Conflict Analysis Based on Cutting Planes for Constraint Programming

Robbin Baauw

Delft University of Technology

Conflict Analysis Based on Cutting Planes for Constraint Programming

by

Robbin Baauw

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday May 1st, 2025 at 10:00 AM.

Student number: 4923138
Project duration: September 2, 2024 – May 1, 2025
Thesis committee: Dr. E. Demirović, Supervisor
Dr. A. Costea,
Ir. M. Flippo, Daily supervisor

TU Delft

Preface

This thesis marks the final step in my journey toward obtaining my Master's degree in Computer Science at the Delft University of Technology. Over the past eights months, I have had the pleasure of working on this project, culminating in a submission to the CP2025 conference. Throughout this process, I have been fortunate to receive the guidance and support of several individuals, whom I would like to thank here.

First and foremost, I want to thank Emir Demirović for his invaluable wisdom and insights. It was always exciting to receive one of his lengthy emails, consisting of a plethora of points that I should consider. I am especially grateful for his encouragement to pursue this more theoretical project over the practical alternatives I initially considered.

I would also like to thank Maarten Flippo for patiently listening to me week after week, where in many cases, I struggled to articulate my thoughts clearly. The critical insights gave me something to think about every week; just when I thought I had everything figured out, our discussions proved otherwise

Lastly, I would like to thank Paula for the constant support—even though the project was not exactly up your alley. I hope your coffee machine will recover from all the overtime it put in this year.

Robbin Baauw
Delft, April 2025

Abstract

This thesis introduces Lazy Linear Generation (LLG), a novel conflict analysis and learning framework for Constraint Programming (CP) that incorporates cutting planes reasoning. By leveraging cutting planes, our approach learns potentially stronger linear constraints than traditional clausal learning, allowing for more pruning of the search space while benefiting from strong CP propagation.

LLG generates linear explanations for both propagations and conflicts, which are then used in linear inequality-based conflict analysis. When cutting planes reasoning fails to derive a linear constraint, we employ Lazy Clause Generation (LCG) as a fallback mechanism. We present linear explanations for various arithmetic constraints, including less-than-or-equal, not-equal, absolute value, maximum, integer multiplication, truncating division, element, and reified constraints. Additionally, we introduce techniques for dynamically generating auxiliary Boolean variables to encode conditions within linear explanations.

We evaluate an LLG prototype on 952 benchmark instances, demonstrating a median conflict reduction of 10%, increasing to 60% for the 25th percentile. Our results confirm that linear inequalities enable stronger reasoning than clauses and show that integrating CP propagators with linear conflict analysis outperforms employing a fully linear model. Furthermore, we show that a clausal fallback mechanism is crucial when linear analysis fails.

While further research and engineering efforts are required for full integration into CP solvers, our findings underscore the potential of linear learning in CP, paving the way for more effective conflict analysis and solving.

Contents

1

Introduction

Nowadays, optimization problems are ubiquitous, arising in a wide range of application domains including scheduling [29, 32, 28, 6, 37, 51], network design [34, 30, 27] and circuit verification [9, 3]. Various solver technologies have been developed to solve these problems, such as (Mixed) (Integer) Linear Programming ((M)(I)LP), Boolean satisfiability (SAT) and Constraint Programming (CP). Each of these approaches has its own advantages and disadvantages. This thesis focuses on Constraint Programming, a powerful combinatorial optimization technique for solving the Constraint Satisfaction Problem (CSP). CP is a widely used paradigm in domains such as scheduling [11, 48, 51, 26], resource allocation [25, 18, 45, 52] and verification [4, 41].

CP solvers explore the search space defined by the model's variables to find solutions to the problem stated by the model. Given the vastness of the search space, fully exploring it is often not feasible. Therefore, it is crucial for CP solvers to identify when the current state will not lead to a solution and backtrack accordingly. The two key methods employed by CP solvers to achieve this are: (1) *inference*, or *propagation*, which identifies infeasible child trees based on the solver's current state and model constraints, and (2) *conflict analysis*, which helps prevent revisiting infeasible sub-trees by learning new constraints during the search. This thesis specifically focuses on conflict analysis.

Significant advancements have been made in conflict analysis, both in CP and in related fields. For example, the Boolean satisfiability problem (SAT) can be efficiently solved using the Conflict-Driven Clause-Learning (CDCL) algorithm [33]. The success of conflict analysis and learning in SAT has inspired conflict learning in CP, leading to various advancements over the last few decades, such as learning (generalized) nogoods [15, 20], Signed Clause Learning [49], General Constraint Learning [31], and Lazy Clause Generation (LCG) [38, 39, 47, 46]. LCG remains a cornerstone of modern solvers, such as OR-Tools [40] and Chuffed [12].

Some of these CP conflict analysis techniques are based on the principles of CDCL, wherein a conflicting clause is resolved iteratively with the reason clause that triggered a propagation. Similarly, techniques in (Mixed) Integer Linear Programming (ILP), such as CutSat [24, 10], IntSat [36], and Achterberg's MIP conflict analysis [2] algorithm, also take inspiration from CDCL. Pseudo-Boolean solving, which involves linear inequalities consisting of binary variables, also greatly benefits from CDCL-based conflict analysis techniques [17, 22, 23]. All these techniques focus on learning linear inequalities using cutting-plane reasoning, while the current state-of-the-art in CP, LCG, is limited to clausal resolution. Since linear inequalities potentially offer stronger reasoning than clauses, learning linear inequalities in CP could enhance the solver's performance.

This work builds on two prior approaches. Both of these approaches incorporate techniques similar to CDCL: they iteratively combine constraints to construct new learned constraints. The first approach, IntSat [36], a conflict analysis algorithm for ILP, combines linear constraints to infer new constraints, which can be added to the constraint database. If this combination fails, IntSat falls back to clausal resolution and attempts to convert the learned clause into a linear constraint. Their experimental results demonstrate competitiveness with state-of-the-art MIP solvers, despite a relatively compact

implementation. A similar approach is proposed by HaifaCSP [31], a conflict analysis algorithm for CP, where a set of combination rules (including linear combinations) are used to infer learned constraints from existing CP constraints. When no combination rule is applicable, the algorithm falls back to clausal resolution. However, its applicability is limited as it can only combine constraints that are specified by the predefined combination rules, and cannot handle combining arbitrary constraints

This gives rise to the question: **"How can we incorporate cutting-planes analysis in CP for arbitrary propagators?"**. This thesis introduces a novel technique called **Lazy Linear Generation** (LLG). LLG explains arbitrary propagations and conflicts in CP using integer linear inequalities, on which cutting-planes analysis is performed. We adapt the conflict analysis algorithm from IntSat for this purpose, defining linear explanations for various CP propagators, such as the integer multiplication, truncating division, absolute-value, maximum, element, linear not equals, linear less or equals and reified propagators. We can derive new learned linear constraints using these explanations.

We present an extensive experimental evaluation, demonstrating that LLG reduces conflicts by at least 60% in a quarter of the instances within a representative dataset. Our results indicate that the propagation strength of the learned linear constraints is significantly stronger than that of learned clauses. We also show that LLG, using CP propagators, outperforms the linear formulation of the same problems. Lastly, we highlight the need for a clausal fallback in case a linear inequality cannot be learned.

1.1. Research aim

In addition to answering the main research question, we aim to answer several sub-questions:

- How can arbitrary CP propagations be explained using linear constraints? How can existing conflict analysis algorithms be adapted to integrate these linear explanations effectively?
- What is the effect of linear learning on the search in terms of the number of encountered conflicts?
- How do learned linear constraints compare to learned clauses in terms of propagation strength?
- Does learning clauses alongside linear inequalities affect the linear learning capabilities?
- How important is learning clauses in case linear learning fails?
- What is the advantage of linear learning in CP over solving the linear decomposition of the problem?

1.2. Structure

This thesis first presents the necessary background information in Section 2. It then explores related algorithms and existing approaches in Section 3. The contribution, Lazy Linear Generation, is detailed in Section 4, followed by an experimental analysis in Section 5. Finally, the thesis concludes with a summary of findings and a discussion of future research directions in Section 6.

This thesis is an extended version of a conference paper that is under review at the time of writing. For completeness, the preprint version of the conference paper is included in Appendix B.

2

Background

This section explores various optimization principles and the connections between them. It starts with an introduction to Constraint Programming (CP), the cornerstone of optimization, in Section 2.1. Next, Boolean satisfiability (SAT) and its conflict analysis algorithm are examined in Section 2.2. Lastly, key prerequisites for Integer Linear Programming (ILP) are presented in Section 2.3.

2.1. Constraint Programming

Constraint Programming (CP) is a well-established paradigm for solving combinatorial problems. This section first provides a formal definition of the Constraint Satisfaction Problem (CSP), which CP solvers aim to solve, and then outlines the operational principles of CP solvers.

2.1.1. Constraint Satisfaction Problem

A **Constraint Satisfaction Problem (CSP)** $\mathcal{P}$ is formally defined as a tuple $(\mathcal{X}, \mathcal{D}, C)$, where:

- Each $x_i \in \mathcal{X}$ represents an *integer decision variable*,
- $\mathcal{D}(x_i) \in \mathcal{D}$ specifies the *domain* of $x_i \in \mathcal{X}$ defining the set of permissible integer values for x_i, and
- Each $C(X) \in C$ is a *constraint* that restricts the solution space over a subset $X \subseteq \mathcal{X}$ of variables. Formally, a constraint is defined as a subset of the Cartesian product of the domains of the involved variables, $C(X) \subseteq \mathcal{D}(x_1) \times \mathcal{D}(x_2) \times \cdots \times \mathcal{D}(x_n)$.

An *assignment* $\theta(X)$ is a total mapping between variables $x_i \in \mathcal{X}$ to a set of elements V within their respective domains, i.e. $V \subseteq \mathcal{D}(x_i)$. If any variable is still assigned multiple values, i.e., $\exists x_i \in \mathcal{X} : |\theta(x_i)| > 1$, the assignment is referred to as a *partial assignment*. A *solution* to a CSP is an assignment θ_s that satisfies all constraints $C(X) \in C$ for every subset $X \subseteq \mathcal{X}$:

$$\forall x_i \in \mathcal{X} : |\theta_s(x_i)| = 1 \quad \wedge \quad \forall C(X) \in C : \theta_s(X) \in C(X)$$

Example 2.1.1 *This example illustrates how a Sudoku puzzle can be formulated as a CSP. The CSP representation is constructed as follows:*

1. ***Variable definition**: The Sudoku grid consists of a 9×9 arrangement of cells, each represented as a variable: $\mathcal{X} = \{x_{(1,1)}, \ldots, x_{(9,9)}\}$*
2. ***Initial domains**: Each variable has an initial domain of possible values: $\forall x_i \in \mathcal{X} : \mathcal{D}(x_i) = 1 \ldots 9$. The problem specification generally also contains several fixed cells, which can be incorporated in the initial domains of these cells, e.g. $\mathcal{D}(x_{(2,1)}) = \{3\}$.*
3. ***Constraints**: We must ensure that each row, column, and 3×3 subgrid contains distinct values. This is*

enforced using the $\texttt{alldifferent}(X)$ *constraint, ensuring that all variables in* X *assume unique values:*

$$\begin{aligned} C = \{&\texttt{alldifferent}(x_{(1,1)}, \ldots, x_{(9,1)}),\\ &\texttt{alldifferent}(x_{(1,1)}, \ldots, x_{(1,9)}),\\ &\texttt{alldifferent}(x_{(1,1)}, \ldots, x_{(3,3)}),\\ &\ldots\} \end{aligned}$$

2.1.2. Solving a CSP

Constraint Programming (CP) is a method for solving CSPs by exploring the search space defined by the Cartesian product of the variable domains: $\mathcal{D}(x_1) \times \cdots \times \mathcal{D}(x_n)$. At each step in the search, *inference* is applied to eliminate infeasible domain values based on the constraints. The remaining search space is then divided into subproblems, making a decision on variable domains and subsequently continuing the search on a lower level in the search tree. At any point within the search tree, the *decision level* is defined as the number of decisions that have been made to reach the current subproblem.

Inference is performed using propagators, which enforce constraints by eliminating values from domains. A propagator is a function $p : \mathcal{D} \to \mathcal{D}$ that prunes (*propagates*) values that cannot be part of a valid solution. Importantly, propagators only remove values, ensuring $p(\mathcal{D}(x_i)) \subseteq \mathcal{D}(x_i)$. If, at any stage, a variable's domain becomes empty, i.e. $\exists x_i \in \mathcal{X} : \mathcal{D}(x_i) = \emptyset$, we denote that a *conflict* has occurred, indicating that the current sub-tree cannot yield a feasible solution. Algorithm 1 presents pseudocode for the full CP algorithm, inspired by the notation in [5, Algorithm 2.1].

Algorithm 1 *solve*($\mathcal{P}$) - Depth-first search CSP solver

Input: A CSP $\mathcal{P} = (C, \mathcal{X}, \mathcal{D})$
Output: *true* if $\mathcal{P}$ has any solution, *false* otherwise

```
1:  P' ← propagate(P)
2:  if any domain in D' is empty then return false end if        ▷ Conflict, backtrack
3:  if any domain in D' contains more than one value then
4:      P'_1, ..., P'_k ← subproblems(P')                        ▷ Divide into subproblems
5:      for i = 1, ..., k do
6:          if solve(P'_i) = true then return true end if        ▷ Check if any subproblem contains a solution
7:      end for
8:      return false
9:  end if
10: return true
```

Generating subproblems A CSP solver generates subproblems by making a *decision* on the domain of a *decision variable*. Various value-selection strategies exist, such as generating a subproblem for every possible domain value or splitting the domain of a variable into two subproblems. Similarly, numerous variable-selection strategies exist, such as the *first fail* strategy [21], which picks the variable with the smallest domain. Activity-based heuristics similar to VSIDS [35] are also commonly found in CP solvers, prioritizing frequently encountered variables.

