	34.07	78.26	269.61
qcp-25-264-8_ext	9.52	48.36	81.10	171.35	998.53
qcp-25-264-9_ext	27.80	153.52	286.20	710.96	1043.52
qcp-25-264-10_ext	30.92	125.67	165.77	346.78	631.13
qcp-25-264-11_ext	0.14	0.06	0.10	0.17	7.16
qcp-25-264-12_ext	0.23	0.21	0.24	0.32	11.90
qcp-25-264-13_ext	0.36	0.29	0.34	0.34	9.83
qcp-25-264-14_ext	107.82	131.88	175.01	176.97	901.36
Arith mean	156.38	103.60	129.57	215.37	938.71
Geom mean	26.40	23.75	30.31	53.07	326.03

using only 44 values in the range $[16..792]$ of 777 possible values. We model the problem using bounds propagators for $|x-y| \geqslant k$ (see Example 13), and model $|x-y| = k$ using the bounds propagators for $|x-y| \geqslant k \wedge x-y \leqslant k \wedge y-x \leqslant k$.

We compare the **f**ull integer representation, **n**on-continuous representation, **b**ounds representation, and n**o**n-continuous bounds representation. For the full integer representation we statically add constraints $\neg[\![x = d]\!], d \in [16..792] - F$ to the SAT solver, while for the bounds representation we statically add the constraints $\neg[\![x \leqslant d_i]\!] \vee [\![x \leqslant d_{i+1}]\!]$ where d_i and d_{i+1} are consecutive values in F. We also compare with Gecode using reified constraints to represent $|x-y| \geqslant k$ as $x-y \geqslant k \vee y-x \geqslant k$.

The results for the various modelling choices are shown for: user time in Table 6, failures in Table 7, and unit propagation executed in Table 8. Clearly the non-continuous representations are significantly better than the continuous representations, they involve around 20× fewer variables. The failure results show that it is not the results of a better search because there are fewer Boolean variables to branch on, instead it is simply the overhead of more unit propagations to deal with the larger number of variables.

Table 6: CELAR problems: user time

Prob	**User Time(sec)**				
	lfb	**lnb**	**lbb**	**lob**	**gecode**
scen01	285.22	13.67	104.65	9.37	> 400
scen02	2.03	0.16	0.86	0.11	> 400
scen03	39.90	3.16	20.06	2.19	> 400
scen04	2.17	0.16	0.88	0.10	0.46
scen05	2.25	0.17	0.96	0.10	0.34

This clearly shows the benefit of separation of propagator implementation from variable representation. The propagator is highly effective on

Table 3: QCP 25 × 25: conflicts (000s)

Benchmark	**Conflicts/Failures**				
	efd	**eb(bd)**	**lfd**	**lbd**	**lbb**
qcp-25-264-0_ext	212	117	159	174	588
qcp-25-264-1_ext	1037	178	119	626	2498
qcp-25-264-2_ext	44	99	29	125	463
qcp-25-264-3_ext	814	393	399	892	3284
qcp-25-264-4_ext	417	405	42	760	2424
qcp-25-264-5_ext	210	256	325	345	7890
qcp-25-264-6_ext	397	282	161	273	1701
qcp-25-264-7_ext	84	9.6	60	178	631
qcp-25-264-8_ext	30	96	102	352	2142
qcp-25-264-9_ext	70	261	291	1301	2307
qcp-25-264-10_ext	76	226	182	709	1503
qcp-25-264-11_ext	0.2	0.2	0.3	0.7	11
qcp-25-264-12_ext	1.6	3.1	2.1	2.8	48
qcp-25-264-13_ext	4.1	4.1	3.9	3.1	31
qcp-25-264-14_ext	192	208	170	352	1832
Arith mean	239	169	136	406	1824
Geom mean	64	61	53	137	736

Table 4: Magic squares: user time

nT	**User Time(sec)**					
	eb(bd)	**lfd**	**lbd**	**lbb**	**lbr**	**gecode**
3F	0.16	0.00	0.00	0.00	0.00	0.00
3A	*	0.00	0.00	0.00	0.00	0.00
4F	8.92	0.04	0.04	0.01	0.16	0.01
4A	*	866.38	745.87	810.84	803.09	2.26
5F	> 400	307.15	1.04	1.19	0.79	0.81
6F	> 400	31.87	0.39	99.92	17.50	0.00
7F	> 400	> 400	> 400	> 400	> 400	5.25

Table 7: CELAR problems: Conflicts/Failures

Prob	**Conflicts/Failures**				
	lfb	**lnb**	**lbb**	**lob**	**gecode**
scen01	5036	4542	4160	4247	—
scen02	202	127	180	261	—
scen03	3039	2380	2667	2553	—
scen04	7	6	2	1	31
scen05	17	22	36	24	74

Table 8: CELAR problems: unit propagations

Prob	**Unit Propagations**			
	lfb	**lnb**	**lbb**	**lob**
scen01	177561515	13081789	133403108	7133763
scen02	1969516	183084	1732660	112612
scen03	43087573	3608960	38598246	1918102
scen04	628192	36289	304949	17368
scen05	901257	65516	1375927	47145

the non-continuous Boolean representations without being modified.

