Conflict Analysis Based on Cutting-Planes for Constraint Programming

Robbin Baauw
Delft University of Technology, The Netherlands

Maarten Flippo
Delft University of Technology, The Netherlands

Emir Demirović
Delft University of Technology, The Netherlands

Abstract

This paper introduces a novel constraint learning mechanism for Constraint Programming (CP) solvers that integrates cutting planes reasoning into the conflict analysis procedure. Drawing inspiration from Lazy Clause Generation (LCG), our approach, named Lazy Linear Generation (LLG), can generate linear integer inequalities to prune the search space, rather than propositional clauses as in LCG. This combines the strengths of constraint programming (strong propagation through global constraints) with cutting-planes reasoning. We present linear constraint explanations for various arithmetic constraints and the element constraint. An experimental evaluation shows that the improved generality of linear constraints has a practical impact on a CP solver by reducing the number of encountered conflicts in 45% of our benchmark instances. Our analysis and prototype implementation show promising results and are an important step towards a new paradigm to make constraint programming solvers more effective.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases constraint programming, learning, conflict analysis

Digital Object Identifier 10.4230/LIPIcs.CP.2025.4

Supplementary Material *Software (Source code and Evaluation)*: `https://doi.org/10.5281/zenodo.15591035`

Funding *Maarten Flippo*: Supported by the project "Towards a Unification of AI-Based Solving Paradigms for Combinatorial Optimisation" (OCENW.M.21.078) of the research programme "Open Competition Domain Science – M" which is financed by the Dutch Research Council (NWO).

1 Introduction

Constraint Programming [39] (CP) is an important paradigm for solving combinatorial optimization problems. It has applications in many domains, including resource allocation [41, 37], scheduling [32, 1], and verification [31, 3]. CP solvers use backtracking search algorithms to find solutions to models. Key to a good backtracking search algorithm is the ability to identify areas of the search space that do not contain solutions. Modern CP solvers use a combination of two types of reasoning to achieve this. The first is *propagation*, which is the process of identifying values that, based on the constraints in the problem, can never be part of a solution. The second is *conflict analysis*, which adds new constraints to the solver during the search process, which enables more propagation to occur.

Deriving new constraints is well-known to be beneficial to backtracking search algorithms [9]. Much work has been done to implement constraint learning effectively in CP solvers [20, 29, 38, 21, 24, 40]. Of the approaches, Lazy Clause Generation (LCG) [29, 38], is the most wide-spread, implemented by solvers such as OR-Tools [30] and Chuffed [8]. For many problems, constraint learning is crucial [38, 35] to the performance of a solver, as the learned constraints prune large parts of the search space.

All these solvers have in common that they reason over clausal constraints. However, other types of constraints can also be learned by CP solvers [40]. For paradigms other than CP, work has also been done exploring the learning of pseudo-Boolean (PB) constraints [12, 17] and integer linear constraints [27, 19, 2]. These systems have the potential of learning stronger constraints than clauses, although in practice more scientific and engineering efforts are needed to make these approaches as mature as the more studied clause-learning algorithms.

An example of conflict analysis on integer linear constraints can be found in the Integer Linear Programming (ILP) solver IntSat [27], which serves as a starting point of our approach. It stands out from other ILP solvers because it does not reason using the LP relaxation of the problem. Instead, it uses cutting planes reasoning and a generalized CDCL [22] algorithm, which combines integer linear constraints to derive new (implied) integer linear constraints. The method is promising, as it is already competitive with other state-of-the-art ILP solvers such as Gurobi [16]. One major difference between IntSat's conflict analysis procedure and a clausal conflict analysis procedure, is that in the former the analysis can fail to derive a new constraint that compactly describes the current conflict. Yet, the empirical evaluation shows that IntSat is effective despite this fact.

The effectiveness of cutting planes reasoning inspired our work with the question "How can CP solvers incorporate cutting planes reasoning?". As IntSat is heavily inspired by CDCL, this indicates that cutting planes reasoning could be incorporated in constraint programming similar to how LCG includes propositional CDCL. The major difference would be how high-level constraint inference is explained to the learning procedure, as the clausal explanation by LCG solvers are not applicable. We further observe that it may not be possible to generate a linear constraint as a reason for propagation without introducing new variables. This is in contrast to clausal explanations, where creating additional variables is supported but not required.

A CP solver that would come close to this idea is HaifaCSP [40], as it can do cutting planes reasoning to derive linear inequalities. However, it cannot explain propagations by arbitrary constraints as linear inequalities. As a result, as soon as a linear inequality interacts with another arbitrary constraint, the solver resorts to clausal learning. As we show in our experiments, the more clauses are present in the solver, the higher the chance that conflict analysis cannot derive a new linear inequality. To maximize the impact of the more general learning, we want the learning procedure to deal with linear inequalities as much as possible.

We present lazy linear constraint generation (LLG), an approach to use cutting planes reasoning within constraint programming. Our approach explains propagations with linear constraints, allowing propagations by arbitrary propagators to be used in the cutting planes conflict analysis procedure. To this end, we modify the conflict analysis procedure from the IntSat solver. We devise explanations for the integer multiplication, absolute value, truncating division, maximum, linear not equals, and element propagators. Our IntSat-based conflict analysis procedure combines the explanations to learn new linear constraints.

To determine whether the theoretical benefits of learning linear constraints have an impact in practice, we ran an empirical evaluation over 952 MiniZinc instances. In 443 of those, the solver terminates and learns at least one linear constraint. Among those instances, the conflict count is reduced, sometimes significantly. However, there are also many instances where the conflict analysis fails for the same reason as the IntSat conflict analysis fails. When we do learn a linear constraint, we show that it is generally more impactful than the clause that would be learned in that same situation. Additionally, we show that the CP propagators can be more effective with LLG than encoding the problem to linear constraints and solving with IntSat. Lastly, we show that resorting to clausal learning when no linear constraint can be learned remains essential, as omitting this step increases the number of conflicts significantly.

The rest of this paper is organized as follows: We start out with some background in Section 2, followed by a discussion of the related work in Section 3. Then, we will describe our contributions in Section 4. After that, we present our empirical evaluation in Section 5. Finally, we give our conclusions and outline ideas for future work in Section 6.

2 Background

2.1 Constraint Satisfaction Problem

A *constraint satisfaction problem* (CSP) is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where:

- $x \in \mathcal{X}$ is a *decision variable*,
- $\mathcal{D}(x) \in \mathcal{D}$ with $x \in \mathcal{X}$ is the *domain* of x, i.e. the set of values x can be assigned to,
- and $C \in \mathcal{C}$ is a *constraint*: a predicate over the variables that is either satisfied or violated.

We use range notation when the domain is a uninterrupted sequence of integers: $[l, u] = \{i | l \leq i \leq u\}$.

An *assignment* is a total function θ that maps every variable $x \in \mathcal{X}$ to a set $V \subseteq \mathcal{D}(x)$. If $|\theta(x_i)| > 1$, i.e. there is more than one possible value for x_i, then the assignment is referred to as a *partial assignment*. Otherwise, the assignment is called *total*. In this paper, unless we explicitly use the term "partial assignment", we refer to a total assignment. We abuse notation to say that $\theta(x) = v$ with $v \in \mathbb{Z}$ to mean that $\theta(x) = \{x\}$. If the assignment θ satisfies all the constraints in $\mathcal{C}$, then θ is called a *solution*. In this paper, we restrict ourselves to integer decision variables, i.e. $\forall x \in \mathcal{X} : \mathcal{D}(x) \subset \mathbb{Z}$, and we assume the domains are finite.

2.2 Constraint Programming

Constraint Programming (CP) is a paradigm for solving CSPs. CP solvers combine inference and search to find solutions to a CSP. The inference prunes the domain based on the constraints in the problem, and once no more inference can be done, the search splits the problem into subproblems to be solved independently. A *conflict* happens when there exists a variable $x \in \mathcal{X}$ such that $\mathcal{D}(x) = \emptyset$.