Propagator scheduling The computational complexity of a propagator depends on multiple factors, such as the number of variables involved in a constraint and the level of consistency enforced by the propagator. Problem formulations typically involve both low-cost propagators, characterized by a limited number of variables and simple constraints, and high-cost propagators, which involve numerous variables and more complex conditions. Since the execution of one propagator may trigger the execution of others, a proper scheduling strategy is essential. A simple yet effective scheduling approach prioritizes low-cost propagators, ensuring that all inexpensive domain updates are completed before executing higher-cost propagators.

2.2. Boolean Satisfiability

The Boolean Satisfiability (SAT) problem is the problem of finding a satisfactory assignment of Boolean variables for a set of clauses. It can be viewed as a restricted form of a CSP, where the set of variables is defined as $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, with domains constrained to binary values $\forall x_i \in \mathcal{X} : \mathcal{D}(x_i) = \{0, 1\}$. For each variable $x_i \in \mathcal{X}$, we define its corresponding *literals* l_i and $\neg l_i$ representing x_i or its negation $1 - x_i$. The constraints in a SAT problem are expressed as *clauses* which are disjunctions of literals, such as $l_1 \vee \cdots \vee l_n$. The goal of SAT solving is to find an assignment of variables that satisfies the conjunction of all clauses.

Key to solving a SAT problem is the propagation of literals, called *unit propagation*. For every clause $C : l_1 \vee \cdots \vee l_n$, we examine the number of fixed literals. *Unit clauses* (or *asserting clauses*) are clauses for which all literals except one, say l_n, have been assigned to false and the clause is not yet satisfied, i.e. $\forall l_i \in C \setminus \{l_n\} : \theta(l_i) = 0$. In this case, the remaining unassigned literal must be set to true to satisfy the clause: $l_n \rightarrow 1$. Similar to CP, conflicts may arise during propagation. If a conflict is detected, backtracking is performed. These operations form the foundation of the DPLL algorithm [14], which serves as the basis for modern SAT solvers.

2.2.1. CDCL

Resolving conflicts solely through backtracking is a sound approach; however, it contains a major limitation. Since DPLL only undoes the previous decision and therefore backtracks a single decision level, it may encounter the same conflict repeatedly, leading to redundant computations. To address these inefficiencies, Conflict-Driven Clause Learning (CDCL) [33] was introduced. When a conflict occurs, CDCL analyzes its underlying cause and derives an asserting clause that represents the partial assignment that lead to the conflict. This clause is then incorporated into the model, ensuring that the same partial assignment does not reoccur. The solver can also backtrack to the decision level at which the clause became unit or unsatisfied, rather than merely the most recent one. This *non-chronological backtracking* allows for skipping irrelevant decision levels and directly addressing the root cause of the conflict.

Iterative resolution. Given two clauses $(l_i \vee Y)$ and $(\neg l_i \vee Z)$, we can apply **resolution** on l_i to obtain a new learned clause $(Y \vee Z)$:

$$\frac{l_i \vee Y, \quad \neg l_i \vee Z}{Y \vee Z} \tag{2.1}$$

We can use resolution to construct learned asserting clauses. First, we initialize conflict analysis by taking the conflicting clause C_{conf}, containing literal l_i. This clause became conflicting because the previous propagation of clause C_{reason} set the inverse literal $\neg l_i$ to true. We subsequently iterate over the previous propagations and apply resolution until we reach the previous decision:

$$C_{\text{conf}} \leftarrow (C_{\text{conf}} \setminus \{l_i\}) \vee (C_{\text{reason}} \setminus \{\neg l_i\}) \tag{2.2}$$

Finally, we add the obtained C_{conf} to the model, and backtrack to the highest decision level of all literals in C_{conf}.

2.2.2. 1UIP

A straightforward approach to clause learning is to apply resolution until the previous decision is reached. This can, however, lead to large, non-general learned clauses. To address this, GRASP [33] introduced the concept of *Unit Implication Points* (UIPs). If a clause contains exactly *one* literal from the current decision level, we refer to this literal as the UIP.

When a UIP is encountered, backtracking can already occur before the previous decision is encountered. Specifically, we backtrack to the highest decision level of the variables in the clause, excluding the UIP itself. After backtracking, we can immediately propagate the UIP, as that is the only unfixed variable. Chaff [35] demonstrated that the most effective strategy is to stop at the *first* UIP, a method known as *1UIP*.

In practice, this means that we apply resolution iteratively until the conflict clause C_{conf} consists of *at most* one literal $l_{top} \in C_{\text{conf}}$ such that its negation $\neg l_{top}$ corresponds to either the previous decision or to a variable propagated after the previous decision. To put it differently, l_{top} is the only literal in C_{conf} that is assigned at the current decision level. Subsequently, all assignments are discarded until the assignment of a literal $l \in C_{\text{conf}} \neq l_{top}$ is reached, or until we have discarded all decisions. Finally, we add C_{conf} to the clause database, enabling the immediate propagation of $l_{top} \rightarrow 1$. Algorithm 2 shows the complete pseudocode of CDCL's algorithm with 1UIP, for which an example execution is shown in Example 2.2.1.

Algorithm 2 CDCL

Input: a set of clauses C and a trail T containing tuples (l, C_{reason}) representing a propagation of literal l by clause C_{reason}
Output: a learned clause and corresponding backtrack level

1: $C_{\text{conf}} \leftarrow$ currently conflicting clause in C
2: **while** there is more than one literal in C_{conf} that was assigned in the current decision level **do**
3: $(l, C_{\text{reason}}) \leftarrow$ PopTrail() ▷ Remove the previous trail entry.
4: **if** $\neg l \notin C_{\text{conf}}$ **then continue end if** ▷ Literal propagation not relevant.
5: $C_{\text{conf}} \leftarrow (C_{\text{conf}} \setminus \{\neg l\}) \vee (C_{\text{reason}} \setminus \{l\})$ ▷ Perform resolution.
6: **end while**
7: $l_{top} \leftarrow$ the only literal in C_{conf} that was assigned in the current decision level
8: **return** C_{conf}, DECISION-LEVEL$(C_{\text{conf}} \setminus \{l_{top}\})$ ▷ Find highest decision level all literals except l_{top}

Example 2.2.1 *Consider the following set of clauses:*

$$\begin{aligned} C_1 &: \ (\neg x_1 \vee x_3 \vee \neg x_5) \\ C_2 &: \ (x_4 \vee \neg x_5) \\ C_3 &: \ (\neg x_1 \vee \neg x_3 \vee \neg x_4) \\ C_4 &: \ (\neg x_1 \vee \neg x_2 \vee x_5) \end{aligned}$$

We execute the following steps:

1. *Decide:* $x_1 \rightarrow 1$
2. *Decide:* $x_2 \rightarrow 1$

   $\Rightarrow$ *Propagate* $C_4 : x_5 \rightarrow 1$

   $\Rightarrow$ *Propagate* $C_2 : x_4 \rightarrow 1$

   $\Rightarrow$ *Propagate* $C_1 : x_3 \rightarrow 1$

   $\Rightarrow$ *Conflict in* C_3

2. *Conflict analysis,* $C_{conf} = C_3 = (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

   $\Rightarrow$ $C_{reason} = C_1 = (\neg x_1 \vee x_3 \vee \neg x_5)$
   Applying resolution eliminating x_3
   $C_{conf} = (\neg x_1 \vee \neg x_4 \vee \neg x_5)$

   $\Rightarrow$ $C_{reason} = C_2 = (x_4 \vee \neg x_5)$
   Applying resolution eliminating x_4
   $C_{conf} = (\neg x_1 \vee \neg x_5)$

   $\Rightarrow$ *1UIP found, only* $l_{top} = \neg x_5$ *was assigned on decision level 2.*
   Backtracking to DECISION-LEVEL$(\neg x_1) = 1$

1. *Adding* $C_5 : (\neg x_1 \vee \neg x_5)$ *to clause database*

   $\Rightarrow$ *Propagate* $C_5 : x_5 \rightarrow 0$

   $\Rightarrow$ *etc...*

Here, we learned a new clause C_5 allowing us to backtrack to decision level 1 and propagate the variable x_5.

2.3. Integer Linear Programming

Integer Linear Programming (ILP) is another restricted CSP formulation, in which an objective function c is optimized with respect to a set of linear constraints. These constraints are of the form $\sum_{i=1}^{n} a_i x_i \leq a_0$, with $a_i, x_i \in \mathbb{Z}$. The objective function is also a linear expression to be minimized or maximized, i.e. $\sum_{i=1}^{n} c_i x_i$ with $c_i \in \mathbb{Z}$.

2.3.1. Linear combinations

The linear counterpart of clausal resolution is the **combination rule**. This rule specifies that we can combine two linear inequalities $C_1 : \sum_i a_i x_i \leq a_0$ and $C_2 : \sum_i b_i x_i \leq b_0$ to eliminate variable x_j. Given $g = \text{GREATEST-COMMON-DENOMINATOR}(a_j, b_j)$, we can define multipliers $\alpha = |b_j|/g$ and $\beta = |a_j|/g$ to obtain the combined linear inequality C_3:

$$C_3 : \sum_i (a_i\beta + \alpha b_i)x_i \leq a_0\beta + \alpha b_0 \tag{2.3}$$

This combination eliminates variable x_j when a_j and b_j have opposite signs, i.e., when $a_j b_j < 0$.

2.3.2. Conflict analysis

The most common method of solving an ILP is to relax it into a real-valued formulation (LP) and disposing of non-integer solutions. However, conflict analysis for ILP can also be approached using techniques inspired by CDCL[33] in SAT.

Unlike in CDCL, where Boolean variables are decided and propagated, ILP deals with integer variables. We therefore propagate *bounds* instead of Boolean truth assignments. A bound $\langle lb \leq x \rangle$ or $\langle x \leq ub \rangle$ represents a propagation of a lower or upper bound of variable x. Given a linear inequality $a_1x_1 + \cdots + a_nx_n \leq a_0$, we can propagate the upper bound of x_1 using the lower bounds of $x_2, \ldots, x_n$ (assuming $a_i > 0$):

$$x_1 \leq \left\lfloor \frac{a_0 - \sum_{2 \leq i \leq n} \min(a_i x_i)}{a_1} \right\rfloor \tag{2.4}$$

A direct adaptation of CDCL to ILP would involve performing the *combination rule* (Section 2.3.1) between the conflicting constraint and the reason constraint. Similar to CDCL, the newly derived constraint should also be conflicting with the current assignment and trigger a propagation when added to the constraint database.

However, applying the combination rule introduces a problem: the derived constraints may not necessarily be conflicting under the current partial assignment, despite being generated from a constraint that was conflicting under the current partial assignment. We call this the *rounding problem*. Example 2.3.1, adapted from [36, Example 3.1], illustrates this issue.

Example 2.3.1 *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 $1 \in \mathcal{D}(x)$.*

Section 3.1 explores several methods from literature that address the rounding problem in ILP conflict analysis.

2.3.3. Pseudo-Boolean solving

Many ILP problems involve only binary variables, i.e., variables restricted to values of either 0 or 1. Such instances are referred to as pseudo-Boolean (PB) problems. Pseudo-Boolean formulations can express certain types of constraints more compactly than traditional SAT formulations. For instance, the

cardinality constraint "at least three variables must be true" can be concisely represented in PB form as $\sum_{i=1}^{n} x_i \geq 3$, whereas encoding this constraint in SAT typically requires a significantly more complex formulation.

The restricted formulation of a PB problem allows for employing an alternative conflict analysis technique compared to the standard ILP formulation. While PB problems remain vulnerable to the rounding problem outlined in Section 2.3.2, PB solvers can employ cutting-plane reductions in order to mitigate this problem. Chai [17] proposed using *saturation* to implement this reduction. Later Elffers et al. [22] introduced an alternative approach based on the *division rule* [13] instead of saturation. Using these cutting-plane reductions is guaranteed to produce asserting linear inequalities, unlike in the general ILP setting where such guarantees do not apply.

3

Related work

This section examines several methods of implementing conflict analysis in both ILP and CP. It covers a broad range of approaches to showcase the progress made over time. Firstly, adaptations of CDCL for ILP are discussed in Section 3.1. Following that, several conflict analysis and learning approaches in CP are discussed in Section 3.2.

3.1. CDCL in ILP

This section explores prior works that solve the ILP problem and implement conflict analysis analogous to CDCL, and demonstrates how they address the rounding problem (Section 2.3.2).

3.1.1. CutSat(++)

One of the first contributions that addressed the rounding problem for ILP was CutSat [16], which introduced several modifications to the standard CDCL algorithm. Firstly, variable decisions are restricted to assigning a variable to either its current lower or upper bound, rather than allowing decisions for arbitrary bounds. When a bound is propagated, a *tightly-propagating* reason inequality for that bound is computed. A tightly-propagating inequality is an inequality for which the variable's coefficient in the propagated bound is -1 or 1. Conflict analysis is then performed on the tightly-propagated reason constraints, which, by definition, do not suffer from the rounding problem as no divisions are necessary. This now always results in a learned constraint that propagates a new bound.

