Explanation-Based Propagators for the Table Constraint

Comparing Eager vs. Lazy Explanations in Lazy Clause Generation Solvers

Markas Aisparas[1]
Supervisor(s): Emir Demirović[1]**, Maarten Flippo**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 26, 2025

Name of the student: Markas Aisparas
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Maarten Flippo, Benedikt Ahrens

Abstract

The table constraint is a fundamental component of constraint programming (CP), used to explicitly define valid value combinations for variables. In modern *Lazy Clause Generation (LCG)* solvers, constraints rely on explanations to justify value removals and enable efficient conflict analysis through nogood generation. However, table constraints have primarily been implemented using direct SAT encodings, such as the state-of-the-art Bacchus method, rather than explanation-based approaches. This paper introduces two types of explanation-based propagators for table constraints: *eager explanations*, generated during propagation, and *lazy explanations*, which adapt explanations to specific conflicts for more general nogoods. Experiments on MiniZinc benchmarks show that *Optimized (Lazy)* explanations reduce conflicts by an average of **23%** across all problems and up to **64%** for specific instances compared to the Bacchus encoding, while also reducing learned clause length by **46%**. Although the current implementations incur a runtime penalty of up to **2x**, these findings highlight the potential of explanation-based propagators to improve conflict resolution and search efficiency with further optimizations.

1 Introduction

Constraint Programming (CP) is a powerful technique for solving combinatorial problems in many areas. It works by exploring the space of possible solutions and using logical inference, facilitated by *propagators*, to rule out values that cannot be part of any valid solution. CP is widely used because of its flexibility and effectiveness in solving real-world problems.

A major advancement in constraint programming is *Lazy Clause Generation (LCG)* [18, 8], which enhances CP by combining it with techniques from *SAT solving*—a method for determining whether a logical formula can be satisfied. In LCG, propagators not only remove invalid values from variable domains but also generate *explanations*. Explanations justify why certain values are removed and enable the solver to dynamically encode parts of the problem into SAT form. More importantly, explanations are used during conflict analysis to construct *nogoods*, which are additional constraints that prevent the solver from revisiting the same conflict. This technique, originally adapted from SAT and formally knows as Conflict-Driven Clause Learning (CDCL)[15], uses nogoods to improve solver efficiency by pruning large portions of the search space without removing valid solutions.

The quality of explanations is crucial to solver performance. Better explanations produce more *general nogoods*—nogoods that rule out larger regions of the search space, making the solver more efficient. Explanations can be generated either *eagerly*, at the time a value is removed during propagation, or *lazily*, at the time of conflict [7]. An interesting observation is that, since lazy explanations are generated at the time of conflict, they have access to the nogood that is being generated. This presents the opportunity to adapt the explanation in such a way that it creates a more general nogood.

This paper focuses on the *table constraint*, which explicitly defines valid combinations of values for a group of variables. The table constraint is widely used because of its versatility: it can model a broad range of combinatorial and real-world problems which would otherwise be difficult to express using logical relationship[13]. However, while propagators for other global constraints, such as *alldifferent*[6] and *cumulative*[19], have been designed to exploit LCG's capabilities, table constraint propagators have received little attention. Instead, table constraints are often handled by direct SAT encodings [2], which fail to take full advantage of LCG's explanation mechanisms.

To illustrate the role of explanations for table constraints, consider the following example.

Example 1.1. Suppose in the current state of the solver we have three variables $x \in \{1, 2\}$, $y \in \{3\}$, and $z \in \{1\}$, and the valid combinations of values are explicitly defined in the table below:

x	y	z
1	1	2
1	2	3
2	3	1

Looking at the table of valid combinations, we see that no valid combination allows $x = 1$. The first valid combination, $(1, 1, 2)$, requires $y = 1$ and $z = 2$, but these values are not in the current domains. Similarly, the second valid combination, $(1, 2, 3)$, requires $y = 2$ and $z = 3$, which are also unavailable. Since both supporting tuples for $x = 1$ are invalid, $x = 1$ is removed. We can come up with four equally valid explanations to justify this removal by identifying combinations of literals over y and z that invalidate the two supporting tuples:

$$(1)\ [y \neq 1] \wedge [y \neq 2], \quad (2)\ [z \neq 2] \wedge [z \neq 3], \quad (3)\ [y \neq 1] \wedge [z \neq 3], \quad (4)\ [z \neq 2] \wedge [y \neq 2].$$

Later, when the solver encounters a conflict, the choice of explanation directly affects the quality of the nogood generated. At the time of propagation, eager explanations are generated without knowledge of future conflicts, limiting their generality. By generating explanations *lazily* during conflict analysis, the solver can adapt explanations to the current nogood, producing more general nogoods.

In this paper, we address the lack of specialized LCG propagators for table constraints by developing two eager explanation algorithms (*Naive* and *Greedy*) and one lazy explanation algorithm (*Optimized*). The *Naive* approach constructs explanations by arbitrarily adding literals until all supporting tuples are invalidated. The *Greedy* approach improves on this by selecting literals that invalidate the largest number of tuples, reducing the size of explanations. The *Optimized (Lazy)* approach further refines this process by prioritizing literals that do not increase the size of the nogood, adapting explanations to the current conflict during conflict analysis.

Experiments on the MiniZinc benchmark suite [20] demonstrate that lazy explanations significantly improve solver performance. Compared to the state-of-the-art Bacchus encoding, lazy explanations reduce conflicts by an average of **23%** across all problems and up to **64%** for specific instances, while also reducing average nogood (learned clause) length by **46%**. However, these benefits come at the cost of runtime, with Bacchus encoding being up to **2x faster** in some cases. These results highlight the potential of explanation-based propagators to improve solver efficiency while identifying runtime as a key area for future optimization.

The remainder of the paper is structured as follows: Section 2 covers the preliminaries. Section 3 reviews related work on table constraints and explanation generation. Section 4 details the proposed eager and lazy explanation algorithms. Section 5 presents the experimental setup and results. Section 6 discusses ethical considerations. Finally, Section 7 summarizes the findings and outlines future research directions.

2 Preliminaries

2.1 Constraint Satisfaction Problems

A *constraint satisfaction problem (CSP)* is defined as a triple $P = (X, D, C)$, where X is a finite set of variables, D is a set of domains where each $D(x)$ specifies the finite set of values that variable $x \in X$ can take, and C is a finite set of constraints that restrict the values that a subset of variables $S \subseteq X$ can take simultaneously.

The goal of solving a CSP is to find an assignment of values to all variables X, such that each variable $x \in X$ is assigned a value in its domain $D(x)$, and all constraints $c \in C$ are satisfied. If such an assignment exists, the problem is satisfiable; otherwise, it is unsatisfiable.

A related class of problems is the *constraint optimization problem (COP)*, where the objective is to find a satisfying assignment that also optimizes a given objective function. For example, a COP may require finding the assignment that minimizes the total cost or maximizes a utility function.

CSPs provide a general framework for modeling a wide range of combinatorial and real-world problems, including scheduling, planning, and resource allocation.