In a CP solver, constraints are enforced by propagators. A *propagator* is a function $p : \mathcal{D} \mapsto \mathcal{D}$ that takes the domain and removes values that do not exist in a solution. This means that $p(\mathcal{D}) \sqsubseteq \mathcal{D}$, where $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$ denotes that $\mathcal{D}_1$ is *stronger* than $\mathcal{D}_2$, i.e. for every $x \in \mathcal{X}$ it is the case that $\mathcal{D}_1(x) \subseteq \mathcal{D}_2(x)$. We highlight two types of constraints and their propagators.

- A *clause*, which is a disjunction of Boolean variables. It has the form $l_1 \vee \cdots \vee l_n$, where every l_i has a Boolean domain $\mathcal{D}(l_i) = \{0, 1\}$. This constraint requires at least one l_i to be 1. The propagator for a clause waits until $n - 1$ variables are fixed to 0, and then sets the final variable to 1.
- A *linear inequality* (also referred to as linear constraint in this paper) has the form $\sum w_i x_i \leq c$, where $w_i \in \mathbb{Z}$ and $c \in \mathbb{Z}$ are constants, and $x_i \in \mathcal{X}$ are decision variables. The propagator for this constraint performs bound propagation [5], as shown in Example 1.

▶ **Example 1.** Let x, y be integer decision variables with domains $\mathcal{D}(x) = [1, 5]$ and $\mathcal{D}(y) = [0, 2]$. The propagator for the constraint $x + 2y \geq 7$ can remove the values 1 and 2 from $\mathcal{D}(x)$ and the value 0 from $\mathcal{D}(y)$, because there are no solutions where $x = 2$ or $y = 0$.

2.3 Integer Linear Programs

An *Integer Linear Program* (ILP) is a CSP in which all constraints are linear inequalities. Given two linear constraints $A : a_1x_1 + \cdots + a_nx_n \leq a_0$ and $B : b_1x_1 + \cdots + b_nx_n \leq b_0$, their *linear combination* results in a new linear constraint $C : c_1x_1 + \ldots c_nx_n \leq c_0$ with $c_i = \alpha a_i + \beta b_i$ that is implied by $A \wedge B$. In the case where $c_i = 0$, we say that x_i has been *eliminated*. Note that we can always pick an α and β such that $c_i = 0$ when $a_ib_i < 0$, i.e., when the coefficients of the variable to be eliminated x_i have opposite signs in A and B.

The combination rule can be used by solvers like IntSat [27], which are inspired by Conflict-Driven Clause Learning (CDCL) [22]. IntSat keeps a trail of bounds in the form $\langle x \diamond v \rangle$, with $\diamond \in \{\leq, \geq\}$, that iteratively tighten the domain of the variables.

Algorithm 1 presents the pseudo-code for the conflict analysis procedure. Just like propositional CDCL, the trail gets traversed backwards. For every entry, it is checked whether reason constraint RC can be combined with the conflicting constraint CC to eliminate propagated variable V (lines 5 and 6). This elimination is feasible only if $V \in CC$ and the signs of the coefficients of V in CC and RC are opposite. If these conditions are met, the reason constraint RC is combined with the conflicting constraint CC (line 7). This process is repeated until the resulting constraint CC can propagate at an earlier decision level (*asserting*), or the previous decision is reached.

A notable difference with propositional CDCL, is that Algorithm 1 may fail to derive a linear constraint that is asserting at a prior decision level. In such cases, the solver resorts to performing resolution on bounds as a fallback strategy, similar to LCG conflict analysis (see next section), which is the clausal counterpart of linear combinations.

Algorithm 1 IntSat conflict analysis as described in [27].

Input: a set of linear constraints $\mathcal{C}$ and a trail T containing tuples (V, RC) representing a propagation of variable V by linear constraint RC
Output: a learned linear constraint or learned clause and corresponding backtrack level

1: $CC \leftarrow$ currently conflicting constraint in $\mathcal{C}$
2: ▷ ***Invariant****: CC is conflicting with the current assignment*
3: **while** top of trail is not a decision **do**
4: $(V, RC) \leftarrow$ PopTrail() ▷ Remove the last trail entry
5: **if** $V \notin CC$ **then continue end if** ▷ Variable not relevant
6: **if** $CC[V] \cdot RC[V] \geq 0$ **then continue end if** ▷ Signs equal, bound not relevant
7: $CC \leftarrow$ combination of CC and RC eliminating V
8: **for** backtrack level $\in$ 0..current decision level $-$ 1 **do**
9: **if** CC propagates a new bound at backtrack level **then**
10: **return** CC, backtrack level ▷ Early Backjump
11: **end if**
12: **end for**
13: **end while**
14: **return** Resolution-fallback$(\mathcal{C}, T)$

The reason for the inability to construct an asserting linear constraint is the *rounding problem*, which is illustrated by Example 2 (adapted from [27, Example 3.1]). As a consequence, linear conflict analysis can generate implied constraints that are not conflicting under the current partial assignment, even though the constraint that identified the conflict initially was conflicting with this assignment.

▶ **Example 2.** Consider constraints $c_1 : x + 2y \leq 2$ and $c_2 : x - 2y \leq 0$, with $\mathcal{D}(x) = [1, 3]$ and $\mathcal{D}(y) = [-5, 5]$. x's lower bound causes c_1 to propagate $2y \leq 1$, which after rounding becomes $y \leq 0$. This leads to a conflict with c_2. Combining c_2 and c_1, eliminating y, produces $2x \leq 2$, or $x \leq 1$. This new constraint is *not* conflicting with the current assignment, as $1 \in \mathcal{D}(x)$.

A CDCL-inspired ILP solver has to deal with the situation from Example 2 in some way. In Section 3 we highlight different approaches taken by various solvers.

2.4 Lazy Clause Generation

Lazy Clause Generation [28, 38] (LCG) is an approach to solving CSPs within the CP paradigm. It combines the domain propagation capabilities of CP solvers with the clause learning capabilities of SAT solvers.

There are two main parts to the integration. The first is the representation of the decision variables in a propositional formula. This is done by creating Boolean variables that map to unary constraints of the following form: $\langle x \diamond v \rangle$, where $x \in \mathcal{X}$, $\diamond \in \{\leq, \geq, \neq, =\}$, and $v \in \mathbb{Z}$. Such a unary constraint is called an *atomic constraint*. Every domain reduction in an LCG solver can be expressed by setting one or more atomic constraints to true.

The second part of the integration allows propagations done by propagators to be used during conflict analysis. Every propagation is explained by an implication $\bigwedge l_i \implies p$, where l_i and p are atomic constraints.

▶ **Example 3.** The explanations for the propagations in Example 1 are $\langle x \leq 5 \rangle \implies \langle y \geq 1 \rangle$ and $\langle y \leq 2 \rangle \implies \langle x \geq 3 \rangle$.

These explanations can be treated as clauses to integrate into the CDCL procedure, allowing the solver to learn clauses based on propagations done by propagators.

3 Related Work

It has long been established that learning can be beneficial to backtracking search algorithms [9]. The inclusion into CP solvers became popular when g-nogoods were introduced [20], which are a conjunction or disjunction of $\langle x \neq v \rangle$ constraints. LCG [29, 38] improves the conciseness of the explanations by introducing $\langle x \leq v \rangle$, $\langle x \geq v \rangle$, and $\langle x = v \rangle$ constraints, although the expressiveness of the two approaches is the same. G-nogoods are further generalized to c-nogoods [21, Chapter 5], implemented by Moore [24, Chapter 5]. A c-nogood can combine arbitrary constraints, not just atomic constraints. Finally, Veksler and Strichman [40] introduced the HaifaCSP solver, which is capable of learning constraints that are not clauses or conjunctions.

Of the CP learning approaches, LCG is the most widespread. Since its introduction, much work has been done to increase the impact that the constraint learning has on the search. The learned constraints can be minimized taking into account the semantics of atomic constraint [14], additional Boolean variables can be introduced [7], and explanations can be fine-tuned to improve the quality of learned constraints [33, 13].