There are several limitations to this, however. First, the iterative combination step is performed until a decision is reached, which is significantly less efficient than 1UIP. Another downside of CutSat is that it might not terminate for instances with unbounded variables. To address these limitations, Bromberger et al. introduced CutSat++ [10] to refine the underlying calculus to ensure soundness, completeness, and guaranteed termination. However, the authors note that an efficient implementation of the CutSat++ calculus is not available and thus no no experimental results are included in their work.

3.1.2. IntSat

Shortly after the initial introduction of CutSat [16], IntSat [42, 36] was introduced. IntSat acknowledges that the rounding problem arises in integer conflict analysis and instead incorporates mechanisms to mitigate its effects. Consequently, IntSat overcomes the limitations associated with CutSat. Its conflict analysis consists of two distinct components, which together define the IntSat algorithm. These two components make up the complete IntSat algorithm as outlined in Algorithm 3.

Linear Combinations. 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 combination rule is applied to CC and RC to eliminate variable V. While the resulting learned constraint can always

be incorporated into the constraint database, its added value depends on whether it suffers from the rounding problem. To determine its effectiveness, it is checked whether the learned constraint can propagate a new bound at any previous decision level. If so, an *early backjump* is performed to the decision level at which the learned constraint is asserting. If the currently constructed conflicting constraint CC is not conflicting at the current decision level, it will not become conflicting when combined with additional reason constraints and the analysis will fail. Therefore, the algorithm maintains the invariant that CC must always be conflicting with the current partial assignment, and a violation of this invariant indicates failure.

Resolution. If an invariant violation occurs or a decision is reached after performing combinations, a resolution-based technique is applied. This method operates on bounds analogously to resolution on literals and is guaranteed to succeed. It constructs an asserting disjunction of bounds, referred to as a *clause* in the context of this section. However, since IntSat's constraint database is constrained to an ILP formulation, it cannot directly store and propagate clauses. These learned clauses are only integrated into the constraint database if they can be converted into a linear inequality—specifically, if at most one variable within the clause is non-binary. Otherwise, such clauses are used solely for propagation after backtracking and are discarded thereafter.

IntSat has shown to be very promising, solving several benchmark instances faster than state-of-the-art commercial MIP solvers. It was, therefore, suggested that IntSat be included as part of a solver's toolbox. To our knowledge, this has not happened yet, and it is interesting to research why.

Algorithm 3 IntSat Conflict Analysis

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

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

3.2. Conflict analysis and learning in CP

When solving a CSP, duplicate sub-trees likely exist within the search tree. *Thrashing* occurs when we encounter an identical sub-tree multiple times, only to repeatedly discover that this sub-tree leads to a conflict. This issue also arises in the context of SAT solving, where learning and conflict analysis have proven to be highly effective in pruning the search space. This section provides an overview of existing conflict analysis techniques for CP.

3.2.1. Nogood learning

Dechter [15] first introduced *nogood learning*. During the search process, nogood learning identifies sets of partial assignments that cannot be extended into a complete solution. If the current partial assignment matches a known nogood, it is guaranteed that this branch will not lead to a solution and can therefore be pruned from the search space.

However, the effectiveness of this basic form of nogood learning has been somewhat limited. Katsirelos

et al. [20] demonstrated that performance can be significantly improved by employing *generalized nogoods*. Unlike traditional nogoods, generalized nogoods capture not only infeasible combinations of assignments but also non-assignments. These non-assignments allow for additional pruning by eliminating search branches where a particular value has not been selected. For example, a generalized nogood $\{x \nleftarrow 1, y \leftarrow 2\}$ ensures that if x is assigned to 0, then 2 is pruned from $\mathcal{D}(y)$.

3.2.2. Signed clause learning

A more general representation of learned knowledge in constraint solving is through *signed clauses*, which are disjunctions of signed literals of the form $x \in D$ or $x \notin D$. These literals specify whether a variable x is or is not part of domain D. Veksler and Strichman's algorithm [49] learns and propagates these signed clauses during the search by employing a dedicated propagator for signed clauses. They make use of Beckert's signed binary resolution [7] around a *pivot variable* x:

$$\frac{((x \in A) \vee Y) \quad ((x \in B) \vee Z)}{(x \in (A \cap B) \vee Y \vee Z)} \tag{3.1}$$

This is a generalization of the resolution found in SAT.

Signed clauses are derived from *explanations* generated by the propagators. An explanation clause must satisfy two key conditions: (1) it must be logically implied by a constraint, and (2) it must be sufficiently strong to lead to the same propagation as the constraint did. For instance, the constraint $x \leq y$ can generate the explanation $x \in (-\infty, m] \vee y \in [m+1, \infty)$ for any value m. This explanation satisfies the first condition and, depending on the choice of m, will also satisfy the second. By iteratively performing signed resolution on the explanations of propagators, one can find an asserting *conflicting clause* that propagates variable domains. This approach mirrors the propositional resolution found in CDCL.

3.2.3. General constraint learning

Veksler and Strichman [31] have improved their signed clause learning scheme described in section 3.2.2 by extending signed clause resolution with non-clausal *combinations* of conflicting and reason constraints. The rationale behind this new approach is that this produces stronger learned constraints compared to their clausal counterparts. Experimental analysis shows that this new learning scheme was competitive with other solvers at the time this work was introduced.

Their approach defines combination rules for constraints c_1 and c_2 to derive an output constraint c^* that satisfies two key requirements. First, the output constraint must be logically inferred from the input constraints. Secondly, the output constraint must remain conflicting with the current partial assignment. However, due to the rounding problem, not all combination rules satisfy this second condition. In this case, the signed clause learning scheme is applied as a fallback.

The clausal fallback is also applied when no combination rule is defined between the conflicting and reason constraints. This approach can be viewed as the CP implementation of IntSat (Section 3.1.2), enhanced with additional combination rules. Since constraint combination in this approach is restricted to a limited set of predefined rules, it does not generalize to arbitrary, non-linear constraints. Our contribution addresses this limitation.

3.2.4. Lazy Clause Generation

Lastly, instead of employing a conflict analysis algorithm native to CP, we can leverage the mature conflict analysis techniques from SAT to aid our CP search. We lazily generate Boolean clauses, called *explanations*, that represent domain propagations of the variables in the CP model. This we call *Lazy Clause Generation* [38, 47]. With these Boolean clauses, we can perform conflict analysis and unit propagation using standard CDCL.

Ohrimenko et al. [38] proposed using a SAT engine to control the search, with CP propagators acting as clause generators. These propagators do not update the CP domains directly but instead generate clausal explanations for domain changes. To facilitate this, a mapping is defined between the SAT engine and CP, encoding the domains of CP variables as literals that describe a part of a variable's domain, such as $\langle x \leq 0 \rangle, \langle x \leq 1 \rangle, \ldots, \langle x \leq 10 \rangle$ for $x \in [0, 10]$. The SAT engine does not interpret the

meaning of these Boolean variables; therefore, inconsistencies can arise, such as $\langle x = 2 \rangle \wedge \langle x \leq 1 \rangle$ being true simultaneously. These inconsistencies can be addressed by adding additional constraints to define the relationships between the variables.

As the SAT engine controls the search, we first apply unit propagation there. Once completed, the Boolean domain encoding is translated back into a concrete domain. The CP propagators are then queried for a clause that explains their intended domain changes. This clause is subsequently added to the SAT engine, and unit propagation is executed again. This process continues until a fixpoint is reached.

Feydy and Stuckey [47] proposed an alternative approach, flipping the roles such that the CP solver controls the search, and the SAT engine becomes the highest priority global propagator. In this scheme, after a CP propagation, the CP propagator generates an explanation clause that describes the reason for the domain change. This in turn triggers the SAT propagator. For example, an explanation might look like $\langle x_1 \leq 10 \rangle \wedge \langle x_2 \leq 5 \rangle \rightarrow \langle x_3 \leq 5 \rangle$, with the intent to propagate $\langle x_3 \leq 5 \rangle$ in the SAT model. Since there can be multiple explanations for the same propagation, the goal is to find the *strongest* explanation, which is the most general explanation (i.e., the one with the fewest terms) that restricts the search space the most. This version of LCG is still considered state-of-the-art.

4

Lazy Linear Generation

We propose the Lazy Linear Generation (LLG) algorithm, an extension of LCG that learns asserting linear inequalities alongside clauses. LLG enhances LCG conflict analysis by incorporating cutting planes analysis, allowing learned constraints to capture linear relationships between variables—which is not possible when learning a single clause, as illustrated by Example 4.0.1.

Example 4.0.1 *Take for instance the constraint $x + y \leq 4$, with $\mathcal{D}(x) = \mathcal{D}(y) = \{0, 4\}$. Representing this simple problem requires four clauses:*

$$\langle x \leq 0 \rangle \vee \langle y \leq 3 \rangle$$
$$\langle x \leq 1 \rangle \vee \langle y \leq 2 \rangle$$
$$\langle x \leq 2 \rangle \vee \langle y \leq 1 \rangle$$
$$\langle x \leq 3 \rangle \vee \langle y \leq 0 \rangle$$

Extending LCG to perform conflict analysis using linear inequalities aligns closely with the challenges addressed by IntSat [36] and HaifaCSP [50]. 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 4.0.2.

Example 4.0.2 *Consider the multiplication $a \cdot b = c$ constraint, with initial domains $a \in \{2, 3\}, b \in \{1, 4\}, c \in \{2, 7\}$. The MiniZinc 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 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 3, 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.

This leads to two modifications. First, rather than directly retrieving the currently conflicting constraint from C, we attempt to derive a linear explanation for the conflict. If this is not feasible, we fall back to resolution. Similarly, rather than directly using RC, we now attempt to linearly explain each propagation. This requires a modification to the preconditions in lines 5 and 6 of Algorithm 3. In addition to these two conditions, we introduce an additional requirement: the combination step cannot proceed if RC cannot be linearly explained.

Finally, the resolution fallback is updated. In IntSat, the effectiveness of resolution is limited due to the absence of a clausal propagator. However, in CP, we can leverage LCG as a fallback, enabling native storage and propagation of learned clauses. When the solver fails to learn a linear constraint, it resorts to LCG, proceeding as though LLG were not present.

4.2. Linear explanations

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

4.2.1. Inferred constraints

Firstly, it is important to highlight that the asserting inequality derived from LLG 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. Put differently, a linear explanation introduces (a part of) a linear decomposition of a constraint. This approach differs fundamentally from clausal explanations. In LCG, explanations describe the partial assignment responsible for a propagation or conflict. In contrast, LLG explanations are constraints that encapsulate both the conditions for propagation and the underlying linear relationship responsible for it. This is demonstrated in Example 4.2.1.

Example 4.2.1 *Consider a constraint* $\max(a, b) = c$ *that propagates* $\langle b \leq 3 \rangle$ *when* $\langle c \leq 3 \rangle$*. Then,* $\langle c \leq 3 \rangle$ *is a clausal explanation for this propagation as at any point in the search tree, the constraint* $\max(a, b) = c$ *and partial assignment* $\langle c \leq 3 \rangle$ *will trigger the propagation* $\langle b \leq 3 \rangle$*.*

There are many correct alternative explanations, such as $\langle a \leq 3 \rangle \wedge \langle c \leq 3 \rangle$*. Given the partial assignment* $(a = 3, c = 3)$*, the same propagation will occur. However, this explanation is weaker than the previous explanation, as it enables propagation in fewer assignments than* $\langle c \leq 3 \rangle$ *because of the inclusion of the* $\langle a \leq 3 \rangle$ *term.*

For LLG, the explanation is an integer linear inequality directly implied from the original model. Ideally, this explanation would propagate given the current partial assignment. A valid LLG explanation for the propagation of $\langle b \leq 3 \rangle$ *would be* $b \leq c$*.*

4.2.2. Explanation signs

Additionally, it is noteworthy that for a linear explanation to accurately capture a propagation, 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 4.2.2 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 4.2.2 *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.*

4.2.3. 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 $\langle a \leq 2 \rangle \rightarrow b \geq 3$. We can encode the condition by introducing a Boolean auxiliary variable p_1, defined as:

$$p_1 = \begin{cases} 1 & \text{if } a \leq 2 \\ 0 & \text{if } a > 2 \end{cases} \tag{4.1}$$

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 [8].

4.3. Implementing auxiliary variables

The auxiliary variables associated with conditional explanations (see Section 4.2.3) 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.

4.3.1. 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 not strictly necessary, 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 4.3.1 illustrates this principle.

Example 4.3.1 *At decision level 10, an explanation introduces an auxiliary variable p, defined as $p \iff$*

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

4.3.2. 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, we explicitly evaluate the conditions of auxiliary variables rather the auxiliary variable's own bounds when checking for assertivity at any previous decision level. Although the auxiliary variables may not have been propagated yet at this previous decision level, it is still possible to infer the truth value of the condition.

4.4. Examples of explanations

We have formulated linear explanations for several global constraints. This section provides a detailed exposition of some of these explanations. A comprehensive overview of all explanations is presented in Appendix A.