2.2 Constraint Programming

Constraint Programming (CP) is a computational framework for solving constraint satisfaction problems (CSPs) and constraint optimization problems (COPs). It combines recursive search with propagation to systematically reduce the search space while maintaining consistency across constraints.

Constraints in CP are enforced through *propagators*, which are functions that reduce the domains of variables by removing values that cannot participate in any valid solution satisfying the constraint. For example, a propagator for the constraint $x + y \leq 10$ might remove values from the domains of x and y that violate this condition, given the current domains.

A central concept in CP is *generalized arc consistency (GAC)*. A constraint is considered GAC if, for every variable in the constraint, each value in its domain participates in at least one valid solution to the constraint, given the current domains of the other variables [12]. Propagators are designed to enforce GAC (or a weaker form of consistency), thereby simplifying the problem by systematically reducing variable domains while preserving all solutions.

2.3 Table Constraint

The *table constraint* is a global constraint that explicitly defines the allowed combinations of values for a subset of variables $S \subseteq X$ in a CSP. It is represented by a set of valid tuples, where each tuple specifies a valid assignment of values to the variables in S that satisfies the constraint.

Formally, let $S = \{x_1, x_2, \ldots, x_k\}$, where each variable $x_i \in S$ has a domain $D(x_i)$. A table constraint is defined by a set of tuples $T \subseteq D(x_1) \times D(x_2) \times \cdots \times D(x_k)$. Each tuple $t = (v_1, v_2, \ldots, v_k) \in T$ represents a valid assignment $x_1 = v_1, x_2 = v_2, \ldots, x_k = v_k$. If a tuple is not in T, it is considered invalid.

2.3.1 Supports

For each variable-value pair (x_i, v), a *support* is a tuple $t \in T$ that includes $x_i = v$ and satisfies the table constraint [12]. The set of all supports for (x_i, v) is denoted as:

$$\text{Supports}(x_i = v) = \{t \in T \mid t[x_i] = v\},$$

where $t[x_i]$ represents the value assigned to x_i in tuple t. During propagation, supports are used to determine whether a value can remain in the domain of a variable. If all supports for (x_i, v) are invalid due to changes in the domains of other variables in S, the value v is removed from $D(x_i)$.

2.3.2 GAC Propagator

Propagators for table constraints enforce *generalized arc consistency (GAC)*. A table constraint is GAC if, for every variable $x_i \in S$ and every value $v \in D(x_i)$, there exists at least one support $t \in T$ such that $t[x_i] = v$ and t satisfies the table constraint given the current domains of the other variables. If no such support exists for (x_i, v), v is removed from $D(x_i)$.

Example 2.1. Consider a table constraint over three variables x, y, and z, where $S = \{x, y, z\}$, with the following domains:

$$D(x) = \{1, 2\}, \quad D(y) = \{3\}, \quad D(z) = \{1\}.$$

The table constraint is defined by the set of valid tuples:

$$T = \{(1, 2, 1), (1, 3, 2), (2, 3, 1)\}.$$

For $x = 1$, the supporting tuples are $(1, 2, 1)$ and $(1, 3, 2)$. However:

- The tuple $(1, 2, 1)$ requires $y = 2$, which is not in $D(y) = \{3\}$.
- The tuple $(1, 3, 2)$ requires $z = 2$, which is not in $D(z) = \{1\}$.

Since both supporting tuples for $x = 1$ are invalid, $x = 1$ is removed from $D(x)$ to ensure GAC. The updated domain becomes $D(x) = \{2\}$.

2.4 SAT

The *Boolean satisfiability problem (SAT)* is a specialized form of a constraint satisfaction problem (CSP) where variables are Boolean. The goal of SAT is to find an assignment of truth values to variables that satisfies all given constraints or to determine that no such assignment exists.

In SAT, constraints are typically expressed in *conjunctive normal form (CNF)*, where the formula is a conjunction (logical AND) of clauses, and each clause is a disjunction (logical OR) of literals. A *literal* is either a Boolean variable x or its negation $\neg x$.

Example 2.2. The formula: $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$ is a SAT instance in CNF with two clauses: $(x_1 \vee \neg x_2)$ and $(\neg x_1 \vee x_3)$.

2.4.1 Unit Propagation

SAT solvers use *unit propagation*, a technique analogous to propagation in CP. A clause becomes a *unit clause* when all but one of its literals are assigned values that make the clause false. In this case, the remaining unassigned literal must be assigned a value that satisfies the clause.

Example 2.3. Consider the clause $(x_1 \vee x_2)$ and assume x_2 is false. If x_1 is unassigned, the clause reduces to (x_1), requiring $x_1 =$ true to satisfy it. Unit propagation iteratively simplifies the formula by assigning

values to variables until no further inferences can be made.

2.5 Lazy Clause Generation (LCG)

Lazy Clause Generation (LCG)[18] combines the global propagator capabilities of constraint programming (CP) with the SAT-style clause representation, enabling the lazy encoding of problems into SAT[8]. Additionally, LCG incorporates conflict-driven[15] learning by dynamically generating and adding *nogoods*—clauses that represent conflicts—to the solver. These nogoods enhance the solver's ability to prune the search space effectively.

2.5.1 Atomic Constraints

At the core of LCG is the representation of *atomic constraints*, which are logical conditions of the form $\langle x \diamond v \rangle$, where x is an integer variable, v is a constant, and $\diamond \in \{\leq, \geq, =, \neq\}$.

2.5.2 Integer Variable Encoding

In LCG, integer variables are encoded as Boolean literals representing their atomic constraints. For a variable x with domain $D(x) = \{1, 2, \ldots, n\}$, the encoding introduces:

- Boolean literals $[x = v]$ for each $v \in D(x)$, representing $x = v$,
- Boolean literals $[x \leq v]$ for each $v \in D(x)$, representing $x \leq v$.

Example 2.4. The domain $D(x) = \{1, 2, 3\}$ is represented by the literals:

$$[x = 1], \quad [x = 2], \quad [x = 3], \quad [x \leq 1], \quad [x \leq 2], \quad [x \leq 3].$$

Different assignments of true or false to these literals correspond to different possible states of the domain. For instance, setting $[x = 1] =$ true, $[x = 2] =$ false, and $[x = 3] =$ false represents the domain $D(x) = \{1\}$.

2.5.3 Eager (Forward) Explanations

When a propagator removes a value from a variable's domain, it generates an *explanation* to justify the removal. Explanations describe the conditions under which the removal occurred in terms of Boolean literals representing variable domains. Formally, an explanation for the removal of $x = v$ is represented as:

$$\bigwedge_{r \in R} r \implies [x \neq v],$$

where R is the set of Boolean literals (from now on referred to as *reasons*) responsible for the removal, and $[x \neq v]$ is shorthand for $\neg[x = v]$, indicating that $x = v$ is no longer valid.

When an explanation is generated, it is added to the underlying SAT solver in clause form and used for unit propagation.