Solvers that are specialized in specific constraint types can also deal with different forms of conflict analysis. Pseudo-Boolean solvers, which solve ILPs where all variables are Boolean, use conflict analysis [12, 17] to great effect. In this special case of ILPs, the rounding problem as described in Subsection 2.3 can be handled systematically. CutSat [18] and CutSat++ [6], which are CDCL-inspired solvers for general ILPs, restrict the search to side-step the rounding problem by only assigning variables to their lower- or upper-bound. Last, the IntSat [26, 27] solver accepts that conflict analysis can fail and does not always learn a new constraint.

There is an interesting comparison between IntSat and HaifaCSP. In HaifaCSP, constraints are combined to derive new constraints according to pre-specified rules. The rule for combining two linear constraints is identical to how IntSat operates. This means that, given a problem with only linear constraints, HaifaCSP is essentially an extension of IntSat. It is an extension, because, unlike IntSat, HaifaCSP will fall back to learning a clause if the derived linear constraint is not propagating.

To make effective use of specialized ILP solvers, much work focuses on translating arbitrary constraints into linear inequalities. The MiniZinc [25] toolchain can convert CP models to efficiently solvable ILPs [4]. These translations form a basis of the explanations we introduce in this paper. Much work has also been done to explain propagations in pseudo-Boolean equations [23], although the aim there is correctness in a proof, rather than propagation impact during search.

4 Our Contribution: Lazy Linear Generation (LLG)

We propose the Lazy Linear Generation (LLG) algorithm, an extension of LCG that learns asserting linear inequalities alongside clauses. A learned clause captures a propositional relationship between *atomic constraints*. LLG enhances clausal conflict analysis by incorporating cutting-planes analysis, allowing learned constraints to capture linear relationships between *variables* – something that cannot be expressed compactly using clauses.

Extending LCG to perform conflict analysis using linear inequalities aligns closely with the challenges addressed by IntSat [27] and HaifaCSP [40]. Similar to IntSat, LLG adapts the CDCL resolution algorithm to iteratively apply linear combinations to the explanations for conflicts or propagations, deriving asserting linear constraints that are added to the model. Additionally, LLG also employs resolution – specifically LCG – when the derived conflicting constraint does not conflict with the current assignment any longer.

The key distinction between our LLG approach and previous works lies in LLG's ability to linearly explain propagations and conflicts originating from arbitrary propagators, rather than being restricted solely to linear propagators. Our method leverages the advantages of both CP and a linear formulation of the problem, combining the propagation strength of CP propagators with more powerful linear conflict analysis. An example of a stronger CP propagator compared to its linear decomposition is shown in Example 4.

▶ **Example 4.** Consider the multiplication $a \cdot b = c$ constraint, with initial domains $a \in \{2,3\}, b \in \{1,4\}, c \in \{2,7\}$. A possible linear decomposition introduces auxiliary binary variables p_{a2}, p_{a3} and represents the constraint using the following constraints (normally expressed linearly using the big-M formulation):

$$p_{a2} + p_{a3} = 1 \quad \wedge \quad a = 2 \cdot p_{a2} + 3 \cdot p_{a3} \quad \wedge \quad p_{a2} \rightarrow 2b = c \quad \wedge \quad p_{a3} \rightarrow 3b = c$$

Under the current partial assignment, a CP propagator can infer, based on the upper bounds of a and c, that $b \leq 3$. However, since the auxiliary variables p_{a2} and p_{a3} remain unfixed, no further propagation can be achieved using the linear decomposition. The CP propagation can be explained using the expression $\langle a \geq 2 \rangle \wedge \langle c \geq 0 \rangle \rightarrow 2b \leq c$, which can be transformed into a linear inequality by introducing auxiliary variables for the conditions and employing the big-M formulation.

This new analysis, however, introduces additional challenges. First, it requires defining explanations for every propagation and conflict. Second, these explanations may require the dynamic introduction of auxiliary variables to accurately capture certain propagations

or conflicts. Finally, the learning procedure may generate new constraints that are not conflicting with the current assignment. This issue, which was previously caused only by rounding problems, can now also arise in cases where propagations or conflicts cannot be directly expressed as linear inequalities or when weak explanations are encountered.

The remainder of this section will outline LLG in more detail. Section 4.1 provides a concise overview of the algorithm, while Section 4.2 discusses methods for constructing explanations. Implementation details of the auxiliary variables are presented in Section 4.3, and Section 4.4 describes examples of linear explanations.

4.1 Conflict Analysis algorithm

The conflict analysis algorithm employed by LLG closely resembles the one presented in Algorithm 1, with several key modifications introduced in this section. The most notable distinction is that constraints are no longer exclusively linear. While the conflicting constraint CC and the reason constraint RC were previously guaranteed to be linear, conflicts and propagations must now be *explained* linearly for them to participate in conflict analysis. Unlike IntSat, there is no guarantee that CC or RC can actually be explained linearly, as certain linear explanations for propagations and conflicts may either be impractical or too expensive to create.

For example, we consciously do *not* convert clauses to linear inequalities, even though that is possible. Any clause $l_1 \vee \cdots \vee l_n$ with l_i being atomic constraints can be turned into the linear inequality $l_1 + \cdots + l_n \geq 1$. However, contrary to many other LCG solvers, our implementation does not create 0-1 variables for atomic constraints in clauses. I.e., a clause is a set of atomic constraints rather than a set of Boolean variables. This means that converting the clause to a linear constraint would require creating auxiliary variables for all atomic constraints l_i. As will become clearer in Subsection 4.3, creating auxiliary variables is not cheap, and we would need a lot of them. Future work can explore an implementation where this is feasible.

Subsection 4.4 presents a few linear explanations, and Appendix A gives the complete list of constraints and explanations we implemented. Given that, our conflict analysis differs from IntSat (as displayed in Algorithm 1) in the following areas:

1. First, rather than directly retrieving the currently conflicting constraint from $\mathcal{C}$ (line 1), we attempt to derive a linear explanation for the conflict. Either the conflict is explicitly identified by a propagator and described as a linear inequality, in which case CC is set to that linear inequality. Alternatively, the domain of a variable x became empty. In that case, the linear inequality $Ax \leq b$ explaining the last propagation on x is conflicting, so it is used as the conflicting constraint.
   If we cannot express the conflict with a linear constraint, then we proceed to the resolution fallback (line 14). A common example when this happens is when the conflicting constraint is a clause.
2. Similarly, instead of getting RC directly from the trail (line 4), RC becomes the linear explanation of the propagation. We also introduce an additional check: the combination step cannot proceed if we cannot create a linear explanation to assign to RC.
3. Finally, if we fail to learn an asserting linear constraint, we fall back to resolution just as IntSat does (line 14). The difference here is that IntSat does not remember clauses because it does not have a clausal propagator, so it just propagates the learned clause once and forgets about it. In our CP solver, we have a clausal propagator, so we remember the clause. Effectively, if we cannot learn an asserting linear inequality, the conflict analysis operates as a standard LCG conflict analysis does.

4.2 Linear explanations

A fundamental aspect of LLG is constructing linear inequality explanations for propagations. This section elaborates on several key aspects of constructing these explanations.

Inferred constraints. Firstly, it is important to highlight that the asserting inequality derived during conflict analysis is added to the model as a constraint. Consequently, all linear explanations that lead to the construction of this inequality must also be inequalities inferred from the model. This is the same requirement for a clausal explanation.

Explanation signs. Additionally, it is noteworthy that for a linear inequality to be an explanation, it must constrain the propagated variable in the same direction as the propagation being explained. The direction in which a linear constraint constrains a variable is determined by the sign of this variable in the inequality. More specifically, a linear constraint $5a - 4y \leq 0$ constrains the upper bound of a, and the lower bound of y. Example 5 provides an intuitive justification for this condition. From a technical perspective, this requirement arises directly from the properties of linear constraint elimination: to eliminate a variable by combining two inequalities, the variable must appear with opposite signs in both. If the propagated variable in the explanation has the same sign as in the conflicting constraint, it did not contribute to the conflict and is therefore not taken into account.