Example 4.4.1 (Explanation of $\langle x_i \neq val \rangle$ for $Ax \neq b$) *The propagation of* $x_i \neq val$ *implies that* $\langle x_i < val \rangle \oplus \langle x_i > val \rangle$ *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 (pick either explanation).*

Example 4.4.2 (Explaining $\langle idx = i \rangle \rightarrow A[i] \leq rhs$ for $A[idx] = rhs$) *The element constraint enforces* $A[idx] = rhs$, *where* A *is an array of variables and* idx *is a variable representing the selected index. In explaining the propagation of* $A[i] \leq rhs$, *it is necessary to include the condition* $idx = i$, *which we encode using an auxiliary variable* p_i. *This leads to the following explanation:*

$$A[i] \leq rhs + M(1 - p_i) \tag{4.2}$$

We can encode the auxiliary variable p_i *in two ways.*

Direct explanation. *We can define the auxiliary variables as follows:*

$$p_i^{lt} \iff idx \leq i, \quad p_i^{gt} \iff idx \geq i, \quad p_i \iff p_i^{lt} + p_i^{gt} \geq 2 \tag{4.3}$$

MiniZinc linear explanation. *Alternatively, we can follow a formulation similar to the MiniZinc linear decomposition [8]. In this approach, we introduce* n *Boolean auxiliary variables corresponding to the* n *elements of the array and enforce the following constraints:*

1. *Exactly one auxiliary variable must be true:* $\sum_i p_i = 1$
2. *Set* idx *to the selected auxiliary variable:* $\sum_i i \cdot p_i = idx$

Do note that this explanation method necessitates bi-directional propagation of p_i *and* idx *to ensure domain changes of one variable, are reflected in the other.*

Due to engineering constraints that limit the number of auxiliary variables that should be created, LLG implements p_i *using the MiniZinc method as this only requires the addition of* n *auxiliary variables, whereas the direct explanation might create* $3n$ *auxiliary variables.*

Example 4.4.3 (Explanation of $\langle b \geq 5\rangle \wedge \langle c \geq 0\rangle \rightarrow a \leq \lfloor ub(c)/lb(b)\rfloor$ 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 $\langle b \geq 5\rangle \wedge \langle c \geq 0\rangle \rightarrow 5a \leq c$. 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:*

$$5a \leq c + M(1 - p_1) + M(1 - p_2). \tag{4.4}$$

Example 4.4.4 (Explanation of $\neg c \rightarrow r \leq 0$ for $r \rightarrow c$) *A noteworthy constraint is the* reified constraint*, 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 $r \leq 0$. Since c is a CP constraint, we can explain its conflict with a linear inequality of the form $Ax \leq b$. However, directly using $Ax \leq b$ to explain the propagation of $r \leq 0$ would incorrectly assert that $Ax \leq b$ (and hence constraint c) must always be enforced. Instead, they are only enforced when $r = 1$. Consequently, we incorporate r into the explanation to capture this conditional enforcement:*

$$Ax \leq b + M(1 - r). \tag{4.5}$$

5

Experiments

We implemented our LLG approach in Pumpkin[1]. The initial version of Pumpkin serves as the baseline LCG solver. We consider the number of conflicts as the main metric. This allows us to draw conclusions that are independent of the runtime and engineering issues. We believe this fairly shows the potential of our LLG approach. With this in mind, we leave optimizing the implementation as future work. To further ensure that the results are because of 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 strategy provided in the instances. Our results demonstrate that:

1. LLG encounters significantly fewer conflicts than LCG (Section 5.2.1).
2. Learned inequalities indeed provide stronger reasoning than clauses (Section 5.2.2).
3. While the success rate of LLG analysis is relatively low and decreases with time (Section 5.2.3), the learned constraints are highly effective.
4. LLG is generally more effective compared to a linear decomposition (Section 5.3).
5. The presence of clausal propagation is instrumental for LLG, underscoring the benefits of utilizing LCG as a fallback (Section 5.4).

These experiments provide a deeper insight into cutting planes conflict analysis in the context of constraint programming.

5.1. Experimental setup

Dataset selection The representative dataset used for the experiments is constructed by combining *all* models from the MiniZinc Challenges [44, 43] and Benchmarks[2] libraries. Each model is subjected to the following criteria and transformations:

- **Model selection:** For models obtained from the Minizinc Challenges, only problems that at least one finite domain solver has successfully solved within the 20-minute time limit are included. This is because the experimental analysis excludes instances that reach their timeout, as many metrics become incomparable in such cases. Furthermore, models containing floating-point numbers or unbounded integers are excluded. To maximize dataset diversity, no more than 10 instances per model are included. If a model exceeds this limit, 10 instances are randomly sampled from the available data files. Finally, only problems utilizing fixed-search branching are considered, ensuring that any variation in the number of conflicts is solely due to differences in learning.
- **Constraint decomposition:** Problems are decomposed into fundamental global constraints, namely, multiplication ($a \times b = c$), truncating division ($a/b = c$), absolute value ($a = |b|$),

[1] `https://github.com/ConSol-Lab/Pumpkin`
[2] `https://github.com/MiniZinc/minizinc-benchmarks`

maximum ($max(A) = b$), not equals ($Ax \neq b$), linear less-than-or-equal-to ($Ax \leq b$), reified ($r \rightarrow$ constraint), element ($A[idx] = b$) and clauses.

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.

Hardware All experiments were conducted on the DelftBlue[1] compute cluster, using 8GB of memory and a 1-hour timeout per instance.

Implementation correctness Developing a CP solver requires careful implementation. To detect as many potential implementation errors as possible, several tests were conducted. All tests were successfully passed. While this does not guarantee a completely bug-free implementation, it increases confidence in the correctness and reliability of the results. The following tests have been executed:

- **Solution Enumeration:** Verifying that the complete set of solutions from Pumpkin matches the one generated by Gecode[19]. The test was conducted on all satisfiability problems in the dataset, as well as on optimization problems that were transformed into satisfiability problems by enumerating all solutions whose objective value matched the optimal value (if Gecode was able to compute the optimal value within 10 seconds).
- **Explanation Validation:** Verifying that adding the negation of a generated explanation to a fresh model results in unsatisfiability. Since an explanation is generated for every propagation and conflict, performing this verification for every explanation in the full dataset was computationally infeasible. Therefore, a representative subset of 5000 explanations—uniformly sampled across different explanation types—was selected for this test.
- **Unit Tests and Assertions:** Incorporating numerous unit tests and assertions throughout the implementation.

5.2. LLG compared to LCG

5.2.1. Reduction of conflicts

This experiment evaluates the ratio of conflicts that are encountered in LLG compared to LCG across the instances that, (1) have been solved within the time-limit for both LLG and LCG, and, (2) have been able to learn at least one linear inequality. Of the 952 instances, LLG can solve 624 instances within the time-limit, whereas LCG can solve 663 instances within the time-limit. This difference can be attributed to the significantly higher number of out-of-memory errors for LLG, given its inefficient implementation. Additionally, in the case of LLG, approximately **70%**, or **443** of successful instances succeeded in learning any inequality. This shows that, with the current state of LLG, not all instances benefit from its linear conflict analysis.

The results in the boxplot in Figure 5.1 reveal several key trends. Firstly, 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 may learn very few linear constraints, or may learn linear constraints that propagate similar bounds as learned clauses. Lastly, the upper 75% to 90% percentile range highlights cases where the number of conflicts increase marginally. This phenomenon can be attributed to some learned constraints that are weaker compared to the clauses that could have been learned.

Our initial hypothesis asserted 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 ($574/171 \approx$ **3.4** instances per model): the top-performing instances are even more varied than the full dataset. Consequently, our initial hypothesis is refuted; a wide variety of problems benefit from LLG.

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

Lastly, although these experiments were conducted using an unoptimized implementation, 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%.

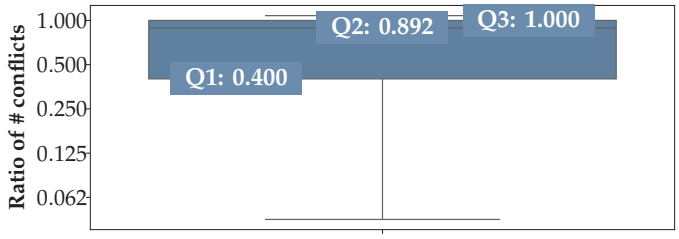


Figure 5.1: Boxplot showing the ratio of conflicts in LLG relative to LCG (lower = better). The boxplot displays the Q1, Q2 and Q3 quartiles, with whiskers extending to the 10% and 90% percentiles. A logarithmic scale is used for equal spacing of ratios. Only instances where both LCG and LLG successfully completed and at least one linear constraint was learned, are considered. LLG encounters significantly fewer conflicts than LCG for 50% of the instances, slightly fewer conflicts for 25% of the instances while for the remaining instances, the increase in conflicts is marginal if any.

5.2.2. Strength of learned inequalities

Since LLG was introduced based on the proposition that linear constraints may provide stronger reasoning than clauses, it is essential to verify whether this assumption is supported by experimental results. To this end, we compared two program variants: a standard LLG program and an alternative version in which linear analysis is performed, but instead of adding learned linear constraints, the fallback clauses are introduced into the model.

For the standard LLG program, we record for each propagation triggered by a learned linear constraint whether the fallback clause—had it been learned instead—would have resulted in the same propagation. Similarly, for the alternative program, we record for each propagation triggered by a learned clause that was generated in cases where a linear constraint could have been learned instead, whether the linear constraint—had it been learned instead—would have produced the same propagation.

The results (not in figure) 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.

Propagation of learned clauses and inequalities We also observe (not in figure) that for 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**, even increasing to **120** when only considering instances that finished within the time-out. This demonstrates that when an inequality can be learned, it generally has a much greater impact on the search compared to a learned clause.

This effect is even more evident when examining the means and standard deviations of the number of propagations. While learned clauses propagate an average of **16** times (std **680**), learned inequalities propagate **6,710** times (std **103,593**). This substantial deviation arises from the fact that there are several instances that contain learned inequalities with hundreds of thousands to even millions of propagations. These instances originate from different models, without a clear correlation among them. Figure 5.2 further illustrates this with a histogram depicting the distribution of propagations.

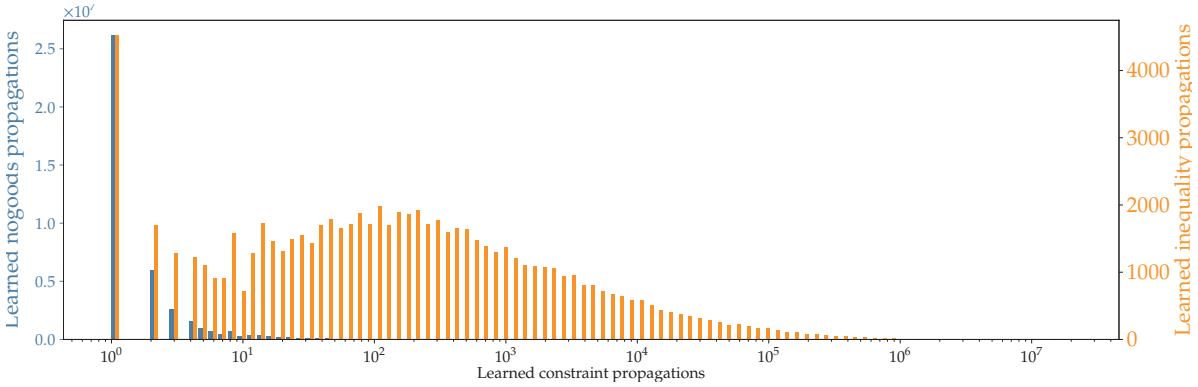


Figure 5.2: Histogram comparing the number of learned nogoods and inequalities (y-axis) that propagated a certain number of times (x-axis). Due to the lower number of inequalities relative to learned nogoods, separate y-scales are used. The x-axis is logarithmic due to the spread in propagation count for learned inequalities. Most learned nogoods propagate only once, with very few exceeding 10 propagations. In contrast, while many learned inequalities also propagate only once, a significant number propagate much more frequently.

5.2.3. 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. Therefore, three subsets of instances were selected: all instances (Figure 5.3a), those with at least a 50% reduction in conflicts (Figure 5.3b), and those with at least a 75% reduction in conflicts (Figure 5.3c).

Firstly, Figure 5.3a demonstrates a declining trend in successful analyses over time, accompanied by an increase in conflicts and propagations that cannot be explained 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%).

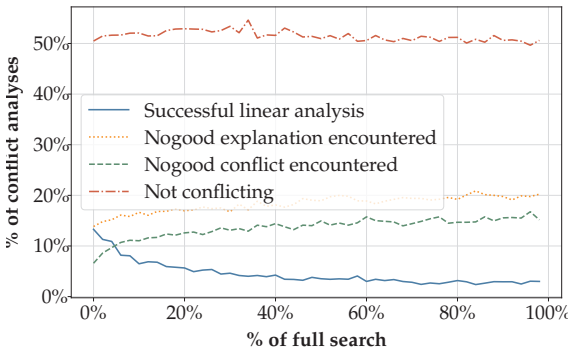
In contrast, Figures 5.3b and 5.3c exhibit some differences compared to the full dataset. Initially, conflict analysis is highly successful—significantly more so than for the full dataset. However, as the search progresses, analysis performance declines sharply. For these instances, we can see an increasing number of analysis failures due to invariant violations increase substantially throughout the search. Figures 5.3a through 5.3c all illustrate that LLG conflict analysis succeeds in only a relatively small fraction of instances. This experiment demonstrates that even the 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 propagations of learned constraints. Figure 5.3d 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.