Example 2.5. Continuing the example 2.1, with $D(x) = \{1, 2\}$, $D(y) = \{3\}$, $D(z) = \{1\}$, and valid tuples $T = \{(1, 2, 1), (1, 3, 2), (2, 3, 1)\}$, the value $x = 1$ was removed. The two supporting tuples for $x = 1$, $(1, 2, 1)$ and $(1, 3, 2)$, are invalid because $(1, 2, 1)$ requires $y = 2$ (not in $D(y)$) and $(1, 3, 2)$ requires $z = 2$ (not in $D(z)$). The explanation for this removal is thus:

$$[y \neq 2] \wedge [z \neq 2] \implies [x \neq 1].$$

This explanation ensures that the removal of $x = 1$ is logically justified in terms of the invalidation of its supporting tuples.

2.5.4 Nogood Generation

When a conflict is detected—where no assignment satisfies all constraints—the solver performs *conflict analysis* to generate a nogood. Suppose a conflict arises between the following two literals:

$$[x \neq 1] \quad \text{and} \quad [z < 4].$$

Starting with these conflicting literals, the solver uses the *1-Unique Implication Point (1UIP)*[17] algorithm to recursively expand the nogood by substituting literals in the nogood with their explanations. The explanation for $[x \neq 1]$ is $[x \neq 2] \wedge [z \neq 2]$. Substituting this explanation into the nogood gives:

$$[x \neq 2] \wedge [z \neq 2] \wedge [z < 4].$$

The resulting nogood is minimal yet sufficient to prevent the conflict from reoccurring. This nogood is added to the SAT solver in clause form, ensuring the conflicting region of the search space is avoided in future exploration.

2.6 Lazy (Backwards) Explanations

Lazy explanations (also referred to as backwards explanations) are an alternative explanation method to eager explanations. While eager explanations are produced immediately when values are removed during propagation, lazy explanations are generated after a conflict occurs. This deferred approach enables the solver to generate explanations that lead to more optimal nogood generation. [7]

2.6.1 Strength of Backwards Explanation Algorithms

Any algorithm for generating lazy explanations must be as strong as the propagator that initially removed a value. This means that any value removal during propagation must be explainable lazily at the time of conflict. For the table constraint, the eager propagation algorithm ensures generalized arc consistency (GAC) by guaranteeing that every value in the domain of a variable is supported by at least one valid tuple. Consequently, any lazy explanation algorithm for the table constraint must be able to generate an explanation for a value that was removed by a GAC propagator.

2.6.2 Nogood Simplification (Subsumption)

In the process of generating nogoods, *nogood simplification* [7] (also referred to as subsumption) plays a crucial role in improving their quality. Nogood simplification reduces the size of a nogood while preserving its logical correctness. A smaller nogood generalizes better, prunes larger portions of the search space, and improves solver efficiency.

Formally, a nogood is simplified when one or more literals can be removed or replaced without changing its logical effect. Simplification generally occurs in two cases:

1. *Combining literals:* If a literal $[y \neq d]$ is added to the nogood and $d = v$, where $[y \leq v]$ is already in the nogood, the two literals can be combined into $[y \leq v - 1]$, reducing the range of valid values for y without increasing the nogood size.

2. *Subsumed literals:* If $d > v$, adding $[y \neq d]$ is redundant because it is subsumed by $[y \leq v]$, which already excludes d.

This ability to simplify nogoods is particularly advantageous when explanations are generated lazily during conflict analysis. If multiple explanations are possible, lazy explanation algorithms can choose those that take advantage of simplification opportunities, ensuring that the generated nogood is as compact and general as possible.

The key observation that facilitates our focus on creating explanations that minimize the number of literals in the nogood is that this strengthens the unit propagation of the underlying SAT solver. Since a clause can only unit propagate if all but one of its literals are assigned values, the fewer literals there are in the clause, the more likely it is to propagate.

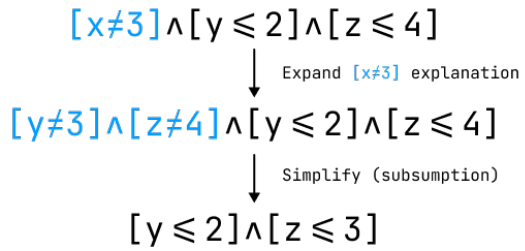

Figure 1: An example of nogood simplification where the literal $[y \neq 3]$ is subsumed by $[y \leq 2]$, and $[z \neq 4]$ is combined with $[z \leq 4]$ to form $[z \leq 3]$.

Figure 1 demonstrates an example of nogood simplification. Starting with the nogood $[x \neq 3] \wedge [y \leq 2] \wedge [z \leq 4]$, the literal $[x \neq 3]$ is explained as $[y \neq 3] \wedge [z \neq 4]$. After substitution and simplification, the resulting nogood becomes $[y \leq 2] \wedge [z \leq 3]$, reducing its size and increasing its generality.

2.7 SAT Encoding for Table Constraints

For table constraints, current state-of-the-art *Lazy Clause Generation (LCG)* solvers, instead of using propagators that utilize explanations to dynamically add clauses to the underlying SAT solver, directly encode the entire constraint into SAT clauses at the start of solving using the Bacchus encoding method [2].

2.7.1 Boolean Variables

On top of the $[x_i = a]$ variables introduced by LCG for each variable x_i and each value $a \in D(x_i)$, the Bacchus encoding introduces auxiliary variables τ_j ($j \in \{1, \ldots, m\}$) for each of the m tuples in the table. τ_j represents whether the j-th tuple is currently valid.

2.7.2 Clauses in the Encoding

The encoding then ensures consistency between variable assignments and tuple validity by introducing two types of clauses:

1. For each tuple τ_j and for every variable x_i in the scope of the constraint, we ensure that t_j becomes false if any x_i does not match the value specified by τ_j:

$$\forall j, \forall i, \; [[x_i = \tau_j[i]]] \vee \neg t_j.$$

   Here, $\tau_j[i]$ represents the value of x_i in tuple τ_j.

2. For each variable x_i and each value $a \in D(x_i)$, we ensure that $[x_i = a]$ becomes false if all tuples supporting $x_i = a$ are invalid:

$$\forall i, \forall a, \; \neg[[x_i = a]] \vee \bigvee \{t_j | \tau_j[i] = a\}.$$

   This ensures that $[[x_i = a]]$ is true only if there is at least one valid tuple τ_j where $x_i = a$.

2.7.3 Strengths and Limitations

The Bacchus encoding offers several significant strengths that make it a robust method for handling table constraints. One of its key advantages is its compact representation of tuple validity. By introducing a single auxiliary variable τ_j for each tuple, the encoding allows nogoods to directly reference and invalidate specific tuples concisely, resulting in a highly efficient SAT representation.

However, the Bacchus encoding also has notable weaknesses. By encoding all clauses upfront, it cannot dynamically guide the solver toward promising parts of the search space during solving, as it lacks the flexibility of explanation-based methods. The order in which clauses propagate via unit propagation is arbitrary and does not adapt based on solver state. Additionally, the use of tuple variables (τ_j) in nogoods can limit their generality compared to explanation-based approaches, which use $[x_i = a]$ literals. While τ_j variables are specific to a single table constraint, $[x_i = a]$ literals can capture interactions across multiple constraints, enabling nogoods to prune larger portions of the search space.