▶ **Example 5.** Consider a conflict analysis with current conflicting constraint $CC : -6x+3y \leq 10$. We aim to derive a CC that is asserting earlier in the search. This requires maximizing its left-hand side – achievable by minimizing x or maximizing y. Now, given the trail entry $(V, RC) = (x, -3x - 6y \leq 10)$, we aim to eliminate x. However, since x has a negative coefficient in RC, it constrains x's lower bound, not contributing to the goal of minimizing x.

Conditional explanations. An explanation may incorporate a conditional component to specify the conditions under which the explanation holds. Such conditional statements can be represented using Boolean *auxiliary variables*, which are binary variables indicating the truth value of a condition. For example, consider the conditional explanation $\langle a \leq 2 \rangle \rightarrow \langle b \geq 3 \rangle$. We can encode the condition by introducing a Boolean auxiliary variable p_1, defined as:

$$p_1 = \begin{cases} 1 & \text{if } a \leq 2 \\ 0 & \text{if } a > 2 \end{cases} \tag{1}$$

By ensuring consistent propagation of p_1 (discussed further in Section 4.3), the conditional explanation can be reformulated into a linear inequality: $b \geq 3 - M(1 - p_1)$. This formulation uses the "Big M" transformation to model the cases of $a \leq 2$ and $a > 2$ within a single explanation. Similar linearization techniques are described in more detail by [4].

4.3 Implementing auxiliary variables

The auxiliary variables associated with conditional explanations cannot be constructed before starting the search procedure, as it is unknown which auxiliary variables will be necessary. Thus, they must be introduced and propagated during the search. A key challenge is maintaining a consistent solver state at decision levels where these variables did not exist yet.

Adding auxiliary variables. When a new auxiliary variable is introduced, it often could have propagated at an earlier decision level, had it existed earlier. While we can propagate the auxiliary variable immediately upon its creation, this propagation is discarded upon backtracking. Ideally, we would retroactively introduce auxiliary variables at the start of the search, allowing them to propagate at the correct decision level. However, we argue that this retroactive introduction is unnecessary, as solver correctness does not require propagation at the earliest possible decision level.

Instead, we ensure that auxiliary variables are propagated as soon as possible. Upon backtracking, propagation normally starts by propagating the newly learned constraint, which may then trigger other propagators. LLG, however, prioritizes auxiliary variable propagation at this stage. This approach seeks to update the auxiliary variables – and, by extension, the variables that depend on them – so that they are identical to the state that would have been achieved had the auxiliary variables been present from the beginning of the search. If the solver backtracks further and discards this propagation, the auxiliary variable can again be re-propagated at the earlier level, ensuring correctness without complex trail manipulations. Example 6 illustrates this principle.

▶ **Example 6.** At decision level 10, an explanation introduces an auxiliary variable p, defined as $p \iff 6a + 7b > 3$. We can immediately propagate $\langle p \geq 1 \rangle$. Had p existed earlier, this propagation would have already occurred at level 3. If the solver now backtracks to decision level 5, the propagation $\langle p \geq 1 \rangle$ is discarded. We immediately re-propagate $\langle p \geq 1 \rangle$ at level 5 and subsequently reach the same state as if p had been propagated at decision level 3.

Evaluating auxiliary variables during conflict analysis. A critical aspect of LLG's conflict analysis algorithm is verifying whether a newly learned constraint is asserting at any earlier decision level. The learned constraint may, however, contain auxiliary variables that did not yet exist at a previous decision level. Even though these auxiliary variables are properly propagated once a backtrack is executed, they might not be propagated on the trail when conflict analysis considers them. This can result in incorrectly classifying the learned constraint as non-asserting.

To resolve this, the truth-value of an auxiliary variable at a particular decision level is always computed based on its definition, rather than looked up as for other variables. This allows us to infer the truth value of the auxiliary variable, regardless of whether it has been propagated.

4.4 Examples of explanations

We have formulated linear explanations for several arithmetic constraints and the element global constraint. This section provides a detailed exposition of some of these explanations. A comprehensive overview of all explanations is presented in Appendix A. These explanations can closely resemble a lazy decomposition, where linear inequalities are added to the model lazily. However, when we refer to inequalities as explanations, we explicitly mean that we do not add the inequalities to the model.

▶ **Example 7** (Explanation of $x_i \neq val$ for $Ax \neq b$)**.** The propagation of $x_i \neq val$ implies that either $\langle x_i < val \rangle$ or $\langle x_i > val \rangle$ must hold. This can be captured by introducing an auxiliary variable p defined by $p \iff Ax < b$, leading to two possible explanations: (1) $Ax < b + M(1 - p)$, (2) $Ax > b - Mp$.

Which explanation is chosen depends on whether the propagation of $x_i \neq val$ decreases the upper bound of x_i (explanation 1), increases its lower bound (explanation 2), or introduces a hole in its domain (either explanation is valid, in our implementation we pick explanation 1).

▶ **Example 8** (Explanation of $\langle b \geq 1\rangle \wedge \langle c \geq 0\rangle \rightarrow a \leq \lfloor ub(c)/lb(b)\rfloor$ for $a \times b = c$)**.** To express this propagation linearly, we fix b to its *current* lower bound. Assuming a lower bound of $b_{\min} = 5$, the explanation is given by $b \geq 5 \wedge c \geq 0 \rightarrow 5a \leq c$. We introduce auxiliary variables (1) $p_1 \iff b \geq 5$ and (2) $p_2 \iff c \geq 0$ to represent the conditions. This leads to the linear explanation:

$$5a \leq c + M(1 - p_1) + M(1 - p_2). \tag{2}$$

▶ **Example 9** (Explanation of $\neg c \rightarrow r \leq 0$ for $r \rightarrow c$)**.** A noteworthy constraint is the *half-reified constraint* [15], which defines the implication $r \rightarrow c$ for Boolean variable r and an arbitrary CP constraint c. This constraint ensures that c must hold whenever $r = 1$. If c is conflicting under the current assignment, we can propagate $\langle r \leq 0\rangle$. Let $Ax \leq b$ be the explanation from c for the conflict. Consequently, we incorporate r into the explanation to capture the relationship between r and the conflict:

$$Ax \leq b + M(1 - r). \tag{3}$$

5 Experiments

We implemented our LLG approach in Pumpkin [11]. The initial version of Pumpkin serves as the baseline LCG solver. In the experiments we use the number of conflicts as the main metric to allow us to draw conclusions that are independent of the runtime and the efficiency of the implementation. We believe this fairly shows the potential of our LLG approach. To further ensure that the results are due to the differences in the conflict analysis procedure, and not other factors, we only consider instances for which the branching strategies of LLG and LCG are fixed according to the provided strategy in the instances. We summarize the results:

1. Subsection 5.2: In 25% of the instances *where at least one linear constraint is learned* the number of conflicts is reduced by at least 60%. The median reduction is around 10%, and in the worst 25% of instances there is a slight increase in the number of conflicts.
2. Subsection 5.3: Learned inequalities indeed provide stronger reasoning than clauses.
3. Subsection 5.4: The success rate of LLG analysis is relatively low and decreases with time.
4. Subsection 5.5: Decomposing to linear constraints, to suit the cutting-planes reasoning as much as possible, encounters more conflicts than LLG when aggregated over the entire benchmark set.
5. Subsection 5.6: The presence of clausal propagation reduces the number of encountered conflicts for LLG, indicating it is useful to remember clauses when linear analysis fails.

5.1 Experimental setup

The instances we used to evaluate our LLG implementation are drawn from the MiniZinc Challenges 2008-2024 [36, 34] and the MiniZinc Benchmarks[1]. Models with unbounded or floating-point domains were excluded, as well as models without a specified search heuristic. Instances from the MiniZinc challenge which are not solved by at least one finite-domain solver are also excluded. For every model with more than 10 instances, we sampled 10 instances randomly.