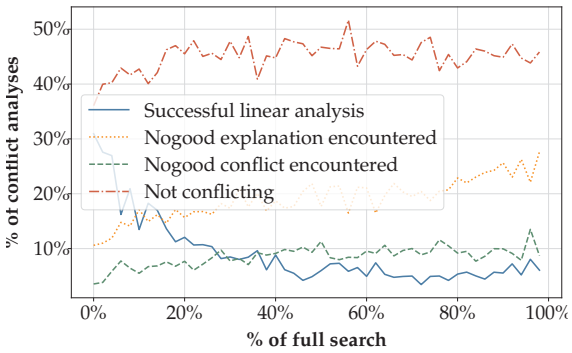
Explanation slacks Finally, we examined the slack of explanations that resulted in a learned inequality compared to those that did not. In the context of LLG, slack is defined as $\textsc{slack}(Ax \leq b) = b - \textsc{lb}(Ax)$, where $\textsc{lb}$ computes the current lower bound. Notably, a negative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

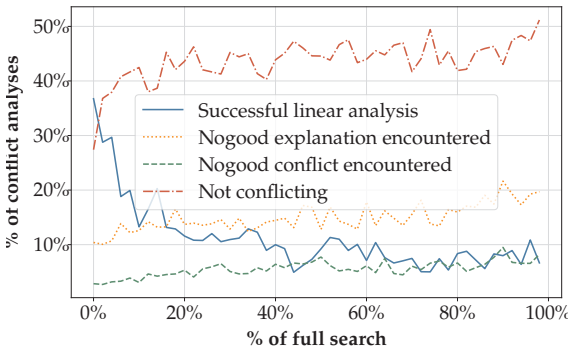
negative slacks are more likely to succeed. The only necessary condition for a conflict explanation is that is has negative slack. Second, we observed a strong correlation between the slack of a propagation explanation and its likelihood of resulting in a learned constraint: explanations with lower slack 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 slacks, prioritizing those that are more likely to contribute to the derivation of a learned constraint.



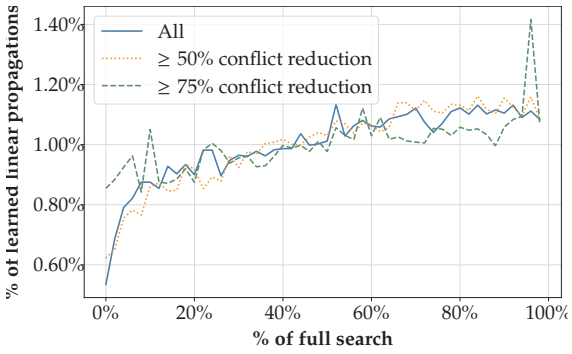
(a) All instances



(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 5.3: 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 nogoods encountered increases. 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 exhibit the highest concentration of constraint propagations. This can be explained by the cumulative effect of both newly introduced and previously learned constraints.

5.3. 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 [3]. 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.

For the 938 successfully converted instances, Figure 5.4a 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,

[3] `https://github.com/MiniZinc/libminizinc/tree/master/share/minizinc/linear`

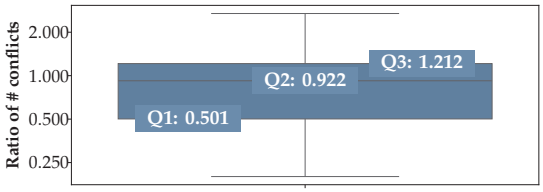
the first and second quartiles (Q1 to Q2) of the boxplot show a notable decline in conflicts, ranging from **50%** to **8%**. The second to third quartiles (Q2 to Q3) display a mix of reductions, from an **8%** reduction to an increase of **20%**. The remaining instances exhibit a slightly larger increase 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 an alternative formulation that can be linearly explained more effectively. Nevertheless, LLG continues to exhibit a significant performance advantage over the linear decomposition.

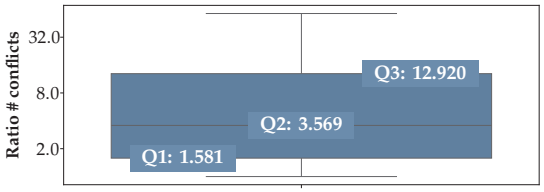
5.4. LLG without clause learning

Previous experiments have shown that a key reason for failing to learn linear constraints during LLG conflict analysis is encountering nogoods. Therefore, we investigate the consequences of only propagating a learned clause once, without storing it in the constraint database. The results, presented in Figure 5.4b, indicate that omitting clause storage leads to a significant increase in conflicts. This observation underscores that even though encountering nogoods resulted 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 [36]. 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.



(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 5.4: 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.

6

Conclusion and Future Work

Constraint analysis and learning in CP have been extensively studied, leading to the highly successful Lazy Clause Generation [47] algorithm, which performs clause learning. However, linear constraints offer the potential for more powerful reasoning capabilities than clauses. Previous works [36, 50] have explored cutting-plane reasoning within Integer Linear Programming (ILP) and CP, but they do not generalize to arbitrary CP propagators. This thesis therefore aims to investigate how cutting-plane analysis can be incorporated into Constraint Programming (CP) for arbitrary propagators and to experimentally investigate its effect on the search process.

For this, we introduce Lazy Linear Generation (LLG), a conflict analysis algorithm that integrates concepts from these prior studies to formulate a novel learning mechanism for linear constraints. A key distinction of LLG from prior methods is that it explicitly derives linear explanations for propagations and conflicts, leveraging these explanations for linear conflict analysis. These explanations, inferred directly from the original model, linearly describe the corresponding propagation and can be constructed using techniques such as those outlined in [8]. Furthermore, explanations may incorporate Boolean auxiliary variables that encode necessary conditions for the explanations to hold. These auxiliary variables are generated dynamically during the search process and are appropriately propagated by LLG.

Experimental evaluations of LLG on 952 instances demonstrate that learning linear constraints reduces the number of conflicts by a median of 10%, reaching up to 60% for the 25th percentile, compared to learning only clauses. We show that linear explanations provide stronger reasoning than clauses, but they do not strictly dominate clauses. Although the success rate of linear conflict analysis is relatively low, the learned linear constraints yield a notable impact on solver performance. When compared to a linear decomposition using the same conflict analysis, we observe that LLG provides a considerable reduction in the number of conflicts encountered. Furthermore, our experiments highlight that using a clausal propagation as a fallback is important, even when using conflict analysis based on cutting planes. These experiments show that our approach shows potential in obtaining much more effective constraint solving.

6.1. Future work

There are several promising avenues for further research of the LLG algorithm.

Additional explanations Firstly, it is interesting to consider linear explanations for additional global constraints, such as the *cumulative* constraint. These global constraints may have a very large linear decomposition that can now be effectively incorporated in conflict analysis, whilst benefiting from the smaller CP formulation of the constraint.

Improving explanations Second, existing linear explanations can be refined by for instance incorporating additional techniques as proposed in [8]. Specifically, various improvements can be implemented to

minimize the need for auxiliary variables or to simplify explanations when the initial variable domains permit. Furthermore, a significant opportunity lies in balancing the trade-off between the strength of an explanation and the use of auxiliary variables.

Lazily generating explanations based on the current conflict analysis state may lead to producing more useful linear explanations. For instance, it may be possible to selectively include or exclude certain variables from an explanation depending on their presence in the conflicting constraint.

LLG and LCG integration Next, clausal propagations could also be linearly explained through the introduction of auxiliary variables for every predicate. It remains unclear whether linear conflict analysis is useful when applied to these non-linear clauses, and whether the benefits outweigh the overhead introduced by the additional auxiliary variables required for this approach. Moreover, enabling LCG to resume from the point where LLG left off—rather than restarting conflict analysis from scratch—could potentially lead to improved clausal explanations and improved run-time efficiency.

Engineering improvements Finally, several engineering improvements can be implemented to allow for LLG's run-time to be evaluated and compared to other solvers. For instance, lazily generating explanations would ensure that auxiliary variables are introduced only when they are present in a learned constraint. Avoiding the generation and propagation of auxiliary variables that are never utilized can significantly reduce their computational overhead and improve runtime performance.

6.2. Closing thoughts

This section provides several comments on the current state of LLG, along with reflections that may guide its future developments. While these remarks do not constitute explicit contributions, they may nonetheless prove valuable in guiding further advancements of LLG.

Firstly, the explanations currently provided represent only an initial attempt and can be easily improved. Due to inefficiencies in the current implementation of LLG, it was necessary to limit the number of auxiliary variables in explanations. Consequently, some explanations are currently overly general, reducing their effectiveness in conflict analysis. However, if auxiliary variables are introduced to the search only when they are part of a learned constraint, they can be incorporated more frequently in explanations, enabling them to more accurately capture the original propagation.

When developing new explanations, it is useful to categorize propagations into three groups:

1. **Propagations also performed by the linear decomposition**. These can be easily explained by outputting the decomposition itself or identifying an equivalent linear explanation.
2. **Propagations that can be expressed as linear relationship, but are not performed by the linear decomposition**. In many cases, the linear decomposition of a constraint cannot replicate the full extent of propagation achieved by a dedicated CP propagator, as doing so would necessitate an impractically large linear model. However, these propagations often include a linear component that is valid only under certain conditions. When such conditions are explicitly encoded using auxiliary variables, linear analysis can yield insights that go beyond those obtainable through standard linear decomposition
3. **Propagations without a discernible linear relationship**. There are also propagations that do not have any linear component. These propagations essentially require clausal explanations that are subsequently converted into linear inequalities using auxiliary variables. However, this does not necessarily indicate that linear analysis for these propagations is ineffective; rather, a linear relationship may still exist between auxiliary variables and other model variables.

It might be helpful to consider these three categories when designing explanations for propagations, as constructing a meaningful explanation may not always be feasible in the absence of an underlying linear relationship. Additionally, it is interesting to evaluate the linear analysis capabilities of these three propagation groups.

Finally, the effectiveness of linear conflict analysis—and consequently, the overall search performance—depends not only on the quality of explanations but also on the specific propagations performed.

The manner in which a constraint is formulated plays a crucial role in the efficacy of linear analysis. Consider, for example, the `alldifferent` constraint. It can be represented using a pairwise not-equals formulation, or through a more involved encoding as in the MiniZinc linear decomposition. While the former formulation avoids the need for any additional variables in the problem formulation, it is challenging to provide meaningful linear explanations. In contrast, the latter formulation introduces additional variables that significantly enhance the linear learning capabilities. This trade-off between propagation efficiency and learning effectiveness has not been investigated in this thesis and is interesting to explore further.

References

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

[2] Tobias Achterberg. "Conflict analysis in mixed integer programming". In: *Discrete Optimization* 4.1 (Mar. 2007), pp. 4–20. DOI: 10.1016/j.disopt.2006.10.006.

[3] Tobias Achterberg. "SCIP: solving constraint integer programs". en. In: *Mathematical Programming Computation* 1.1 (July 2009), pp. 1–41. ISSN: 1867-2949, 1867-2957. DOI: 10.1007/s12532-008-0001-1. URL: http://link.springer.com/10.1007/s12532-008-0001-1 (visited on 09/25/2024).

[4] Roberto Amadini et al. "Constraint Programming for Dynamic Symbolic Execution of JavaScript". en. In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. ISSN: 1611-3349. Springer, Cham, 2019, pp. 1–19. ISBN: 978-3-030-19212-9. DOI: 10.1007/978-3-030-19212-9_1. URL: https://link.springer.com/chapter/10.1007/978-3-030-19212-9_1 (visited on 03/28/2025).

[5] Andreas Schutt. "Improving scheduling by learning". In: (Jan. 2011). MAG ID: 1725453496 S2ID: 70dd68b2b0981c9382cc8b50faae6aa5c2d082ae.

[6] Nicholas Beaumont. "Scheduling staff using mixed integer programming". en. In: *European Journal of Operational Research* 98.3 (May 1997), pp. 473–484. ISSN: 03772217. DOI: 10.1016/S0377-2217(97)00055-6. URL: https://linkinghub.elsevier.com/retrieve/pii/S0377221797000556 (visited on 09/25/2024).

[7] B. Beckert, R. Hahnle, and F. Manya. "The 2-SAT problem of regular signed CNF formulas". In: *Proceedings 30th IEEE International Symposium on Multiple-Valued Logic (ISMVL 2000)*. Portland, OR, USA: IEEE Comput. Soc, 2000, pp. 331–336. ISBN: 978-0-7695-0692-0. DOI: 10.1109/ISMVL.2000.848640. URL: http://ieeexplore.ieee.org/document/848640/ (visited on 10/02/2024).

[8] Gleb Belov et al. "Improved Linearization of Constraint Programming Models". In: *Principles and Practice of Constraint Programming*. Ed. by Michel Rueher. Cham: Springer International Publishing, 2016, pp. 49–65. ISBN: 978-3-319-44953-1.

[9] R. Brinkmann and R. Drechsler. "RTL-datapath verification using integer linear programming". In: *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design*. Bangalore, India: IEEE Comput. Soc, 2002, pp. 741–746. ISBN: 978-0-7695-1441-3. DOI: 10.1109/ASPDAC.2002.995022. URL: http://ieeexplore.ieee.org/document/995022/ (visited on 09/25/2024).

[10] Martin Bromberger, Thomas Sturm, and Christoph Weidenbach. "A complete and terminating approach to linear integer solving". In: *Journal of Symbolic Computation* 100 (Sept. 2020), pp. 102–136. DOI: 10.1016/j.jsc.2019.07.021.

[11] Eray Cakici, Ibrahim Kucukkoc, and Mustafa Akdemir. "Advanced constraint programming formulations for additive manufacturing machine scheduling problems". en. In: *Journal of the Operational Research Society* (July 2024), pp. 1–16. ISSN: 0160-5682, 1476-9360. DOI: 10.1080/01605682.2024.2382867. URL: https://www.tandfonline.com/doi/full/10.1080/01605682.2024.2382867 (visited on 10/08/2024).

[12] Geoffrey Chu et al. *Chuffed: The chuffed CP solver*. URL: https://github.com/chuffed/chuffed.