The Bacchus encoding remains a strong baseline for evaluating the explanation-based propagators developed in this paper. Its compactness, simplicity, and widespread adoption underscore its effectiveness, while its limitations provide a valuable benchmark to assess the potential benefits of explanation-based methods in reducing conflicts and improving nogood generality.

3 Related Work

In this section, we group related work into three main categories: (1) *finite domain (FD) propagators* for table constraints, (2) *lazy clause generation (LCG) propagators* for other global constraints, and (3) *SAT encodings* for table constraints. We conclude by positioning our work as addressing the unexplored intersection of LCG and table constraint propagation.

3.1 Finite Domain Propagators for Table Constraints

The table constraint is a fundamental and versatile constraint in constraint programming, as it explicitly enumerates valid combinations of values for a group of variables. Extensive research has been conducted on *finite domain (FD) propagators* that enforce *generalized arc consistency (GAC)* by efficiently filtering values unsupported by any valid tuple.

Various FD-based propagators have been developed to achieve GAC with different trade-offs between memory usage and filtering speed. *STR2* improves upon earlier techniques using *bitwise filtering* and compact *support data structures*, enabling efficient and incremental propagation [11]. *Compact-Table* focuses on achieving GAC through the use of bitwise operations and efficient support data structures, optimizing the filtering process without compressing the table representation itself [5]. *Smart-Table*, on the other hand, enhances the representation of table constraints by employing a compact encoding where multiple tuples are compressed into *smart tuples*. This reduces memory usage and can improve propagation efficiency by minimizing the size of the table representation [14].

While these FD-based propagators have proven highly effective in traditional CP solvers, they are not directly compatible with *lazy clause generation (LCG)* solvers, which require explanations for value removals. Explanations are critical for conflict-driven clause learning, as they enable the solver to construct nogoods that guide the search process and prevent redundant exploration. Thus, adapting table propagators to generate explanations is necessary for integrating their benefits into LCG solvers.

3.2 LCG Propagators for Other Global Constraints

Lazy clause generation combines CP's global constraint propagation with conflict-driven clause learning techniques from SAT solving. Propagators in LCG are enhanced to generate *explanations* for value removals, which are later used to produce nogoods during conflict analysis. This approach has been applied to various global constraints, such as *alldifferent*, where explanations are generated using Hall intervals and matching theory to justify value removals [6], and *cumulative*, where explanations are derived from reasoning over resource usage and scheduling conflicts [19].

These propagators demonstrate the power of explanation-based propagation in LCG, where dynamically generated nogoods improve search efficiency by pruning the search space. A notable advancement is the use of *lazy explanations* (or backward explanations), where explanations are generated during conflict analysis rather than at propagation, which allows adapting explanations to the current nogood.

While lazy explanations have been successfully applied to various constraints, their application to the table constraint remains unexplored, representing a key gap in the current state of the art[7].

3.3 SAT Encodings for Table Constraints

In the absence of dedicated LCG propagators, table constraints are often handled through *SAT encodings*. Bacchus introduced a widely adopted method for encoding table constraints into SAT by representing valid tuples as auxiliary Boolean literals and maintain consistency between variable assignments and valid tuples via *clause constraints* [2]. This approach has been integrated into state-of-the-art solvers such as *Chuffed* [4], which rely on efficient unit propagation and conflict-driven clause learning to solve the SAT-encoded problem.

While SAT encodings provide a simple and effective method for handling table constraints, they encode the entire constraint upfront, missing the opportunity to dynamically generate explanations during solving. In contrast, LCG propagators allow explanations to be produced on demand, enabling more flexible approach.

3.4 Positioning Our Work

This paper addresses the unexplored intersection of *LCG propagators* and the *table constraint*. While LCG has shown significant success with other global constraints, no explanation-based propagators exist for table constraints. In this work, we explore both *eager explanations*, which are generated immediately during propagation, and *lazy explanations*, which are generated at the time of conflict with access to the current nogood. By introducing these two explanation-based propagators, we aim to evaluate whether lazy explanations can take advantage of the conflict context to produce more general nogoods and improve solver performance over direct SAT encoding.

4 Main Contribution

In this section, we focus on developing explanation algorithms for table constraints in a LCG solver. The goal is to produce explanations that minimize the size of resulting nogoods, enabling more efficient conflict resolution and better pruning of the search space.

The remainder of this section details these contributions. We begin by constructing a base propagator (Section 4.1) that enforces generalized arc consistency (GAC) without generating explanations. Next, we introduce an algorithm to generate naive eager explanations (Section 4.2), which provides explanations for all values removed by the base propagator. To improve on this, we propose a greedy eager explanation algorithm (Section 4.3) that focuses on reducing the size of individual explanations. Finally, we present an optimized lazy explanation algorithm (Section 4.4), which builds on the greedy explanation algorithm and utilizes the current nogood to generate explanations that directly minimize the size of the nogood itself.

4.1 Base Propagator

As a foundation for our explanation-generating methods, we introduce the base propagator, whose purpose is to enforce generalized arc consistency (GAC) for table constraints by removing invalid values from variable domains. Its role is solely to ensure that only consistent values remain, and it does not generate explanations for the removals it performs. Instead, the explanation algorithms introduced in later sections will build upon this propagator to provide justifications for value removals. The base propagator operates in two steps. First, it removes all tuples from the table that are invalidated by the current variable domains. A tuple is considered invalid if it contains any value that no longer exists in the corresponding variable's domain. Second, for each variable and each value in its domain, the propagator checks whether at least one valid tuple supports that value. If no supporting tuple exists, the value is removed from the variable's domain. A key property of this base propagator is that it achieves GAC in a single pass. A value is removed only if it has no supporting tuples, and such removals cannot invalidate other supporting tuples—since no such tuples exist. Thus, after the propagator completes, all remaining values are guaranteed to participate in at least one valid tuple, ensuring that GAC is achieved. While the base propagator is sufficient to enforce GAC, it forms the groundwork for the explanation algorithms developed in subsequent sections. These algorithms extend the functionality of the base propagator to generate explanations for value removals, enabling their use in lazy clause generation solvers.

4.2 Naive Explanations

To integrate the base propagator into a lazy clause generation (LCG) solver, we need to generate explanations for each value removal. For a value a removed from the domain of a variable x by the base propagator, the solver requires a conjunction of conditions—referred to as an explanation—that justifies the removal. The *naive eager explanation algorithm* systematically identifies and records the variable-value pairs that invalidate all tuples supporting $[x = a]$, providing the required explanation.

4.2.1 Explanation Framework

All explanation algorithms introduced in this paper will rely on the following shared structures:

- **`valid_tuples`**: The set of all valid tuples defined by the table constraint.
- **`supports`**: The subset of `valid_tuples` that initially includes all tuples where $[x = a]$. These tuples are progressively invalidated during the explanation process.
- **`table_domains[`y`]`**: The set of all values that variable y can take in `valid_tuples`.
- **`invalid_domains[`y`]`**: The set of values in `table_domains[`y`]` that are no longer valid in the current domain of y.
- **`reasons`**: A buffer that collects the conditions justifying the removal of $[x = a]$. The explanation is the conjunction of all conditions in `reasons`.