[1] `https://github.com/MiniZinc/minizinc-benchmarks`

Each instance is decomposed into the following constraints: multiplication ($a \times b = c$), truncating division ($a/b = c$), absolute value ($a = |b|$), maximum ($max(A) = b$), not equals ($Ax \neq b$), linear less-than-or-equal-to ($Ax \leq b$), reified ($p \rightarrow constraint$), element ($A[idx] = b$) and clauses. The explanations for these constraints are given in Appendix A.

This results in a dataset of **952** instances based on **223** unique models. These instances result in roughly **50.5M** linear inequalities, **18.4M** reified constraints, **3.7M** not-equals constraints, **1.3M** multiplication constraints, **654k** element constraints, **417k** maximum constraints, only **4.5k** absolute and only **1.4k** division constraints.

All experiments were conducted on the DelftBlue [10] compute cluster. Each run used a single thread of an Intel Xeon E5-6248R 24C 3.0GHz and had access to 8GB of memory. The time limit was set to 1 hour per instance.

5.2 Reduction of conflicts

The first experiment shows how LLG impacts the number of conflicts encountered compared to the LCG baseline. We can break down the instances into three categories:

1. There are 328 instances that timed out or ran out of memory for LLG. This is compared to 289 in the LCG solver. That is an increase, however, the instances for which this happens are different between the two solvers. I.e., the LLG solver times out or errors on different instances than the LCG solver. In many of these instances, the LLG solver does learn linear constraints, but their impact is not big enough to prevent a time-out.
2. Then, there are 181 instances that are solved by the LLG solver, but not a single linear inequality was learned. In these instances, even though LLG has the overhead of cutting-planes reasoning, that overhead did not prevent the solver from terminating within the time-out.
3. The final 443 instances are solved within the time limit, and at least one linear inequality is learned. Figure 1 aggregates the ratio of the number of conflicts for LLG relative to LCG. The lower half of the distribution exhibits significant reductions, with a decrease in conflicts of up to **60%** for a quarter of the instances, even increasing to over **90%** when looking at the top 10% of instances. Then, the Q2 to Q3 quartile indicates that in roughly a quarter of the completed instances, the number of conflicts is reduced only slightly. For these instances, LLG learns very few linear constraints, or learns linear constraints that propagate similar bounds as learned clauses. Lastly, the upper 75% to 90% percentile range highlights cases where the number of conflicts increases marginally. This is due to some learned constraints being weaker compared to the clauses that would have been learned by the LCG solver.

These numbers show that, with the current state of LLG, not all instances benefit from its linear conflict analysis. However, when a linear inequality is derived, it has the potential to massively reduce the number of conflicts. We initially assumed that the top-performing instances would be derived from a limited subset of models that fit LLG especially well. However, upon examining the instances exhibiting at least a 50% reduction in conflicts, we identify 117 instances originating from 52 models ($117/52 \approx$ **2.3** instances per model). In comparison, a total of 574 instances from 171 models successfully complete for both LCG and LLG ($547/165 \approx$ **3.3** instances per model): the top-performing instances are even more varied than the full dataset. Consequently, our assumption was wrong; particular models are not more suitable for LLG than others.

The distribution of constraints across these instances also seems highly variable. The successful instances exhibit a slightly higher percentage of linear inequality constraints, though the difference is within only a few percent.

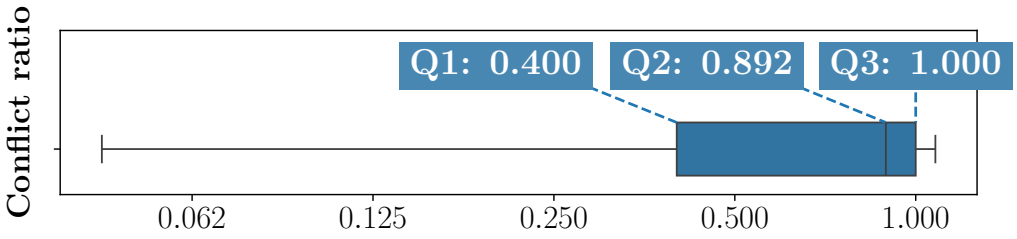


Figure 1 Boxplot showing the ratio of conflicts in LLG over LCG (lower = better). The boxplot displays the Q1, Q2 and Q3 quartiles, with whiskers extending to the 10% and 90% percentiles. A logarithmic scale is used for equal spacing of ratios. Only instances where both LCG and LLG successfully completed, and at least one linear constraint was learned, are considered.

Lastly, although the implementation of the LLG solver can be improved on many fronts, we observe that for the top 25% of instances – each achieving at least a 60% reduction in conflicts – runtime performance already surpasses our baseline LCG, owing to the significant decrease in conflicts of LLG. Specifically, for this subset of instances (excluding those with execution times below 5 seconds), the median runtime improvement is 75%.

5.3 Strength of learned inequalities

The motivation for LLG is based on the proposition that linear constraints may provide stronger reasoning than clauses. With this experiment, we show that the theoretical benefit translates to practice. For the 443 completed instances with at least one learned inequality, we examine whether the learned inequality indeed propagates more often than the clause that would have been learned in its place.

The results indicate that when a linear constraint is learned, it replicates over **91%** of the propagations that the fallback clause would have produced. In contrast, only **37%** of the propagations triggered by learned linear constraints would have been replicated by the clause. This confirms that, in nearly all cases, the linear constraint is at least as strong as the fallback clause. This further suggests that neither conflict analysis method strictly dominates the other, yet learned linear constraints tend to be more general in practice.

We also observe that in a normal LLG execution, the majority of learned clauses do not propagate more than once. In contrast, the median number of propagations per learned learned inequality is **90**. This demonstrates that when an inequality can be learned, it generally has a much greater impact on the search compared to a learned clause.

5.4 Analysis success rate

This experiment aims to illustrate the success rate of LLG analyses as the search progresses, and to identify reasons for failed analyses. Recall that a successful LLG analysis results in learning a linear constraint, whereas a failed analysis reverts to LCG for reasons such as a violated conflicting invariant. Here, it is relevant to compare successful instances with less successful instances. We divide the 624 instances that the LLG solver could finish into three subsets: all instances (Figure 2a), those with at least a 50% reduction in conflicts (Figure 2b), and those with at least a 75% reduction in conflicts (Figure 2c).

Firstly, Figure 2a demonstrates a declining trend in successful analyses over time, accompanied by an increase in conflicts and explanations that cannot be expressed linearly. This trend persists when results are categorized by the number of conflicts (comparing small and

large instances) and by problem type (satisfaction versus optimization). The only correlation we can observe is the one between a decreasing LLG success rate and an increase in failed LLG analyses attributed to encountered clauses. Approximately **38%** of LLG failures can be attributed to these factors. The remaining **62%** consist of failures that occur at a relatively constant rate throughout the search, including invariant violations ($\approx 50\%$), combinations that fully cancel out ($\approx 6\%$), overflows ($\approx 3\%$), and cases where a decision is reached before identifying an asserting constraint ($\approx 3\%$).

In contrast, Figures 2b and 2c exhibit some differences compared to the full dataset. Initially, conflict analysis is highly successful – significantly more so than for the full dataset. However, as the search progresses, analysis performance declines sharply. For these instances, we can see that the number of analysis failures due to invariant violations increases substantially throughout the search. Figures 2a through 2c all illustrate that LLG conflict analysis succeeds in only a relatively small fraction of instances. This experiment demonstrates that even a relatively small number of learned linear constraints can significantly decrease the number of conflicts encountered.