[13] W. Cook, C.R. Coullard, and Gy. Turán. "On the complexity of cutting-plane proofs". en. In: *Discrete Applied Mathematics* 18.1 (Sept. 1987), pp. 25–38. ISSN: 0166218X. DOI: 10.1016/0166-218X(87)90039-4. URL: https://linkinghub.elsevier.com/retrieve/pii/0166218X87900394 (visited on 09/20/2024).

[14] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Communications of the ACM* 5.7 (July 1962). Publisher: Association for Computing Machinery (ACM), pp. 394–397. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/368273.368557.

[15] Rina Dechter. "Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition". en. In: *Artificial Intelligence* 41.3 (Jan. 1990), pp. 273–312. ISSN: 00043702. DOI:

10.1016/0004-3702(90)90046-3. URL: https://linkinghub.elsevier.com/retrieve/pii/0004370290900463 (visited on 09/27/2024).

[16] Dejan Jovanović et al. "Cutting to the Chase solving linear integer arithmetic". In: *Automated Deduction* CADE-23 (July 2011), pp. 338–353. DOI: 10.1007/978-3-642-22438-6_26.

[17] Donald Chai et al. "A fast pseudo-Boolean constraint solver". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.3 (Feb. 2005), pp. 305–317. DOI: 10.1109/tcad.2004.842808.

[18] Soroush Fatemi-Anaraki et al. "Scheduling of Multi-Robot Job Shop Systems in Dynamic Environments: Mixed-Integer Linear Programming and Constraint Programming Approaches". en. In: *Omega* 115 (Feb. 2023), p. 102770. ISSN: 03050483. DOI: 10.1016/j.omega.2022.102770. URL: https://linkinghub.elsevier.com/retrieve/pii/S0305048322001773 (visited on 10/08/2024).

[19] Gecode Team. *Gecode: Generic Constraint Development Environment*. Available from http://www.gecode.org. 2006.

[20] George Katsirelos et al. "Generalized nogoods in CSPs". In: *AAAI Conference on Artificial Intelligence* (July 2005), pp. 390–396.

[21] Robert M. Haralick and Gordon L. Elliott. "Increasing tree search efficiency for constraint satisfaction problems". en. In: *Artificial Intelligence* 14.3 (Oct. 1980), pp. 263–313. ISSN: 00043702. DOI: 10.1016/0004-3702(80)90051-X. URL: https://linkinghub.elsevier.com/retrieve/pii/000437028090051X (visited on 10/09/2024).

[22] Jan Elffers et al. "Divide and Conquer: Towards Faster Pseudo-Boolean Solving". In: *International Joint Conference on Artificial Intelligence* (July 2018), pp. 1291–1299. DOI: 10.24963/ijcai.2018/180.

[23] Jo Devriendt et al. "Learn to relax: Integrating 0-1 integer linear programming with pseudo-Boolean conflict-driven search". In: *Constraints - An International Journal* (2021), pp. 1–30. DOI: 10.1007/s10601-020-09318-x.

[24] Dejan Jovanović and Leonardo de Moura. "Cutting to the Chase Solving Linear Integer Arithmetic". In: *Automated Deduction – CADE-23*. Ed. by Nikolaj Bjørner and Viorica Sofronie-Stokkermans. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 338–353. ISBN: 978-3-642-22438-6.

[25] Carla Juvin, Laurent Houssin, and Pierre Lopez. "Constraint Programming for the Robust Two-Machine Flow-Shop Scheduling Problem with Budgeted Uncertainty". English. In: *INTEGRATION OF CONSTRAINT PROGRAMMING, ARTIFICIAL INTELLIGENCE, AND OPERATIONS RESEARCH, CPAIOR 2023*. Ed. by A. A. Cire. Vol. 13884. Cham: Springer International Publishing Ag, 2023, pp. 354–369. DOI: 10.1007/978-3-031-33271-5_23. URL: https://link.springer.com/chapter/10.1007/978-3-031-33271-5_23 (visited on 10/08/2024).

[26] Dongyun Kim et al. "Iterated Greedy Constraint Programming for Scheduling Steelmaking Continuous Casting". English. In: *INTEGRATION OF CONSTRAINT PROGRAMMING, ARTIFICIAL INTELLIGENCE, AND OPERATIONS RESEARCH, CPAIOR 2023*. Ed. by A. A. Cire. Vol. 13884. Cham: Springer International Publishing Ag, 2023, pp. 477–492. DOI: 10.1007/978-3-031-33271-5_31. URL: https://link.springer.com/chapter/10.1007/978-3-031-33271-5_31 (visited on 10/08/2024).

[27] Arie Koster, Manuel Kutschka, and Christian Raack. "Towards robust network design using integer linear programming techniques". In: *6th EURO-NGI Conference on Next Generation Internet*. Paris, France: IEEE, June 2010, pp. 1–8. ISBN: 978-1-4244-8167-5. DOI: 10.1109/NGI.2010.5534462. URL: http://ieeexplore.ieee.org/document/5534462/ (visited on 09/25/2024).

[28] Wen-Yang Ku and J. Christopher Beck. "Mixed Integer Programming models for job shop scheduling: A computational analysis". en. In: *Computers & Operations Research* 73 (Sept. 2016), pp. 165–173. ISSN: 03050548. DOI: 10.1016/j.cor.2016.04.006. URL: https://linkinghub.elsevier.com/retrieve/pii/S0305054816300764 (visited on 09/25/2024).

[29] Alexander Lieder, Dirk Briskorn, and Raik Stolletz. "A dynamic programming approach for the aircraft landing problem with aircraft classes". en. In: *European Journal of Operational Research* 243.1 (May 2015), pp. 61–69. ISSN: 03772217. DOI: 10.1016/j.ejor.2014.11.027. URL: https://linkinghub.elsevier.com/retrieve/pii/S0377221714009503 (visited on 09/25/2024).

[30] Paramet Luathep et al. "Global optimization method for mixed transportation network design problem: A mixed-integer linear programming approach". en. In: *Transportation Research Part B: Methodological* 45.5 (June 2011), pp. 808–827. ISSN: 01912615. DOI: 10.1016/j.trb.2011.02.002.

URL: https://linkinghub.elsevier.com/retrieve/pii/S0191261511000166 (visited on 09/25/2024).

[31] M. A. Veksler et al. "Learning General Constraints in CSP". In: *Artificial Intelligence* (May 2015), pp. 410–426. DOI: 10.1007/978-3-319-18008-3_28.

[32] Inês Marques, M. Eugénia Captivo, and Margarida Vaz Pato. "An integer programming approach to elective surgery scheduling: Analysis and comparison based on a real case". en. In: *OR Spectrum* 34.2 (Apr. 2012), pp. 407–427. ISSN: 0171-6468, 1436-6304. DOI: 10.1007/s00291-011-0279-7. URL: http://link.springer.com/10.1007/s00291-011-0279-7 (visited on 09/25/2024).

[33] João Marques-Silva and Karem A. Sakallah. "GRASP: a search algorithm for propositional satisfiability". In: *IEEE Transactions on Computers* 48.5 (May 1999), pp. 506–521. DOI: 10.1109/12.769433.

[34] Filipe F. Mazzini and Geraldo R. Mateus. "A mixed-integer programming model for the cellular telecommunication network design". en. In: *Proceedings of the 5th international workshop on Discrete algorithms and methods for mobile computing and communications*. Rome Italy: ACM, July 2001, pp. 68–76. ISBN: 978-1-58113-421-6. DOI: 10.1145/381448.381458. URL: https://dl.acm.org/doi/10.1145/381448.381458 (visited on 09/25/2024).

[35] Matthew W. Moskewicz et al. "Chaff: engineering an efficient SAT solver". In: *Proceedings - Design Automation Conference* (June 2001), pp. 530–535. DOI: 10.1145/378239.379017.

[36] Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. "IntSat: integer linear programming by conflict-driven constraint learning". In: *Optim. Methods Softw.* (Sept. 2023), pp. 1–28. DOI: 10.1080/10556788.2023.2246167.

[37] Huimin Niu, Xuesong Zhou, and Ruhu Gao. "Train scheduling for minimizing passenger waiting time with time-dependent demand and skip-stop patterns: Nonlinear integer programming models with linear constraints". en. In: *Transportation Research Part B: Methodological* 76 (June 2015), pp. 117–135. ISSN: 01912615. DOI: 10.1016/j.trb.2015.03.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0191261515000478 (visited on 09/25/2024).

[38] Olga Ohrimenko et al. "Propagation = lazy clause generation". In: *International Conference on Principles and Practice of Constraint Programming* 4741 (Sept. 2007), pp. 544–558. DOI: 10.1007/978-3-540-74970-7_39.

[39] Olga Ohrimenko et al. "Propagation via lazy clause generation". In: *Constraints - An International Journal* 14.3 (Sept. 2009), pp. 357–391. DOI: 10.1007/s10601-008-9064-x.

[40] Laurent Perron and Frédéric Didier. *CP-SAT*. Version v9.11. Google, May 7, 2024. URL: https://developers.google.com/optimization/cp/cp_solver/.

[41] Olivier Ponsini, Claude Michel, and Michel Rueher. "Verifying floating-point programs with constraint programming and abstract interpretation techniques". en. In: *Automated Software Engineering* 23.2 (June 2016). Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 2 Publisher: Springer US, pp. 191–217. ISSN: 1573-7535. DOI: 10.1007/s10515-014-0154-2. URL: https://link.springer.com/article/10.1007/s10515-014-0154-2 (visited on 03/28/2025).

[42] Robert Nieuwenhuis. "The IntSat Method for Integer Linear Programming". In: *International Conference on Principles and Practice of Constraint Programming* (Sept. 2014), pp. 574–589. DOI: 10.1007/978-3-319-10428-7_42.

[43] Peter J. Stuckey, Ralph Becket, and Julien Fischer. "Philosophy of the MiniZinc challenge". en. In: *Constraints* 15.3 (July 2010), pp. 307–316. ISSN: 1383-7133, 1572-9354. DOI: 10.1007/s10601-010-9093-0. URL: http://link.springer.com/10.1007/s10601-010-9093-0 (visited on 03/17/2025).

[44] Peter J. Stuckey et al. "The MiniZinc Challenge 2008–2013". en. In: *AI Magazine* 35.2 (June 2014), pp. 55–60. ISSN: 0738-4602, 2371-9621. DOI: 10.1609/aimag.v35i2.2539. URL: https://onlinelibrary.wiley.com/doi/10.1609/aimag.v35i2.2539 (visited on 03/17/2025).

[45] Pierre Talbot, Tingting Hu, and Nicolas Navet. "Constraint Programming with External Worst-Case Traversal Time Analysis". In: *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Ed. by Roland H. C. Yap. Vol. 280. Leibniz International Proceedings in Informatics (LIPIcs). ISSN: 1868-8969. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 34:1–34:20. ISBN: 978-3-95977-300-3. DOI: 10.4230/LIPIcs.CP.2023.34. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2023.34 (visited on 03/28/2025).

[46] Thibaut Feydy. "Constraint programming: improving propagation". In: *University of Melbourne* (2010).

[47] Thibaut Feydy et al. "Lazy clause generation reengineered". In: *International Conference on Principles and Practice of Constraint Programming* 5732 (Sept. 2009), pp. 352–366. DOI: 10.1007/978-3-642-04244-7_29.

[48] Charles Thomas and Pierre Schath. "A Constraint Programming Approach for Aircraft Disassembly Scheduling". English. In: *INTEGRATION OF CONSTRAINT PROGRAMMING, ARTIFICIAL INTELLIGENCE, AND OPERATIONS RESEARCH, PT II, CPAIOR 2024*. Ed. by B. Dilkina. Vol. 14743. ISSN: 0302-9743, 1611-3349 Num Pages: 10 Series Title: Lecture Notes in Computer Science Web of Science ID: WOS:001283965200013. Cham: Springer International Publishing Ag, 2024, pp. 211–220. DOI: 10.1007/978-3-031-60599-4_13. URL: https://link.springer.com/chapter/10.1007/978-3-031-60599-4_13 (visited on 10/08/2024).

[49] Michael Veksler and Ofer Strichman. "A Proof-Producing CSP Solver". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 24.1 (July 2010), pp. 204–209. ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v24i1.7543. URL: https://ojs.aaai.org/index.php/AAAI/article/view/7543 (visited on 09/23/2024).

[50] Michael Veksler and Ofer Strichman. "Learning general constraints in CSP". en. In: *Artificial Intelligence* 238 (Sept. 2016), pp. 135–153. ISSN: 00043702. DOI: 10.1016/j.artint.2016.06.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S0004370216300650 (visited on 10/04/2024).

[51] Hang Wang et al. "Optimal scheduling of micro-energy grid with integrated demand response based on chance-constrained programming". en. In: *International Journal of Electrical Power & Energy Systems* 144 (Jan. 2023), p. 108602. ISSN: 01420615. DOI: 10.1016/j.ijepes.2022.108602. URL: https://linkinghub.elsevier.com/retrieve/pii/S0142061522005981 (visited on 10/08/2024).

[52] Sameela Suharshani Wijesundara, Maria Garcia de la Banda, and Guido Tack. "Addressing Problem Drift in UNHCR Fund Allocation". In: *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Ed. by Roland H. C. Yap. Vol. 280. Leibniz International Proceedings in Informatics (LIPIcs). ISSN: 1868-8969. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 37:1–37:18. ISBN: 978-3-95977-300-3. DOI: 10.4230/LIPIcs.CP.2023.37. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2023.37 (visited on 03/28/2025).