4.2.2 Naive Explanation Algorithm

The algorithm is defined as follows:

Algorithm 1 Naive Explain

for all $y \in \texttt{variables} \setminus \{x\}$ **do**
  for all $b \in \texttt{invalid\_domains}[y]$ **do**
    if $\exists t \in \texttt{supports}$ such that $t[y] = b$ **then**
      $\texttt{reasons} \leftarrow \texttt{reasons} \cup \{[y \neq b]\}$
      $\texttt{supports} \leftarrow \texttt{supports} \setminus \{t \in \texttt{supports} \mid t[y] = b\}$
    end if
  end for
end for

The algorithm iterates over each variable $y \neq x$ and each invalid value $b \in \texttt{invalid\_domains}[y]$. For each b, it checks whether any tuple in `supports` contains $t[y] = b$. If such tuples exist, the condition $[y \neq b]$ is added to `reasons`, and all such tuples in `supports` are removed.

This process continues until all invalid values contributing to the removal of $[x = a]$ have been accounted for. By the end of the algorithm, `supports` is guaranteed to be empty, as the base propagator removes $[x = a]$ only when all tuples supporting $[x = a]$ are invalid. The conditions in `reasons` provide a complete explanation for why $[x = a]$ was removed.

4.2.3 Correctness of the Naive Explanation Algorithm

The correctness of this algorithm is a direct consequence of the properties of the base propagator. Since the base propagator removes $[x = a]$ only when there are no valid tuples supporting $[x = a]$, each tuple in `supports` must rely on at least one variable-value pair $[y = b]$ such that $b \notin \texttt{domain}[y]$. The algorithm systematically identifies these invalid values, ensuring that the collected `reasons` fully and correctly explain the removal of $[x = a]$.

4.3 Greedy Explanations

The *greedy explanation algorithm* improves upon the naive approach by selecting reasons that explain the invalidation of the largest number of tuples in `supports` for $[x = a]$. At each step, it adds the condition $[y \neq b]$ to the explanation, aiming to minimize the size of the explanation (i.e., the number of conditions in `reasons`) compared to processing values in arbitrary order, as in the naive algorithm.

4.3.1 Greedy Reason Selection

To make the algorithm easier to define and understand, we first introduce the following helper function:

Algorithm 2 `Pick Best Greedy Reason`

Precondition: $\texttt{candidates}[y] \subseteq \texttt{invalid\_domains}[y]$ for all variables y

function PICK_BEST_GREEDY_REASON(candidates)
  $\texttt{best\_reason} \leftarrow \texttt{none}$
  $\texttt{best\_count} \leftarrow 0$
  for all $y \in \texttt{variables} \setminus \{x\}$ **do**
    for all $b \in \texttt{candidates}[y]$ **do**
      $\texttt{count} \leftarrow$ number of tuples in `supports` where $t[y] = b$
      if $\texttt{count} > \texttt{best\_count}$ **then**
        $\texttt{best\_reason} \leftarrow [y \neq b]$
        $\texttt{best\_count} \leftarrow \texttt{count}$
      end if
    end for
  end for
  if $\texttt{best\_reason} \neq \texttt{none}$ **then**
    $\texttt{reasons} \leftarrow \texttt{reasons} \cup \{\texttt{best\_reason}\}$
    $\texttt{supports} \leftarrow \texttt{supports} \setminus \{t \in \texttt{supports} \mid t[y] = b\}$
    return `true`
  end if
  return `false`
end function

This function, given a set of candidates, picks the candidate $[y = b]$ that invalidates the largest number of tuples in `supports`, adding the corresponding condition $[y \neq b]$ to the explanation. It also removes all affected tuples from `supports`. If no valid reason can be found, it returns `false`. The set of *candidates* refers to invalid values that can be used to explain the removal of tuples in `supports`. For this section, candidates are defined as `invalid_domains`, which contains all values $[y = b]$ where $b \in \texttt{table\_domains}[y] \setminus \texttt{domain}[y]$. However, the algorithm is general and will be used with other subsets of invalid values as candidates in later sections.

4.3.2 Greedy Explanation Algorithm

The greedy explanation algorithm is then defined as:

Algorithm 3 `Greedy Explain`

```
1: while supports ≠ ∅ do
2:     PICK_BEST_GREEDY_REASON(invalid_domains)
3: end while
```

The algorithm repeatedly invokes `pick_best_greedy_reason`, using `invalid_domains` as candidates, until all tuples in `supports` are explained. At each step, it selects and processes the value $[y = b]$ that invalidates the largest number of remaining tuples, ensuring that `supports` is progressively reduced to an empty set.

4.3.3 Advantages and Correctness

The greedy explanation algorithm guarantees correctness because it terminates only when `supports` is empty, ensuring that all tuples supporting $[x = a]$ are invalidated and explained. By prioritizing invalid values that explain the largest number of removals, the greedy approach often produces shorter explanations compared to the naive algorithm. In the worst case, the explanation length matches that of the naive algorithm, but in most scenarios, the greedy approach results in a smaller number of conditions in `reasons`.

4.4 Optimized Lazy Explanations

The previous sections have focused on *eager explanation methods*, which generate explanations during propagation. In contrast, *lazy explanation methods* defer explanation generation until a conflict occurs. At this point, they can use the current nogood, represented as a conjunction of literals ($[l_1] \wedge [l_2] \wedge \cdots \wedge [l_k]$), to tailor explanations that directly contribute to nogood minimization. Each literal $l = [y \diamond v]$ specifies a variable y, a value v, and a relational operator $\diamond$ (e.g., $\leq, \geq, =, \neq$).

The *optimized lazy explanation algorithm* aims to minimize the size of the nogood, defined as the number of literals it contains after simplification. By focusing on generating concise explanations during conflict analysis, the algorithm improves solver performance through better pruning and improved unit propagation strength.

4.4.1 Optimal Candidate Identification

To minimize the nogood size, for each variable y, the algorithm determines a subset S of $\texttt{invalid\_domains}[y]$ consisting of values that, when $[y \neq d]$ (where $d \in S$) is added to the explanation, do not increase the number of literals in the nogood after simplification. This subset S of values is referred to as $\texttt{subsumed\_domains}[y]$.

The computation of `subsumed_domains` is as follows:

Algorithm 4 Compute Subsumed Domains

for all literal $l = [y \diamond v]$ in the current nogood **do**
  if y is not in the table constraint **then**
    continue
  end if
  if $\diamond = '\leq'$ **then**
    $\texttt{subsumed\_domains}[y] \leftarrow \texttt{subsumed\_domains}[y] \cup \{d \in \texttt{invalid\_domains}[y] \mid d \geq v\}$
  end if
  if $\diamond = '\geq'$ **then**
    $\texttt{subsumed\_domains}[y] \leftarrow \texttt{subsumed\_domains}[y] \cup \{d \in \texttt{invalid\_domains}[y] \mid d \leq v\}$
  end if
end for

Note: In the algorithm, the condition $\diamond =' \leq '$ or $\diamond =' \geq '$ refers to the operator in the literal being either less-than-or-equal-to or greater-than-or-equal-to, respectively.