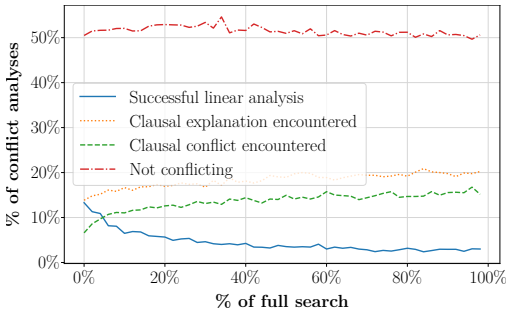
Given the decreasing frequency of conflict analyses leading to newly learned linear constraints, it is worth exploring whether a similar pattern is observed in the propagation of learned constraints. Figure 2d plots the density function (area under the curve equals 1) of all propagations triggered by learned constraints, presented for the same three subsets of instances. The results indicate that, despite the decline in newly learned linear constraints, propagations of learned constraints increase over time. This indicates that, as the search progresses, previously learned linear constraints continue to propagate consistently.

Explanation slacks. Finally, we examined the slack of explanations that resulted in a learned inequality compared to those that did not. In the context of LLG, slack is defined as $\text{SLACK}(Ax \leq b) = b - \text{LB}(Ax)$, where LB computes the current lower bound. Notably, a negative slack indicates a conflict.

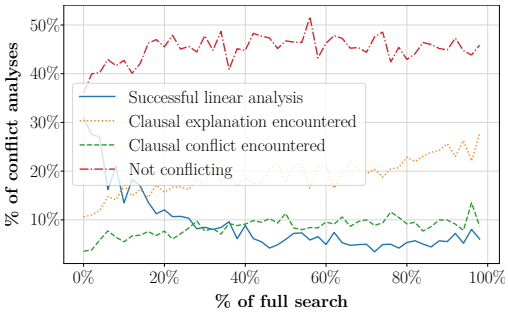
First, we found no clear correlation between the slack of a conflict explanation and the proportion of these explanations that eventually led to a learned constraint, contrary to our expectation that large negative slacks are more likely to succeed. The only necessary condition for a conflict explanation is that is has negative slack. Second, we observed a strong correlation between the slack of a propagation explanation and its likelihood of resulting in a learned constraint: explanations with lower slack demonstrate a higher probability of leading to a learned constraint. This result is sensible, as lower slack values indicate that the explanation is closer to being either asserting or conflicting. In contrast, explanations with high slack may be overly general or involve large big-M coefficients. These results suggest the possibility of selecting certain explanations based on their slack, prioritizing those that are more likely to contribute to the derivation of a learned constraint.

5.5 LLG compared to linear decomposition

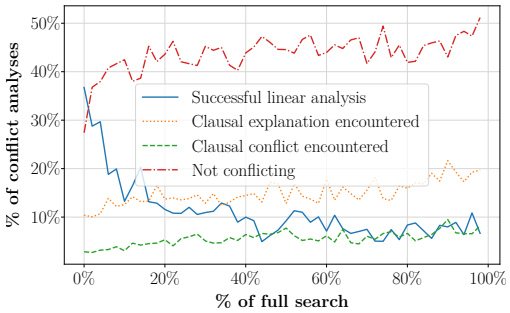
In addition to comparing LLG with LCG, it is also valuable to evaluate Pumpkin's decomposition as described in Section 5.1 against a decomposition consisting only of linear inequalities. This evaluation shows that using CP propagators – even though not all its linear explanations are equally successful – is still beneficial over a fully linear model. For this experiment, all instances from the test set were reformulated into a fully linear model using the MiniZinc Linear Library [4]. Among the 952 instances in the test set, 938 could be transformed into a linear model within a 300-second timeout. The 14 instances that could not be converted likely experienced excessive model growth when represented fully linearly.



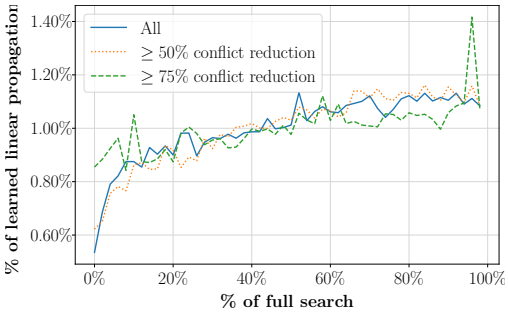
(a) All instances.



(b) Instances with at least 50% conflict reduction over LCG.



(c) Instances with at least 75% conflict reduction over LCG.

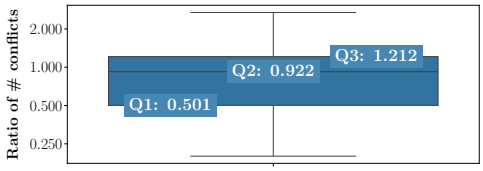


(d) Probability density function showing the distribution of propagations of learned inequalities (y-axis) as a function of search progress, measured by the percentage of total conflicts (x-axis).

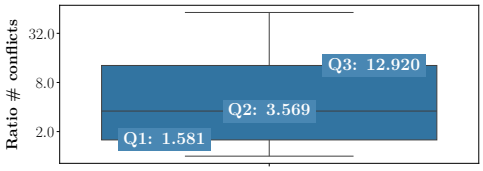
Figure 2 Figures (a) through (c) show the success rate of conflict analyses (y-axis), plotted against the percentage of total conflicts (x-axis). The four most significant outcomes are included. Figure (a) demonstrates a decrease in analysis success as more clauses are encountered. Figures (b) and (c) show a strong initial success, followed by an increase in invariant violations. Figure (d) demonstrates that, although the number of newly learned linear constraints decreases as the search progresses, the tail end exhibits the highest concentration of constraint propagations. This can be explained by the cumulative effect of both newly introduced and previously learned constraints.

For the 938 successfully converted instances, Figure 3a presents the ratio of conflicts between the linear decomposition (baseline) and LLG. The results indicate a substantial reduction in conflicts when using LLG. Specifically, at least 25% of the instances exhibit a conflict reduction of **50%** or more. Furthermore, the first and second quartiles (Q1 to Q2) of the boxplot show a notable decline in conflicts, ranging from **50%** to **8%**. The second to third quartiles (Q2 to Q3) display a mix of reductions, from an **8%** reduction to an increase of **20%**. The remaining instances exhibit a slightly larger increase in conflicts.

These increases in conflicts can be attributed to two primary factors. Firstly, the preprocessing optimizations performed by the linear decomposition can result in the generation of smaller linear problem instances compared to those produced by Pumpkin's decomposition. Secondly, some of Pumpkin's propagators perform propagations that cannot be easily linearly explained, such as *set* or *alldifferent* propagations, whereas the linear decomposition provides alternative propagations that can be linearly explained more effectively. Nevertheless, LLG continues to exhibit a significant performance advantage over the linear decomposition.



(a) Plot showing the conflict ratio between the linear decomposition (baseline) and Pumpkin's decomposition (lower = better). Pumpkin's decomposition encounters significantly fewer conflicts in over 50% of the instances. In 25% of instances, the linear decomposition performs either slightly worse or slightly better. In the remaining 25% of instances, the linear decomposition performs better than Pumpkin's decomposition.



(b) Evaluation of a variation of LLG where learned clauses are propagated but not added to the model. The results show a notable increase in conflicts (note the y-axis labels), highlighting the importance of clausal propagation for LLG.

Figure 3 Boxplots showing the ratio of conflicts in LLG relative to a variation on LLG. The boxplots display the Q1, Q2 and Q3 quartiles, with whiskers extending to the 10% and 90% percentiles. A logarithmic scale is used for equal spacing of ratios. Only instances where both LCG and the variation successfully completed are considered.

5.6 Forgetting learned clauses

Subsection 5.4 shows that as more clauses are encountered in the conflict analysis, the success-rate of the the linear conflict analysis drops. Therefore, we investigate the consequences of only propagating a learned clause once, without storing it in the constraint database. The results, presented in Figure 3b, indicate that omitting clause storage leads to a significant increase in conflicts. This observation underscores that even though encountering clauses likely results in learning fewer linear inequalities, the effectiveness of LLG relies heavily on the ability to store and propagate learned clauses.