A

All explanations

Table A.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 A.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	$lb(Ax) = ub(Ax) = b$	$Ax \geq b + 1 - Mp$	
$a = \lvert b \rvert$	$a \geq 0$	-	$a \geq 0$	• $p_1 \iff b \leq 0$ • $p_2 \iff b \geq 0$
	$a \geq lb(b)$	$lb(b) > 0$	$a \geq b$	
	$b \geq -ub(a)$	-		
	$a \geq \lvert ub(b) \rvert$	$ub(b) < 0$	$a \geq -b$	
	$b \leq ub(a)$	-		
	$a \leq \max(\lvert lb(b) \rvert, \lvert ub(b) \rvert)$	-	• $a + b \leq M(1-p_1)$ • $a - b \leq M(1-p_2)$	
	$b \leq -lb(a)$	$ub(b) \leq 0$		
	$b \geq lb(a)$	$lb(b) \geq 0$		
$max(A) = b$	$a_i \leq ub(b)$	-	$a_i \leq b$	• $\forall i \in \lvert A \rvert$: auxiliary p_i • $\sum p_i = 1$
	$b \geq \max_{a_i \in A}(lb(a_i))$	-		
	$b \leq \max_{a_i \in A}(ub(a_i))$	-	$rhs \leq a_i + M(1-p_i)$	
	$a_i \geq lb(b)$	a_i only for $ub(a_i) \geq lb(b)$		
	$b \leq ub(a_i)$	$p_i = 1$		
	$a_i \geq lb(b)$	$p_i = 1$		
$a \times b = c$	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: $rhs \geq 0$	$lb(num) \geq 0 \wedge lb(den) \geq 0$	$rhs \geq 0 - Mn_{num} - Mn_{den}$	• $n_{num} \iff num < 0$ • $n_{den} \iff den < 0$ • $p_{den} \iff den > 0$ • $rhs_{lb} \iff rhs \geq lb(rhs)$ • $den_{ub} \iff den \leq ub(den)$
	$rhs \leq ub(num)/lb(den)$	$ub(num) \geq 0 \wedge ub(den) \geq 0$	$lb(rhs) \cdot den \leq num + M(1-rhs_{lb}) + Mn_{num}$ Swap rhs and den for alternative	
	$den \leq ub(num)/lb(rhs)$	$lb(rhs) \geq 0 \wedge ub(num) \geq 0$		
	$num \geq lb(rhs) \cdot lb(den)$	$lb(rhs) \geq 0 \wedge ub(den) \geq 0$		
	$rhs \geq lb(num)/ub(den)$	$lb(num) \geq 0 \wedge ub(den) \geq 0$	$ub(den) \cdot rhs + den - 1 \geq num - M(1-den_{ub}) - Mn_{num} - M(1-p_{den})$	
	$num \leq (ub(rhs)+1) \cdot ub(den) - 1$	$ub(den) \geq 0 \wedge ub(rhs) \geq 0$	Swap rhs and den for alternative	
$A[idx] = b$	$b \leq \max_{a_i \in A if i \in idx}(ub(a_i))$	-	$b \leq a_i + M(1-p_i)$	• $\forall i \in \lvert A \rvert$: auxiliary p_i • $\sum p_i = 1$ • $\sum i \cdot p_i = idx$
	$a_i \leq ub(b)$	$idx = i$		
	$b \geq \min_{a_i \in A if i \in idx}(lb(a_i))$	-	$b \geq a_i - M(1-p_i)$	
	$a_i \geq lb(b)$	$idx = i$		
	$p_i \geq 1$	$idx = i$	$\sum i \cdot p_i = idx$	
	$p_i \leq 0$	$idx \neq i$		
$p \rightarrow cond$	$p \leq 0$	$\neg cond$	$expl(cond) \rightarrow Ax \leq b$ $Ax \leq b + M(1-p)$	-
	Conflict	$p \geq 1 \wedge \neg cond$		

B

Conference Paper

This appendix includes a copy of the preprint version of the conference paper based on this thesis, submitted to the CP2025 conference. At the time of the submission of this thesis, the paper is still under review.

Conflict Analysis Based on Cutting Planes for Constraint Programming

Robbin Baauw
Delft University of Technology, Netherlands

Maarten Flippo
Delft University of Technology, Netherlands

Emir Demirović
Delft University of Technology, 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 to 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 arithmetic constraints (not-equal, absolute value, maximum, integer multiplication, truncating division) and the element constraint. An experimental evaluation on 952 MiniZinc Challenge instances shows that our approach greatly reduces the number of conflicts compared to LCG. The number of conflicts is also reduced compared to decomposing to linear inequalities, due to the stronger propagation in the CP solver. While more engineering efforts are needed to fully exploit the potential, our analysis and prototype implementation shows 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

Funding Maarten Flippo is 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).
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 [37] (CP) is an important paradigm for solving combinatorial optimization problems. It has applications in many domains, including resource allocation [39, 35], scheduling [31, 2], and verification [30, 4]. 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 [10]. Much work has been done to implement constraint learning effectively in CP solvers [19, 28, 36, 20, 23, 38]. Of the approaches, Lazy Clause Generation (LCG) [28, 36], is the most wide-spread, implemented by solvers such as OR-Tools [29] and Chuffed [9]. For

many problems, constraint learning is crucial [36, 1] 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 [38]. For paradigms other than CP, work has also been done exploring the learning of pseudo-Boolean (PB) constraints [12, 16] and integer linear constraints [26, 18, 3]. 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 [26], 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 [21] 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 [15]. One major difference between IntSat's conflict analysis procedure and a clausal conflict analysis procedure, is that in the former the analysis can fail. In that situation, the analysis cannot 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 additional variables are not required.

A CP solver that would come close to this idea is HaifaCSP [38], 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.

Our empirical evaluation shows several new insights. First, compared to LCG, our LLG approach reduces the number of conflicts by at least 60% in a quarter of the instances, indicating that cutting planes reasoning is promising for constraint programming. Second, keeping the problem structure in the form of global constraints is useful because of the stronger propagation, as opposed to decomposing the problem into linear inequalities. Third, 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 by a factor of at least 3 in

50% of the instances, and by a factor of at least 12 in 25% of the instances.

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. It is satisfied by the assignment θ if and only if $\sum w_i \cdot \theta(x_i) \leq c$ is true. The propagator for this constraint performs bound propagation [6], 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 $\mathcal{D}(y)$.

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 [26], which are inspired by Conflict-Driven Clause Learning (CDCL) [21]. CDCL is a popular solver paradigm for propositional solvers. 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

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$(\mathcal{C}, T)$

One reason for the inability to construct an asserting linear constraint is the *rounding problem*. Due to this issue, 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, adapted from [26, Example 3.1], illustrates this issue.

▶ **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 [27, 36] (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 literals 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 Boolean variable 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 to 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 are atomic constraints and p is the propagated atomic constraint.

▶ **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 [10]. The inclusion into CP solvers became popular when g-nogoods were introduced [19], which are a conjunction or disjunction of $\langle x \neq v \rangle$ constraints. LCG [28, 36] 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 [20, Chapter 5], implemented by Moore [23, Chapter 5]. A c-nogood can combine arbitrary constraints, not just atomic constraints. Finally, Veksler and Strichman [38] 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 [8], and explanations can be fine-tuned to improve the quality of learned constraints [32, 13].

Solvers which 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, 16] to great effect. In this special case of ILPs, the rounding problem as described in Subsection 2.3 can be handled systematically. CutSat [17] and CutSat++ [7], which are CDCL-inspired solvers for general ILPs, restrict the search to

side-step the rounding problem. Last, the IntSat [25, 26] solver accepts that conflict analysis can fail and does not always learn a new constraint.

There is an intersting 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 [24] toolchain can convert CP models to efficiently solvable ILPs [5]. 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 [22], 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. LLG enhances LCG conflict analysis by incorporating cutting planes analysis, allowing learned constraints to capture linear relationships between variables—something that cannot be expressed using clauses.

Extending LCG to perform conflict analysis using linear inequalities aligns closely with the challenges addressed by IntSat [26] and HaifaCSP [38]. 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 $a \in \{2, 3\}, b \in \{1, 4\}, c \in \{2, 7\}$. The MiniZinc 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 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.

This leads to two modifications. First, rather than directly retrieving the currently conflicting constraint from $\mathcal{C}$,we attempt to derive a linear explanation for the conflict. If this is not feasible, we fallback to resolution. Similarly, rather than directly using RC, we now attempt to linearly explain each propagation. This requires a modification to the preconditions in lines 5 and 6 of Algorithm 1. In addition to these two conditions, we introduce an additional requirement: the combination step cannot proceed if RC cannot be linearly explained.

Finally, the resolution fallback is updated. In IntSat, the effectiveness of resolution is limited due to the absence of a clausal propagator. However, in CP, we can leverage LCG as a fallback, enabling native storage and propagation of learned clauses. When the solver fails to learn a linear constraint, it resorts to LCG, proceeding as though LLG were not present.

4.2 Linear explanations

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

Inferred constraints Firstly, it is important to highlight that the asserting inequality derived from LLG 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. Put differently, a linear explanation introduces (a part of) a linear decomposition of a constraint. This approach differs fundamentally from clausal explanations. In LCG, explanations describe the partial assignment responsible for a propagation or conflict. In contrast, LLG explanations are constraints that encapsulate both the conditions for propagation and the underlying linear relationship responsible for it.

Explanation signs Additionally, it is noteworthy that for a linear explanation to accurately capture a propagation, 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 include 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, given the explanation $\langle a \leq 2 \rangle \rightarrow b \geq 3$, we can define an auxiliary variable p_1 can be defined as $p_1 \iff \langle a \leq 2 \rangle$. We can now rewrite the explanation linearly as $b \geq 3 - M(1 - p_1)$. 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 [5].

4.3 Implementing auxiliary variables

Auxiliary variables cannot be constructed at the start of the search procedure, as it is unknown which auxiliaries will be necessary for the explanations. Thus, they must be introduced and subsequently propagated during the search. A key challenge is maintaining a consistent solver state at decision levels where these variables did not yet exist.

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. Thereby, we try to reach a state that is identical to the state if the auxiliary variable had been present from the start 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, we explicitly evaluate the conditions of auxiliary variables rather the auxiliary variable's bounds when checking for assertivity at any previous decision level. Although the auxiliary variables may not have been propagated yet, it is still possible to infer the truth value of the condition.

4.4 Examples of explanations

We have formulated linear explanations for several global constraints. This section provides a detailed exposition of some of these explanations. A comprehensive overview of all explanations is presented in Appendix A.

▶ **Example 7** (Explanation of $x_i \neq val$ for $Ax \neq b$)**.** The propagation of $x_i \neq val$ implies that $x_i < val \oplus x_i > val$ 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 (pick either explanation).

▶ **Example 8** (Explanation of $b \geq 1 \wedge c \geq 0 \rightarrow a \leq \lfloor ub(c)/lb(b) \rfloor$ 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:

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

▶ **Example 9** (Explanation of $\neg cond \rightarrow p \leq 0$ for $p \rightarrow cond$)**.** A noteworthy constraint is the reified constraint, which defines the implication $p \rightarrow cond$. Suppose $cond$ is false (i.e. it is conflicting). It then follows that $p \leq 0$, which is explained by $\neg cond$. In turn, we can explain $\neg cond$ with a linear constraint of the form $Ax \leq b$. However, using this explanation directly to explain the propagation of $p \leq 0$ would erroneously imply that $Ax \leq b$ must always hold. Instead, it must hold only when $p = 1$. Therefore, we incorporate p into the explanation:

$$Ax \leq b + M(1-p). \tag{2}$$

5 Experiments

We implemented our LLG approach in Pumpkin. The initial version of Pumpkin serves as the baseline LCG solver. We consider the number of conflicts as the main metric. This allows us to draw conclusions that are independent of the runtime and engineering issues.

We believe this fairly shows the potential of our LLG approach. With this in mind, we leave optimizing the implementation as future work. To further ensure that the results are because of 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. Our results demonstrate that:

1. LLG encounters significantly less conflicts than LCG (Section 5.2.1).
2. Learned inequalities indeed provide stronger reasoning than clauses (Section 5.2.2).
3. While the success rate of LLG analysis is relatively low and decreases with time (Section 5.2.3), the learned constraints are highly effective.
4. LLG is generally more effective compared to a linear decomposition (Section 5.3).
5. The presence of clausal propagation is instrumental for LLG, underscoring the benefits of utilizing LCG as a fallback (Section 5.4).

These experiments provide a deeper insight into cutting planes conflict analysis in the context of constraint programming.

5.1 Experimental setup

Dataset selection The representative dataset used for the experiments is constructed by combining *all* models from the MiniZinc Challenges [34, 33] and Benchmarks[1] libraries. Each model is subjected to the following criteria and transformations:

- **Model selection:** For models obtained from the Minizinc Challenges, only problems that at least one finite domain solver has successfully solved within the 20-minute time limit are included. This is because the experiments exclude instances that reach their timeout, as many metrics become incomparable in such cases. Furthermore, models containing floating-point numbers or unbounded integers are excluded. To maximize dataset diversity, no more than 10 instances per model are included. If a model exceeds this limit, 10 instances are randomly sampled from the available data files. Finally, only problems utilizing fixed-search branching are considered, ensuring that any variation in the number of conflicts is solely due to differences in learning.
- **Constraint decomposition:** Problems are decomposed into fundamental global constraints, namely, 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.

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.

Hardware All experiments were conducted on the DelftBlue[11] compute cluster, using 8GB of memory and a 1-hour timeout per instance.