To understand why the algorithm chooses certain values, let's consider the literal $[y \diamond v]$ in the nogood and the case where $\diamond =' \leq '$. In this case, all values $d \geq v$ (with $d \in$ `invalid_domains`$[y]$) are added to `subsumed_domains`$[y]$ because the addition of $[y \neq d]$ will not increase the size of the nogood. There are two cases to consider:

- If $d = v$, adding $[y \neq d]$ combines with $[y \leq v]$ to form $[y \leq v - 1]$, reducing the range of valid values for y without increasing the nogood size.
- If $d > v$, adding $[y \neq d]$ is redundant because it is subsumed by $[y \leq v]$, which already excludes d.

For $\diamond =' \geq '$, the logic is equivalent, with $d \leq v$ being added to `subsumed_domains`$[y]$.

At the end of this process, `subsumed_domains`$[y]$ contains all values that can be added to the explanation without increasing the number of literals in the nogood.

4.4.2 Optimized Lazy Explanation Algorithm

The explanation algorithm is then defined as:

Algorithm 5 Optimized Lazy Explanation

```
1: while supports ≠ ∅ do
2:     if not PICK_BEST_GREEDY_REASON(subsumed_domains) then
3:         break
4:     end if
5: end while
6:
7: while supports ≠ ∅ do
8:     PICK_BEST_GREEDY_REASON(invalid_domains)
9: end while
```

After computing `subsumed_domains`, the algorithm first prioritizes values from `subsumed_domains`. These values are guaranteed not to increase the size of the nogood after simplification and are added to the explanation as long as they contribute to invalidating the remaining `supports`. If `subsumed_domains` is exhausted before all tuples in `supports` are explained, the algorithm proceeds to use `invalid_domains`, which contains all invalid values, as candidates.

By iteratively calling `pick_best_greedy_reason` with these candidate sets, the algorithm ensures that all tuples in `supports` are invalidated and explained. The use of `subsumed_domains` prioritizes nogood minimization, while the fallback to `invalid_domains` guarantees correctness by explaining any remaining tuples.

4.4.3 Advantages and Correctness

The *optimized lazy explanation algorithm* ensures correctness because it terminates only when `supports` is empty, guaranteeing that all tuples supporting $[x = a]$ are invalidated and explained. By prioritizing values from `subsumed_domains`, it minimizes the number of literals in the nogood while maintaining the same strength as the base propagator.

In the worst case, the algorithm performs as well as the greedy explanation algorithm, which is at least as good as the naive explanation algorithm. Since the naive explanation algorithm can justify any removal made by the base propagator, this algorithm is at least as strong as the base propagator. However, in practical scenarios, by leveraging nogood simplification, the optimized lazy explanation algorithm often produces shorter and more efficient explanations.

5 Experimental Setup and Results

The purpose of this experiment is to evaluate the explanation strategies proposed in this paper: **Naive (Eager)** (Section 4.2), **Greedy (Eager)** (Section 4.3), and **Optimized (Lazy)** (Section 4.4). Additionally, we include the current state-of-the-art **Bacchus SAT encoding** (Section 2.7) of the table constraint as a baseline for comparison.

Section 5.1 describes the experimental setup, including the benchmark problems, evaluation metrics, and the methodology used to ensure a fair comparison between propagators. Section 5.2 presents the results of the experiments, analyzing the performance of each explanation strategy in terms of conflicts, clause length, and runtime.

5.1 Experimental Setup

The three explanation-based propagators, as well as the Bacchus SAT encoding method have been implemented in the **Pumpkin solver** [10], an LCG CP solver developed by ConSol Lab at TU Delft, with added support for table constraints in the Minizinc problem format. For more details on how the algorithms were implemented in the Pumpkin solver, refer to Appendix A. Pumpkin's nogood simplification capabilities make it an ideal platform for evaluating different explanation strategies.

All experiments were run on a ROG Zephyrus G14 with an AMD Ryzen 9 8945HS CPU @ 4.00 GHz and 32 GB of RAM.

5.1.1 Benchmarks and Dataset

The benchmark problems were sourced from the **Minizinc Challenge (2009–2024)** [20], comprising a selection of problems tagged as using table constraints. The dataset includes a diverse range of combinatorial and real-world-inspired problems, with objectives such as minimization, maximization, and satisfiability. Some problems exclusively use table constraints, while others combine table constraints with additional constraints. This diversity ensures the experiments evaluate propagators across a wide range of problem types and scenarios.

5.1.2 Objective Threshold

Many problems in the MiniZinc Challenge are minimization or maximization problems designed to test solvers' ability to find near-optimal solutions within a fixed time limit. To ensure a fair comparison of propagators, we did not limit solvers by time, as slower explanation algorithms would otherwise explore less of the solution space, skewing results. Instead, we introduced an *objective threshold*: the solver halts when a solution with an objective value below this threshold (for minimization problems) is found. This ensures all propagators explore the same solution space.

To determine the threshold, each problem instance was first solved using the Bacchus encoding with a 15-minute timeout. The best objective value obtained served as the threshold for subsequent experiments. This approach ensures that the results are not biased against slower propagators and provide a fair comparison.

5.1.3 Controlling for Branching Strategy

To ensure a fair comparison, the branching strategy was fixed for each problem based on the recommendations specified in the Minizinc dataset. This eliminates variability caused by random branching paths and isolates the effects of the propagators.

5.1.4 Metrics

For each problem instance we recorded the following metrics:

- $\overline{L}$ - Average Learned Clause Length. Looking at the average length of a learned clause, otherwise known as the negation of a nogood [8], will help evaluate whether different explanation strategies are effective at producing explanations that minimize the nogood in the hopes they are more general.
- C - Number of Conflicts. This metric will help assess the efficiency of the solver by measuring the number of conflicts encountered during solving. A lower number of conflicts indicates better pruning of the search space.

The above two metrics do not change between runs of the problem instance because of the fixed branching strategy and use of objective thresholds. They will help us evaluate the theoretical performance of the algorithms. Additionally, we will also look at the practical performance of the current propagator implementations:

- $\overline{T}$ - Average Run Time. The runtime is averaged over running the same problem instance 10 times.

Additionally since the metrics can vary highly between problem instances we will instead look at normalized metrics $\overline{L}*$, $C*$, $\overline{T}*$. The normalized metric for a specific propagator P was computed by taking the metric of propagator P and dividing it by the metric produced by our baseline Bacchus encoding.

5.2 Results and Analysis

Figure 2 presents all the performance metrics for each technique on each problem instance.