Interestingly for these problems the disjunctive propagator explained in Example 13 does not improve upon the bounds propagator.

6 Related Work and Conclusion

The motivating earlier work for the lazy clause generation approach was twofold.

The paper [5] described a hybrid binary decision diagram (BDD) and SAT solver for solving problems involving set variables, which used the SAT solver as nogood engine for a BDD propagation solver. The hybrid leaves control of search to the BDD solver, and does not include integer variables. Lazy clause generation imbeds the propagation engine in the SAT solver and puts the SAT solver in charge of search. Set variables have only a single possible Boolean representation so the modelling choices we explore here do not arise.

The paper [14] explained how to statically encode linear arithmetic constraints into CNF (hence eager modelling) using the propositions $[\![x \leqslant d]\!]$. The approach is manifestly impractical when the linear constraint involves a significant number of variables (as illustrated by e.g. magic squares 5). The lazy clause generation approach makes the encoding of linear arithmetic possible for large linear constraints, and allows encoding of arbitrary propagators.

There are propagation solvers which allow different representation of integers, in particular Minion [8] and Gecode [3]. All representations either support all atomic constraints or are restricted in the propagators they can be used. The views ap-

Table 5: Magic squares: conflicts

nT	Conflicts/Failure					
	eb(bd)	**lfd**	**lbd**	**lbb**	**lbr**	**gecode**
3F	15	3	9	14	12	6
3A	*	31	34	43	34	36
4F	51	569	560	160	1326	892
4A	*	1000776	898347	1050572	869813	235545
5F	—	201531	7578	8149	3212	72227
6F	—	53332	2991	137545	21644	27
7F	—	—	—	—	—	481301

proach of Gecode [13] allows variables defined by simple constraints to be seen as mappings from atomic constraint to atomic constraints, and hence has some similarity with the mapping idea of this paper. For example a variable $y = x + 3$ effectively rewrites atomic constraint like $x \geqslant 4$ to $y \geqslant 6$ and vice versa. It would be useful to include views in the lazy clause generation solver, since it reduces the number of Boolean variables required.

In this paper we examine the modelling choices that arise when using the lazy clause generation hybrid solving approach devised in [11]. We find that the separation of choice of propagator from Boolean variable representation leads to an increased number of modelling choices. The direct representation of non-continuous variables is clearly advantageous, and there is some evidence that the use of disjunctive propagators (domain propagators for bounds variables) can improve upon other modelling approaches.

References

[1] B. Cabon, S. de Givrey, L. Lobjois, T. Schiex, and L.P. Warners. Radio link frequency assignment. *Constraints*, 4(1):78–89, 1999.

[2] CSP competition 2006. `http://cpai.ucc.ie/06/Competition.html`. [Jun07].

[3] GECODE. `www.gecode.org`. [Feb07].

[4] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, 10:26–30, 1935.

[5] P. Hawkins and P.J. Stuckey. A hybrid BDD and SAT finite domain constraint solver. In P. Van Hentenryck, editor, *Proceedings of the Practical Applications of Declarative Programming*, number 3819 in LNCS, pages 103–117. Springer-Verlag, 2006.

[6] P. Laborie. Complete MCS-based search: Application to resource constrained project scheduling. In *Proceedings IJCAI 2005*, pages 181–186, 2005.

[7] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[8] Minion. `minion.sourceforge.net`. [Feb07].

[9] MiniSat. `www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`. [Dec06].

[10] R. Niewenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In *LPAR'04*, volume 3452 of *LNAI*, pages 36–50, 2004.

[11] O. Ohrimenko, P.J. Stuckey, and M. Codish. Propagation = lazy clause generation. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, LNCS, page to appear. Springer-Verlag, 2007.

[12] J-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 1, pages 362–367, Seattle, WA, USA, 1994. AAAI Press.

[13] Guido Tack, Christian Schulte, and Gert Smolka. Generating propagators for finite set constraints. In Fréderic Benhamou, editor, *12th International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 575–589. Springer, 2006.

[14] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP to SAT. In *Proceedings of CP-2006*, volume 4204 of *LNCS*, pages 590–603, 2006.

[15] W.J. van Hoeve. The alldifferent constraint: a survey. `http://arxiv.org/abs/cs/0105015`, 2001.