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

5.2 LLG compared to LCG

5.2.1 Reduction of conflicts

This experiment evaluates the ratio of conflicts that are encountered in LLG compared to LCG across the instances that, (1) have been solved within the time-limit for both LLG and LCG, and, (2) have been able to learn at least one linear inequality. Of the 952 instances, LLG can solve 624 instances within the time-limit, whereas LCG can solve 663 instances within the time-limit. This difference can be attributed to the significantly higher number of out-of-memory errors for LLG, given its inefficient implementation. Additionally, in the case of LLG, approximately **70%**, or **443** of successful instances succeeded in learning any inequality. This shows that, with the current state of LLG, not all instances benefit from its linear conflict analysis.

The results in the boxplot in Figure 1 reveal several key trends. Firstly, 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 may learn very few linear constraints, or may learn linear constraints that propagate similar bounds as learned clauses. Lastly, the upper 75% to 90% percentile range highlights cases where the number of conflicts increase marginally. This phenomenon can be attributed to some learned constraints that are weaker compared to the clauses that could have been learned.

Our initial hypothesis asserted 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 ($117/52 \approx \mathbf{2.3}$ instances per model). In comparison, a total of 574 instances from 171 models successfully complete for both LCG and LLG ($547/165 \approx \mathbf{3.3}$ instances per model): the top-performing instances are even more varied than the full dataset. Consequently, our initial hypothesis is refuted; a wide variety of problems benefit from LLG.

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

Lastly, although these experiments were conducted using an unoptimized implementation, 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%.

5.2.2 Strength of learned inequalities

Since LLG was introduced based on the proposition that linear constraints may provide stronger reasoning than clauses, it is essential to verify whether this assumption is supported by experimental results. To this end, we compared two program variants: a standard LLG program and an alternative version in which linear analysis is performed, but instead of adding learned linear constraints, the fallback clauses are introduced into the model.

For the standard LLG program, we record for each propagation triggered by a learned linear constraint whether the fallback clause—had it been learned instead—would have resulted in the same propagation. Similarly, for the alternative program, we record for each propagation triggered by a learned clause that was generated in cases where a linear

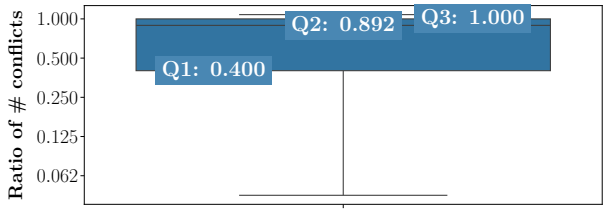


Figure 1 Boxplot showing the ratio of conflicts in LLG relative to LCG (lower = better). The boxplot displays the Q1, Q2 and Q3 quartiles, with whiskers extending to the 10% and 90% percentiles. A logarithmic scale is used for equal spacing of ratios. Only instances where both LCG and LLG successfully completed and at least one linear constraint was learned, are considered. LLG encounters significantly fewer conflicts than LCG for 50% of the instances, slightly fewer conflicts for 25% of the instances while for the remaining instances, the increase in conflicts is marginal if any.

constraint could have been learned instead, whether the linear constraint—had it been learned instead—would have produced the same propagation.

The results (not in figure) 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 (not in figure) that for 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.

5.2.3 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. Therefore, three subsets of instances were selected: 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 an increasing number of analysis failures due to invariant violations increase 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 the 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 propagations 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.

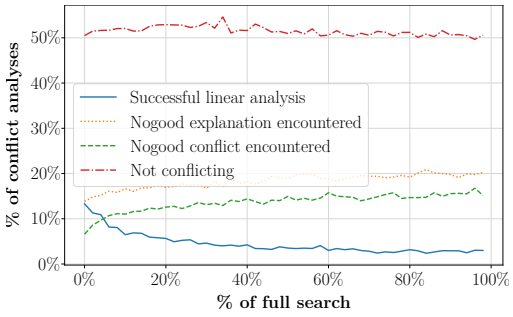
First, we found no clear correlation between the slack of a conflict explanation and the proportion of these explanations that eventually led to a learned constraint, contrary to our expectation that large negative slacks are more likely to succeed. The only necessary condition for a conflict explanation 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 slacks, prioritizing those that are more likely to contribute to the derivation of a learned constraint.

5.3 LLG compared to linear decomposition

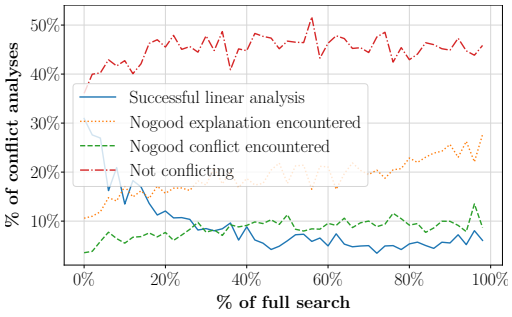
In addition to comparing LLG with LCG, it is also valuable to evaluate Pumpkin's decomposition as described in Section 5.1 against a decomposition consisting only of linear inequalities. This evaluation shows that using CP propagators—even though not all its linear explanations are equally successful—is 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 [2]. 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.

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

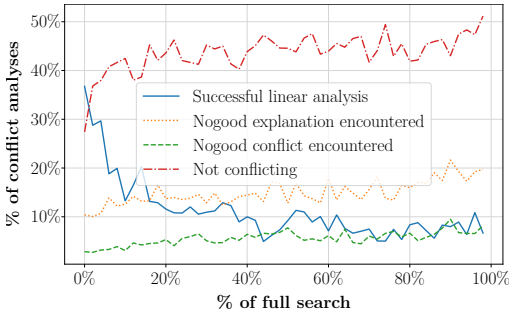
[2] `https://github.com/MiniZinc/libminizinc/tree/master/share/minizinc/linear`



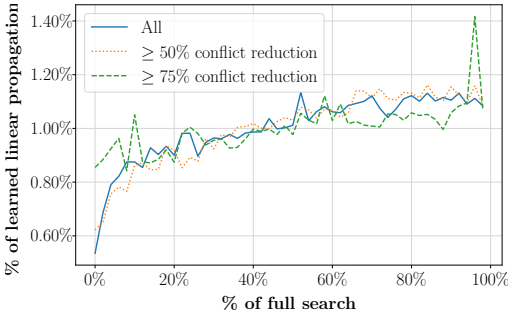
(a) All instances



(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 nogoods encountered increases. 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 exhibit the highest concentration of constraint propagations. This can be explained by the cumulative effect of both newly introduced and previously learned constraints.

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.

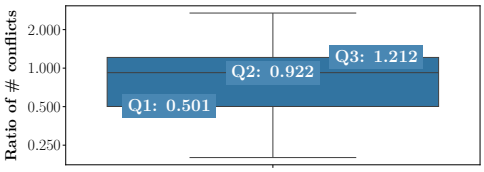
These increases in conflicts can be attributed to two primary factors. Firstly, the preprocessing optimizations performed by the linear decomposition can result in the generation of smaller linear problem instances compared to those produced by Pumpkin's decomposition. Secondly, some of Pumpkin's propagators 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.

5.4 LLG without clause learning

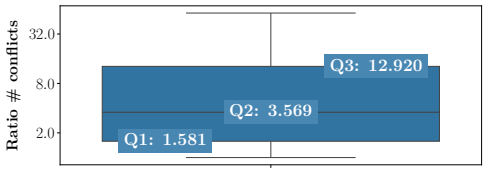
Previous experiments have shown that a key reason for failing to learn linear constraints during LLG conflict analysis is encountering nogoods. Therefore, we investigate the consequences of only propagating a learned clause once, without storing it in the constraint database. The

results, presented in Figure 3b, indicate that omitting clause storage leads to a significant increase in conflicts. This observation underscores that even though encountering nogoods resulted 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 [26]. 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.



(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.

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 [21], IntSat [26], HaifaCSP [38], and Lazy Clause Generation [36] 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 dynamically introduces new auxiliary variables while maintaining a consistent solver state. Our experimental analysis of LLG shows that learning linear constraints leads to a substantial reduction in the number of conflicts over solely learning clauses. Although the success rate of linear conflict analysis is relatively low, the impact of learned linear constraints on overall solver performance is significant. When compared to a linear decomposition with the same conflict analysis, we also observe a considerable reduction in the number of conflicts encountered. Furthermore, our experiments highlight that clausal propagation is still important, even when using conflict

analysis based on cutting planes. To conclude, we believe our approach shows potential in obtaining much more effective constraint solving.

There are several promising avenues for further research of the LLG algorithm. First, it would be interesting to consider the impact on a wider range of global constraints. Second, we could improve our linear explanations based on decomposition optimisations [5]. Third, studying different branching strategies that utilise information from cutting planes conflict analysis could lead to additional improvements. Lastly, engineering improvements would be of interest. We believe these research directions offer valuable opportunities to further enhance the effectiveness and applicability of solvers.

References

1 MiniZinc Challenge 2024 results [online]. URL: https://www.minizinc.org/challenge/2024/results/.

2 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.

3 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.

4 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.

5 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.

6 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.

7 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.

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

9 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.

10 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.

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

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.

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

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

16 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`.

17 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`.

18 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.

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

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

21 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`.

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

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

24 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`.

25 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. URL: `http://link.springer.com/10.1007/978-3-319-10428-7_42`, `doi:10.1007/978-3-319-10428-7_42`.

26 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`.

27 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`.

28 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`.

29 Laurent Perron and Frédéric Didier. Cp-sat [online]. URL: `https://developers.google.com/optimization/cp/cp_solver/`.

30 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.

31 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.

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

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

34 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. URL: https://onlinelibrary.wiley.com/doi/10.1609/aimag.v35i2.2539, doi:10.1609/aimag.v35i2.2539.

35 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.

36 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.

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

38 Michael Veksler and Ofer Strichman. Learning general constraints in CSP. *Artificial Intelligence*, 238:135–153, September 2016. URL: https://linkinghub.elsevier.com/retrieve/pii/S0004370216300650, doi:10.1016/j.artint.2016.06.002.

39 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	$lb(Ax) = ub(Ax) = b$	$Ax \geq b + 1 - Mp$	
$a = \|b\|$	$a \geq 0$	-	$a \geq 0$	• $p_1 \iff b \leq 0$ • $p_2 \iff b \geq 0$
	$a \geq lb(b)$	$lb(b) > 0$	$a \geq b$	
	$b \geq -ub(a)$	-		
	$a \geq \|ub(b)\|$	$ub(b) < 0$	$a \geq -b$	
	$b \leq ub(a)$	-		
	$a \leq \max(\|lb(b)\|, \|ub(b)\|)$	-	• $a + b \leq M(1-p_1)$ • $a - b \leq M(1-p_2)$	
	$b \leq -lb(a)$	$ub(b) \leq 0$		
	$b \geq lb(a)$	$lb(b) \geq 0$		
$max(A) = b$	$a_i \leq ub(b)$	-	$a_i \leq b$	• $\forall i \in \|A\|$: auxiliary p_i • $\sum p_i = 1$
	$b \geq \max_{a_i \in A}(lb(a_i))$	-		
	$b \leq \max_{a_i \in A}(ub(a_i))$	-	$rhs \leq a_i + M(1-p_i)$	
	$a_i \geq lb(b)$	a_i only for $ub(a_i) \geq lb(b)$		
	$b \leq ub(a_i)$	$p_i = 1$		
	$a_i \geq lb(b)$	$p_i = 1$		
$a \times b = c$	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: $rhs \geq 0$	$lb(num) \geq 0 \wedge lb(den) \geq 0$	$rhs \geq 0 - Mn_{num} - Mn_{den}$	• $n_{num} \iff num < 0$ • $n_{den} \iff den < 0$ • $p_{den} \iff den > 0$ • $rhs_{lb} \iff rhs \geq lb(rhs)$ • $den_{ub} \iff den \leq ub(den)$
	$rhs \leq ub(num)/lb(den)$	$ub(num) \geq 0 \wedge ub(den) \geq 0$	$lb(rhs) \cdot den \leq num + M(1-rhs_{lb}) + Mn_{num}$ Swap rhs and den for alternative	
	$den \leq ub(num)/lb(rhs)$	$lb(rhs) \geq 0 \wedge ub(num) \geq 0$		
	$num \geq lb(rhs) \cdot lb(den)$	$lb(rhs) \geq 0 \wedge ub(den) \geq 0$		
	$rhs \geq lb(num)/ub(den)$	$lb(num) \geq 0 \wedge ub(den) \geq 0$	$ub(den) \cdot rhs + den - 1 \geq num - M(1-den_{ub}) - Mn_{num} - M(1-p_{den})$ Swap rhs and den for alternative	
	$num \leq (ub(rhs)+1) \cdot ub(den) - 1$	$ub(den) \geq 0 \wedge ub(rhs) \geq 0$		
$A[idx] = b$	$b \leq \max_{a_i \in A if i \in idx}(ub(a_i))$	-	$b \leq a_i + M(1-p_i)$	• $\forall i \in \|A\|$: auxiliary p_i • $\sum p_i = 1$ • $\sum i \cdot p_i = idx$
	$a_i \leq ub(b)$	$idx = i$		
	$b \geq \min_{a_i \in A if i \in idx}(lb(a_i))$	-	$b \geq a_i + M(1-p_i)$	
	$a_i \geq lb(b)$	$idx = i$		
	$p_i \geq 1$	$idx = i$	$\sum i \cdot p_i = idx$	
	$p_i \leq 0$	$idx \neq i$		
$p \rightarrow cond$	$p \leq 0$	$\neg cond$	$expl(cond) \rightarrow Ax \leq b$ $Ax \leq b + M(1-p)$	-
	Conflict	$p \geq 1 \wedge \neg cond$		