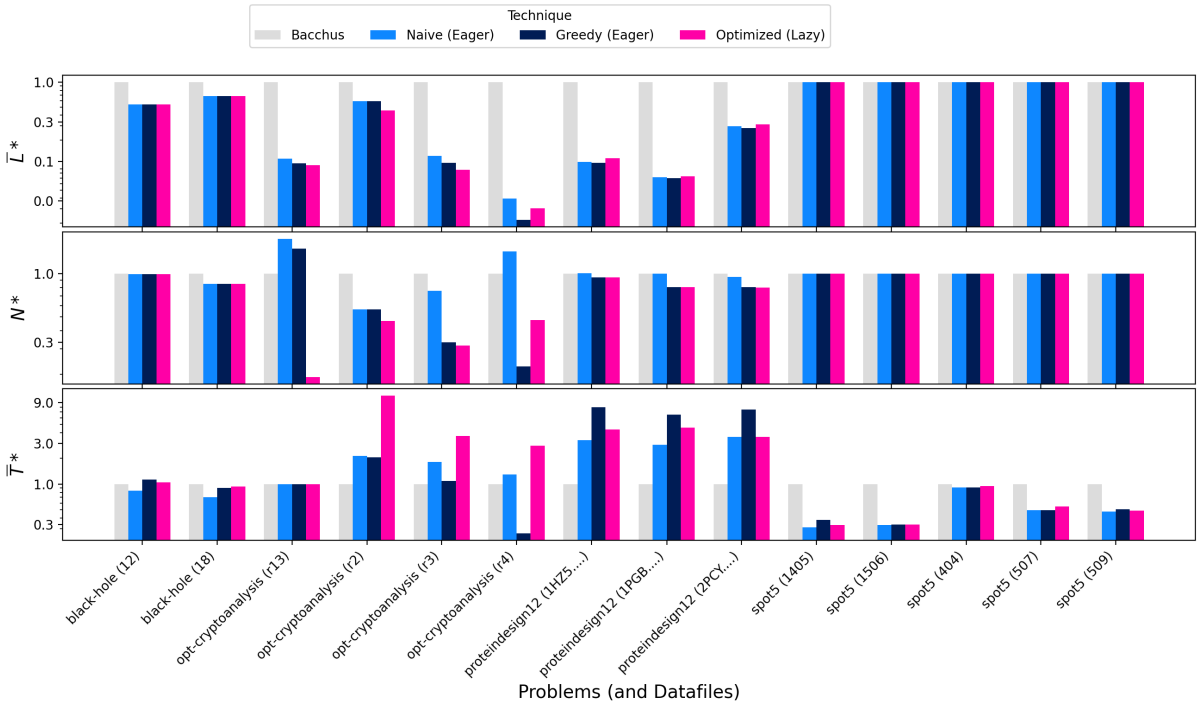

Figure 2: Normalized metrics for each technique across problem instances, visualized with a logarithmic scale for better clarity.

5.2.1 Average Learned Clause Length

Across all problem instances, explanation-based propagators consistently outperform the Bacchus encoding in reducing the average learned clause length by an average of **46%**. For *opt-cryptoanalysis* and *proteindesign12*, explanation-based propagators reduce the average learned clause length by an average of **82%**, with Optimized (Lazy) being **11%** more effective than Naive (Eager) approach. However, for *black-hole* and *spot5*, there is only a small **11%** reduction for explanation-based approaches over the Bacchus encoding method.

This trend can be explained by the characteristics of the table constraints in these problems. In *opt-cryptoanalysis* and *proteindesign12*, table constraints typically involve 3–9 variables, providing more opportunities for the propagators to use literals in explanations that are subsumed by the nogood, thereby shortening its length. In contrast, *spot5* and *black-hole* have simpler table constraints, with only 1–3 variables, limiting the advantages of explanation-based methods in this regard.

5.2.2 Number of Conflicts

The number of conflicts depends significantly on the problem. On average, the number of conflicts decreases by **11%** for explanation-based methods compared to Bacchus. The reduction was most pronounced for problems *opt-cryptoanalysis* and *proteindesign12* where explanation-based propagators reduced the number of conflicts by an average of **21%**. This is likely the case because these problems utilized larger tables with 100-2400 tuples per table. With larger table sizes there is more opportunity for explanation-based propagators to pick out only the literals that matter compared to Bacchus SAT encoding where the order of literals that propagate is arbitrary. For the two problems, among the explanation-based propagators, the *Optimized (Lazy)* propagator outperforms both the *Greedy (Eager)* and *Naive (Eager)* propagators, with a performance improvement over *Naive (Eager)* of **39%** for *opt-cryptoanalysis* and *proteindesign12*.

Additionally, explanation-based propagators benefit from their ability to dynamically encode only the relevant parts of the problem during solving, unlike the Bacchus approach, which statically encodes all constraints upfront. While this is not the primary factor behind the reduction in conflicts, it provides further support for their adaptability and effectiveness in larger-scale problems.

However, for problems with smaller table constraints, such as *spot5*, this advantage is less pronounced. The *spot5* problem used over 10,000 separate table constraints, each containing only 3–63 tuples and 1-3 variables. In

contrast, other problems consisted of far fewer tables, each with significantly larger numbers of tuples (100–2400). With constraints divided among many small tables, explanation-based propagators are unable to leverage global information effectively. As a result, *spot5* showed no significant change in the number of conflicts compared to Bacchus. For such cases, the choice of propagator has minimal impact.

5.2.3 Runtime

In most cases, the Bacchus encoding outperforms explanation-based propagators in terms of runtime by a factor of 2x on average. Among the explanation-based methods, the *Naive (Eager)* propagator achieves similar runtime to the *Optimized (Lazy)* propagator for **black-hole**, **protein-design12** and **spot5**, with differences of only **5%** in favor of *Naive (Eager)* on average. This can be attributed to the computational simplicity of the *Naive (Eager)* method, which offsets the runtime gains achieved by reducing conflicts in the *Optimized (Lazy)* method.

The *Greedy (Eager)* propagator, on the other hand, is consistently the slowest among the explanation-based methods. This observation suggests that if a more advanced algorithm is desired to reduce conflicts and runtime, it is more effective to use the fully-fledged *Optimized (Lazy)* propagator, which builds upon the Greedy approach.

A notable exception is *opt-cryptoanalysis*, where the *Optimized (Lazy)* propagator shows significantly slower runtime compared to other propagators. Profiling the algorithm with a Rust flamegraph [9] reveals that a substantial portion of time is spent in the lazy explanation algorithm. This problem involves table constraints with the highest number of variables, resulting in conflicts with many literals that require lazy explanations. Due to current solver architecture limitations, the lazy explanation algorithm recomputes the set of optimal explanation literals for the current nogood individually, even when this information is identical across multiple variables. This inefficiency leads to up to 70 redundant expensive computations per conflict, significantly impacting runtime for this problem.

5.2.4 General Analysis

The explanation-based propagators in this study are not fully optimized, as the focus is primarily on theoretical metrics. Their current implementation involves inefficient nested iterations and frequent rebuilding of propagator states for each explanation. By contrast, the Bacchus encoding leverages the solver's already optimized clause propagation algorithms, giving it a natural runtime advantage. This disparity partially explains why Bacchus consistently outperforms explanation-based methods in terms of runtime.

However, the explanation-based propagators demonstrate a clear strength in reducing conflicts, Optimized (Lazy) propagator achieving reduction in the number of conflicts over Bacchus method in all problems by an average of **23%** and up to **64%** for *opt-cryptoanalysis*. This result highlights their potential for future improvements. By optimizing the explanation algorithms, introducing more efficient data structures, and addressing architectural limitations—such as enabling shared precomputed nogood data across explanations during a single conflict—it may be possible to significantly enhance the runtime performance of explanation-based propagators.