We note the clear parallel between this experiment and the standard behavior of IntSat [27]. However, two key distinctions differentiating the two approaches prevent an accurate comparison. First, IntSat operates solely on fully linear models, which inherently provide better linear explanations for propagations than a CP model can, increasing the success rate of IntSat's conflict analysis. Second, IntSat tries to transform learned clauses into linear inequalities when at most one bound of the clause is non-binary. Within our CP problems, such conversions are rarely feasible due to the predominant use of integer variables. This limitation might change if the CP models were reformulated as linear programs.

6 Conclusion & Future Work

The effectiveness of constraint learning in Constraint Programming has been extensively studied, with much of the focus on learning clauses. However, linear constraints have the potential to provide more powerful reasoning capabilities. In response, we propose Lazy Linear Generation (LLG), a conflict analysis algorithm that incorporates concepts from CDCL [22], IntSat [27], HaifaCSP [40], and Lazy Clause Generation [38] to develop a novel learning mechanism for linear constraints. LLG generates explanations for propagations and conflicts for arbitrary CP propagators. To achieve this, the algorithm introduces new auxiliary variables while maintaining a consistent solver state.

Our experimental analysis of LLG shows that, on the one hand, learning linear constraints can lead to a substantial reduction in the number of conflicts over solely learning clauses. Although the success rate of linear conflict analysis is relatively low, when a linear constraint

is derived, the impact of that constraint on overall solver performance is significant. On the other hand, there are many instances where LLG fails to derive any new linear constraint, as the explanations end up being overly general. When compared to a linear decomposition with the same conflict analysis, we observe a reduction in the number of conflicts encountered when aggregating over all instances, suggesting that the strong CP propagation is useful and decomposing to linear constraints is not the preferred approach. Furthermore, our experiments highlight that clausal propagation is still important, even when using conflict analysis based on cutting-planes. To conclude, whilst there are still improvements to be obtained, we believe our approach shows potential to improve the constraint learning capabilities of CP solvers.

There are several promising avenues for further research of the LLG algorithm. First, it would be interesting to consider the impact on more global constraints. Second, it should be investigated whether we can increase the success rate of the cutting-planes conflict analysis. Additionally, we could improve our linear explanations based on decomposition optimisations [4]. Finally, the branching heuristic can likely utilize information from cutting-planes conflict analysis to make the search more dynamic. We believe these research directions offer valuable opportunities to further enhance the effectiveness and applicability of solvers.

References

1 Younes Aalian, Gilles Pesant, and Michel Gamache. Optimization of Short-Term Underground Mine Planning Using Constraint Programming. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2023.6.

2 Tobias Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, March 2007. MAG ID: 1990267895 S2ID: 2446332bd4193d61129622041298218e0f1c7676. doi:10.1016/j.disopt.2006.10.006.

3 Roberto Amadini, Mak Andrlon, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Constraint Programming for Dynamic Symbolic Execution of JavaScript. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 1–19, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-19212-9_1.

4 Gleb Belov, Peter J. Stuckey, Guido Tack, and Mark Wallace. Improved linearization of constraint programming models. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 49–65, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-44953-1_4.

5 A. L. Brearley, G. Mitra, and H. P. Williams. Analysis of mathematical programming problems prior to applying the simplex algorithm. *Mathematical Programming*, 8(1):54–83, December 1975. doi:10.1007/BF01580428.

6 Martin Bromberger, Thomas Sturm, and Christoph Weidenbach. A complete and terminating approach to linear integer solving. *Journal of Symbolic Computation*, 100:102–136, September 2020. MAG ID: 2966563153 S2ID: ded68accef3d475b5e17fa9c1ae9bdfb6bd5968c. doi:10.1016/j.jsc.2019.07.021.

7 Geoffrey Chu and Peter J. Stuckey. Structure Based Extended Resolution for Constraint Programming, June 2013. doi:10.48550/arXiv.1306.4418.

8 Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed: The chuffed cp solver. URL: https://github.com/chuffed/chuffed.

9 Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, January 1990. doi:10.1016/0004-3702(90)90046-3.

10 Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2, 2024.

11 Emir Demirović, Maarten Flippo, Imko Marijnissen, Konstantin Sidorov, and Smits Jeff. Pumpkin: A lazy clause generation constraint solver in rust, 2024. URL: `https://github.com/ConSol-Lab/Pumpkin`.

12 Donald Chai and Andreas Kuehlmann. A fast pseudo-Boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):305–317, February 2005. MAG ID: 1985602827 S2ID: 6ac45677642ce52faf49a15dc3f1cf3ffdaf504d. `doi:10.1109/tcad.2004.842808`.

13 Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining alldifferent. In *Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122*, ACSC '12, pages 115–124, AUS, January 2012. Australian Computer Society, Inc. URL: `http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV122Downing.html`.

14 Thibaut Feydy, Andreas Schutt, and Peter Stuckey. Semantic Learning for Lazy Clause Generation. In *TRICS workshop, held alongside CP*. Citeseer, 2013.

15 Thibaut Feydy, Zoltan Somogyi, and Peter J. Stuckey. Half Reification and Flattening. In Jimmy Lee, editor, *Principles and Practice of Constraint Programming – CP 2011*, volume 6876, pages 286–301. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. `doi:10.1007/978-3-642-23786-7_23`.

16 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024. URL: `https://www.gurobi.com`.

17 Jan Elffers and Jakob Nordström. Divide and Conquer: Towards Faster Pseudo-Boolean Solving. *International Joint Conference on Artificial Intelligence*, pages 1291–1299, July 2018. MAG ID: 2808706008 S2ID: fe4efc3fab6a3a5b01a58f9434bd86c1587fe1d3. `doi:10.24963/ijcai.2018/180`.

18 Dejan Jovanović and Leonardo de Moura. Cutting to the Chase. *Journal of Automated Reasoning*, 51(1):79–108, June 2013. `doi:10.1007/s10817-013-9281-x`.

19 Dejan Jovanović and Leonardo de Moura. Cutting to the Chase Solving Linear Integer Arithmetic. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 338–353, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

20 George Katsirelos. Generalized NoGoods in CSPs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 5, pages 390–396, 2005.

21 George Katsirelos. *Nogood Processing in CSPs*. PhD thesis, University of Toronto, January 2009.

22 João Marques-Silva and Karem A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. MAG ID: 2044560939 S2ID: 38be6e613f2c30d21bffe8b468bc0cd46edba0d0. `doi:10.1109/12.769433`.

23 Matthew McIlree and Ciaran McCreesh. Certifying Bounds Propagation for Integer Multiplication Constraints. In *39th Annual AAAI Conference on Artificial Intelligence (AAAI'25)*, Philadelphia, 2025.

24 Neil C. A. Moore. *Improving the Efficiency of Learning CSP Solvers*. Thesis, University of St Andrews, May 2011.

25 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer. `doi:10.1007/978-3-540-74970-7_38`.

26 Robert Nieuwenhuis. The IntSat Method for Integer Linear Programming. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, volume 8656, pages 574–589. Springer International Publishing, Cham, 2014. Series Title: Lecture Notes in Computer Science. `doi:10.1007/978-3-319-10428-7_42`.

27 Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. IntSat: integer linear programming by conflict-driven constraint learning. *Optim. Methods Softw.*, pages 1–28, September 2023. ARXIV_ID: 2402.15522 MAG ID: 4387099028 S2ID: 3843288a76ab7ad11e0d0f664b35c5fd885acc94. `doi:10.1080/10556788.2023.2246167`.

28 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints - An International Journal*, 14(3):357–391, September 2009. MAG ID: 2059035667 S2ID: 1e00201f51a2f2161c8d7ae0a7843774955659cf. doi:10.1007/s10601-008-9064-x.

29 Peter J. Stuckey Olga Ohrimenko and Michael Codish. Propagation = lazy clause generation. *International Conference on Principles and Practice of Constraint Programming*, 4741:544–558, September 2007. MAG ID: 1546943373 S2ID: 73d784cd29c977ee83874229bcf29a0b143922bb. doi:10.1007/978-3-540-74970-7_39.