While the Bacchus method remains a practical and widely adopted approach due to its simplicity and efficiency, the results suggest that explanation-based propagators could surpass it in runtime performance with further research and optimization. This makes them a promising direction for advancing the state of the art in solving problems with table constraints.

6 Responsible Research

This research adheres to the principles outlined in the Netherlands Code of Conduct for Research Integrity[16]. The study has been conducted with a commitment to openness, transparency, and reproducibility, ensuring that all processes and results are in line with ethical research standards.

To promote accessibility and reproducibility, the MiniZinc Challenge dataset used in this research is publicly available[20]. Importantly, this dataset does not contain any personal information, ensuring compliance with data privacy and ethical standards.

The research results have been designed to remain largely hardware independent, with the exception of runtime, which is not the focus of the research. Furthermore, the runtime has been averaged over multiple instances, and care was taken to minimize background process interference on the runtime. This approach broadens the applicability of the findings across various computational environments and aligns with the principle of enabling equitable access to research findings and tools for the wider academic community.

In the interest of fostering transparency and facilitating further research, the explanation algorithms developed during this study, as well as the benchmarking scripts used to evaluate them, have been made publicly available[1].

By providing these resources, this work supports the replication of experiments and encourages other researchers to build upon these contributions in their own studies.

7 Conclusions and Future Work

This paper presented a study on explanation generation for table constraints in *Lazy Clause Generation (LCG)* solvers, introducing and evaluating eager and lazy propagators, including an optimized lazy propagator that leverages nogood knowledge. Experiments using the Pumpkin solver demonstrated that incorporating nogood knowledge into the explanation process produces more general nogoods, significantly reducing both the number of conflicts (by **23%**) and the average learned clause length (by **46%**) over the current state-of-the-art Bacchus method. These results validate the hypothesis that optimized explanations tailored to the current nogood can enhance the solver's ability to prune the search space effectively.

While explanation-based propagators demonstrated clear advantages in reducing conflicts and clause length, the current implementations come with a notable runtime overhead, with the Bacchus encoding being up to **2x faster** on average. Future work should focus on reducing this runtime penalty by addressing inefficiencies in explanation generation, such as avoiding redundant computations and leveraging shared precomputed nogood data. Improving runtime efficiency will be critical for making explanation-based propagators more competitive and practical in real-world applications.

Additionally, further research could explore how nogood knowledge could be utilized by propagators for other constraints to optimize their explanations via lazy explanations. This approach has the potential to generalize the benefits observed for table constraints, enhancing solver performance across a broader range of problems. Another promising direction is the optimization of other metrics that were not considered in this paper, such as the nogood literal block distance (LBD) [3], which could further improve conflict resolution and solver performance. By advancing these areas, explanation-based propagators and lazy explanations could offer not only theoretical but also practical benefits for solving complex problems.

References

[1] Markas Aisparas. Pumpkin solver fork. https://github.com/MarkasAis/pumpkin. Fork of Pumpkin Solver with explanation-based propagators for table constraints. Accessed: January 25, 2025.

[2] F. Bacchus. Gac via unit propagation. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 133–147. Springer, 2007.

[3] Md Solimul Chowdhury, Martin Müller, and Jia-Huai You. Characterization of glue variables in cdcl sat solving, 2019. Accessed: January 25, 2025.

[4] Geoffrey Chu, Peter J. Stuckey, and Andreas Schutt et al. Chuffed solver. https://github.com/chuffed/chuffed. Accessed: January 25, 2025.

[5] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: Efficiently filtering table constraints with reversible sparse bitsets. *arXiv preprint arXiv:1604.06641*, 2016. Accessed: January 25, 2025.

[6] Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining alldifferent. In *Proceedings of the Australasian Computer Science Conference (ACSC 2012)*, pages 115–124. ACM, 2012. Accessed: January 25, 2025.

[7] Thibaut Feydy, Andreas Schutt, and Peter J. Stuckey. Semantic learning for lazy clause generation. In *Proceedings of TRICS Workshop: Techniques foR Implementing Constraint Programming Systems, TRICS 2013, Uppsala, Sweden*, 2013.

[8] Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In *Lecture Notes in Computer Science*, pages 352–366, 2009.

[9] Brendan Gregg and Contributors. Flamegraph for rust. https://github.com/flamegraph-rs/flamegraph, 2014. Accessed: January 25, 2025.

[10] ConSol Lab. Pumpkin solver. https://github.com/consol-lab/pumpkin. Accessed: January 25, 2025.

[11] C. Lecoutre. Str2: Optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.

[12] Christophe Lecoutre and Radoslaw Szymanek. Generalized arc consistency for positive table constraints. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, pages 284–298, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[13] Jean-Baptiste Mairy. *Propagators for Table Constraints*. PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2015. Accessed: January 25, 2025.

[14] Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre. The smart table constraint. In Laurent Michel, editor, *Integration of AI and OR Techniques in Constraint Programming*, pages 271–287, Cham, 2015. Springer International Publishing.

[15] J. P. Marques-Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *Digest of IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 220–227, November 1996. Accessed: January 25, 2025.

[16] N. C. of Conduct for Research Integrity. Netherlands code of conduct for research integrity. Technical report, Netherlands Organisation for Scientific Research (NWO), 2018. Accessed: January 25, 2025.

[17] Alexander Nadel. *Understanding and Improving a Modern SAT Solver*. PhD thesis, Tel Aviv University, August 2009. Accessed: January 25, 2025.

[18] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, September 2009. Accessed: January 25, 2025.

[19] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Explaining the cumulative propagator. *Constraints*, 16(3):250–282, 2011. Accessed: January 25, 2025.

[20] Guido Tack and Peter J. Stuckey. Minizinc challenge. https://www.minizinc.org/challenge. Accessed: January 25, 2025.

A Propagator Implementation in Pumpkin

In the Pumpkin solver, propagators interact with the solver through the following interface:

- **propagate**: Called when the domains of variables change, this method must remove invalid values to enforce GAC. Propagators can either:
  - Provide eager explanations for the removed values immediately.
  - Defer explanation generation until a conflict occurs, indicating that explanations will be provided lazily.
- **lazy_explain**: If the propagator deferred explanation generation, this method is called at the time of conflict. At this point, the propagator has access to the current nogood, which can be used to generate explanations.

Following the interface, the propagators used in the experiments are implemented as follows:

- **Naive (Eager) Propagator**:
  - **propagate**: Uses the base propagator to remove values and the naive explanation algorithm to immediately generate eager explanations for the removed values.
- **Greedy (Eager) Propagator**:
  - **propagate**: Uses the base propagator to remove values and the greedy explanation algorithm to immediately generate eager explanations for the removed values.
- **Optimized (Lazy) Propagator**:
  - **propagate**: Uses the base propagator to remove values and defers explanation generation until the time of conflict.
  - **lazy_explain**: Uses the optimized explanation algorithm to generate explanations that leverage knowledge of the current nogood to produce more general nogoods.