30 Laurent Perron and Frédéric Didier. Cp-sat. URL: https://developers.google.com/optimization/cp/cp_solver/.

31 Olivier Ponsini, Claude Michel, and Michel Rueher. Verifying floating-point programs with constraint programming and abstract interpretation techniques. *Automated Software Engineering*, 23(2):191–217, June 2016. doi:10.1007/s10515-014-0154-2.

32 Stéphanie Roussel, Thomas Polacsek, and Anouck Chan. Assembly Line Preliminary Design Optimization for an Aircraft. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2023.32.

33 Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Explaining the cumulative propagator. *Constraints*, 16(3):250–282, July 2011. doi:10.1007/s10601-010-9103-2.

34 Peter J. Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, July 2010. doi:10.1007/s10601-010-9093-0.

35 Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. MiniZinc Challenge 2024 results. URL: https://www.minizinc.org/challenge/2024/results/.

36 Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The MiniZinc Challenge 2008–2013. *AI Magazine*, 35(2):55–60, June 2014. doi:10.1609/aimag.v35i2.2539.

37 Pierre Talbot, Tingting Hu, and Nicolas Navet. Constraint Programming with External Worst-Case Traversal Time Analysis. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2023.34.

38 Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. *International Conference on Principles and Practice of Constraint Programming*, 5732:352–366, September 2009. MAG ID: 2114231362 S2ID: bc1b44d4e041280edc0e26fd55c630e69f4e8163. doi:10.1007/978-3-642-04244-7_29.

39 P. Van Beek, Francesca Rossi, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

40 Michael Veksler and Ofer Strichman. Learning general constraints in CSP. *Artificial Intelligence*, 238:135–153, September 2016. doi:10.1016/j.artint.2016.06.002.

41 Sameela Suharshani Wijesundara, Maria Garcia de la Banda, and Guido Tack. Addressing Problem Drift in UNHCR Fund Allocation. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:18, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2023.37.

A List of explanations

Table 1 contains a comprehensive overview of all explanations used for LLG. In case several explanations are possible for a propagation, the right one is chosen based on whether a lower or upper bound is propagated. The explanations have undergone multiple manual reviews but have not been formally verified for correctness.

Table 1 Table containing all explanations used for LLG.

Constraint	Propagation / conflict	Conditions	Explanation	Auxiliaries
$Ax \leq b$	$x_i \leq v$	$ub(x_i) > (b - lb(Ax) + lb(x_i))$	$Ax \leq b$	-
	Conflict	$lb(Ax) \geq b$	$Ax \leq b$	
$Ax \neq b$	$x_i \neq v$	x_i is only unfixed var in x	• $Ax \leq b - 1 + M(1-p)$ • $Ax \geq b + 1 - Mp$	$p \iff Ax < b$
	Conflict	$lb(Ax) = ub(Ax) = b$	$Ax \geq b + 1 - Mp$	
$a = \lvert b \rvert$	$a \geq 0$	-	$a \geq 0$	
	$a \geq lb(b)$	$lb(b) > 0$	$a \geq b$	
	$b \geq -ub(a)$	-		
	$a \geq \lvert ub(b) \rvert$	$ub(b) < 0$	$a \geq -b$	• $p_1 \iff b \leq 0$ • $p_2 \iff b \geq 0$
	$b \leq ub(a)$	-		
	$a \leq \max(\lvert lb(b) \rvert, \lvert ub(b) \rvert)$	-		
	$b \leq -lb(a)$	$ub(b) \leq 0$	• $a + b \leq M(1-p_1)$ • $a - b \leq M(1-p_2)$	
	$b \geq lb(a)$	$lb(b) \geq 0$		
$max(A) = b$	$a_i \leq ub(b)$	-	$a_i \leq b$	
	$b \geq \max_{a_i \in A}(lb(a_i))$	-		• $\forall i \in \lvert A \rvert$: auxiliary p_i • $\sum p_i = 1$
	$b \leq \max_{a_i \in A}(ub(a_i))$	-		
	$a_i \geq lb(b)$	a_i only for $ub(a_i) \geq lb(b)$	$rhs \leq a_i + M(1-p_i)$	
	$b \leq ub(a_i)$	$p_i = 1$		
	$a_i \geq lb(b)$	$p_i = 1$		
$a \times b = c$	Sign propagations, for instance: $c \geq 0$	$lb(a) \geq 0 \wedge lb(b) \geq 0$	$c \geq 0 - Mn_a - Mn_b$	• $n_a \iff a < 0$ • $n_b \iff b < 0$ • $n_c \iff c < 0$ • $b_{lb} \iff b \geq lb(b)$ • $b_{ub} \iff b \leq ub(b)$
	a/b upper bound propagations, for instance: $a \leq \lfloor ub(c)/lb(b) \rfloor$	$lb(b) \geq 1 \wedge lb(c) \geq 0 \wedge ub(c) \geq 1$	$lb(b) \cdot a \leq c + M(1 - b_{lb}) + Mn_c$	
	$c \geq lb(a) \times lb(b)$	$lb(a) \geq 0 \wedge lb(b) \geq 0$		
	a/b lower bound propagations, for instance: $a \geq \lceil lb(c)/ub(b) \rceil$	$lb(b) \geq 0 \wedge ub(b) \geq 1 \wedge lb(c) \geq 1$	$ub(b) \cdot a \geq c + M(1 - b_{ub}) - Mn_b - Mn_c$	
	$c \leq ub(a) \times ub(b)$	$lb(a) \geq 0 \wedge lb(b) \geq 0$		
$num \mathbin{//} den = rhs$	Sign propagations, for instance: $rhs \geq 0$	$lb(num) \geq 0 \wedge lb(den) \geq 0$	$rhs \geq 0 - Mn_{num} - Mn_{den}$	• $n_{num} \iff num < 0$ • $n_{den} \iff den < 0$ • $p_{den} \iff den > 0$ • $rhs_{lb} \iff rhs \geq lb(rhs)$ • $den_{ub} \iff den \leq ub(den)$
	$rhs \leq ub(num)/lb(den)$	$ub(num) \geq 0 \wedge ub(den) \geq 0$		
	$den \leq ub(num)/lb(rhs)$	$lb(rhs) \geq 0 \wedge ub(num) \geq 0$	$lb(rhs) \cdot den \leq num + M(1 - rhs_{lb}) + Mn_{num}$ Swap rhs and den for alternative	
	$num \geq lb(rhs) \cdot lb(den)$	$lb(rhs) \geq 0 \wedge ub(den) \geq 0$		
	$rhs \geq \lfloor lb(num)/ub(den) \rfloor$	$lb(num) \geq 0 \wedge ub(den) \geq 0$	$ub(den) \cdot rhs + den - 1 \geq num - M(1 - den_{ub}) - Mn_{num} - M(1 - p_{den})$	
	$num \leq (ub(rhs) + 1) \cdot ub(den) - 1$	$ub(den) \geq 0 \wedge ub(rhs) \geq 0$	Swap rhs and den for alternative	
$A[idx] = b$	$b \leq \max_{a_i \in A if idx}(ub(a_i))$	-	$b \leq a_i + M(1-p_i)$	• $\forall i \in \lvert A \rvert$: auxiliary p_i • $\sum p_i = 1$ • $\sum i \cdot p_i = idx$
	$a_i \leq ub(b)$	$idx = i$		
	$b \geq \min_{a_i \in A if idx}(lb(a_i))$	-	$b \geq a_i - M(1-p_i)$	
	$a_i \geq lb(b)$	$idx = i$		
	$p_i \geq 1$	$idx = i$	$\sum i \cdot p_i = idx$	
	$p_i \leq 0$	$idx \neq i$		
$p \rightarrow cond$	$p \leq 0$	$\neg cond$	$expl(cond) \rightarrow Ax \leq b$	-
	Conflict	$p \geq 1 \wedge \neg cond$	$Ax \leq b + M(1-p)$	