Evaluating Nogood Management Heuristics in Constraint Programming Solvers

Master Thesis

Hubert Damian Nowak

Delft University of Technology

TU Delft

Evaluating Nogood Management Heuristics in Constraint Programming Solvers

How forgetting the right information improves solver performance

by

Hubert Damian Nowak

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 2, 2026 at 10:00 AM.

Student number:	5552435	
Project duration:	November 17, 2025 – July 2, 2026	
Thesis committee:	Dr. E. Demirović,	Supervisor
	Dr. S. Chakraborty,	
	Ir. M. Flippo,	Daily supervisor

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

This thesis marks the end of my five-year-long journey at the Delft University of Technology. It was an incredible time, full of challenges and learning, which showed me that there is more to computer science than for-loops and conditional statements that I learned for a competition in primary school. I gained many invaluable skills that I look forward to putting into practice in the future. However, none of this would have been possible without the help of many people, both lecturers and fellow peers, who guided and supported me along the way.

First and foremost, I would like to thank my supervisor, Emir Demirović, for introducing me to the world of Constraint Programming during the Bachelor's and Master's courses, and for his advice throughout this project. Each meeting and each email provided me with plenty of new ideas and points to consider. I also extend my gratitude to Maarten Flippo for the day-to-day supervision and all the morning/evening meetings. Even the large timezone difference did not affect his patience when I sometimes struggled to articulate my thoughts clearly.

Next, I wanted to thank all my friends and the people I had the pleasure of meeting during my studies. Their friendship, support, encouragement, and companionship definitely made this time not only more enjoyable but also more meaningful.

Last but not least, I thank my parents for their unconditional support, both in my *unexpected* decision to move abroad for university, and ever since. They were always willing to listen to me, even when I could not wholly explain what exactly I was doing in my thesis. Let's see if this document does a better job at conveying my research.

Hubert Nowak
Rotterdam, June 2026

Abstract

Conflict analysis has become one of the key techniques behind the success of modern Constraint Programming (CP) solvers. In Lazy Clause Generation (LCG), conflicts are analysed to derive learned constraints, known as nogoods, which help the solver avoid revisiting infeasible regions of the search space. While learned nogoods can substantially improve search efficiency, maintaining large nogood databases introduces computational and memory overheads, making periodic removal of low-quality nogoods essential. Existing approaches largely rely on heuristics adopted from SAT solving, yet their comparative performance in the CP setting has not been systematically studied.

This thesis investigates the impact of nogood quality metrics and database reduction strategies on the performance of CP solvers. We analyse eight nogood quality metrics, including both established measures such as activity and Literal Block Distance (LBD), as well as new CP-specific metrics. We analyse their correlation with nogood usefulness, defined in terms of propagation behaviour across progressively stricter notions of utility. We find that all metrics can, to some extent, predict the usefulness of nogoods, and we identify four metrics for further study.

We then propose and evaluate a set of nogood management schemes that combine these metrics in different ways. Experiments across hundreds of optimisation and satisfaction instances from the MiniZinc Challenge show that a scheme retaining nogoods with either low LBD or high activity achieves the fewest conflicts and outperforms the default management strategy of the state-of-the-art solver Pumpkin. For anytime behaviour, incorporating the number of variables enables faster discovery of high-quality solutions. We further demonstrate that periodic database reductions are necessary, but that the choice of database size parameters involves a trade-off between finding solutions quickly and proving optimality. Finally, we find that differences in solver performance cannot be fully explained by schemes' ability to remove nogoods that cause few propagations and retain the active ones, suggesting avenues for future research.

Contents

1 Introduction

Some problems in computer science are considered to be *hard*. One way to assess this hardness is by analysing their runtime – intuitively, if we can obtain a result in several milliseconds, then the task likely was not very complicated. For certain problems, clever algorithms offer fast solutions even when using the simplest machines. In contrast, others require brute-force approaches that check exponentially many potential solutions one by one, leading to prohibitively long runtimes, even when leveraging the computational power of modern supercomputers. Despite the lack of efficient algorithms for such problems, solving them within a reasonable time remains essential, as many of these tasks arise in various everyday domains such as scheduling [5, 21], air traffic control [13], resource allocation [12, 22], and chip design [2].

Many of these problems are combinatorial in nature, meaning they involve finding the optimal solution from an exceptionally large set of potential objects. As an exhaustive search through all options is not tractable, we need specialised algorithms that can quickly rule out large parts of the search space. To address this challenge, numerous solving approaches have been proposed, including Boolean satisfiability (SAT), Mixed Integer Linear Programming (MILP), Constraint Programming (CP), Local Search, and various approximation algorithms. Each of these methods has its advantages, such as ease of problem modelling, the time required to find the first solution, or guarantees of optimality, thereby making them successful in specific domains. Through sophisticated heuristics and careful engineering, these techniques can efficiently solve many surprisingly hard problems.

In this thesis, we focus on Constraint Programming – a paradigm in which problems are expressed using a set of integer variables and various high-level constraints imposed on them. A general CP solver examines the search space by interleaving *exploration*, in which it assigns values to variables, and *propagation* (or *inference*), which eliminates values that cannot be a part of any valid solution based on the current state of the solver. Modern solvers include various additional optimisations, such as specialised search heuristics and powerful global propagators.

One of the most significant improvements in solvers' performance was caused by the introduction of *learning* via the Conflict-Driven Clause-Learning (CDCL) algorithm [28] in SAT solving. In solvers with a learning scheme applied, each propagation needs to have a corresponding *explanation*. When the solver's state becomes conflicting, meaning it enters a part of the search space without any valid solutions, these explanations are analysed to construct a *learned clause* (also called *nogood* in CP) that describes the reason for the conflict. This clause is then saved, allowing the solver to detect and avoid similar conflicting branches of the search tree, reducing the amount of work needed in the future. The success of CDCL in SAT solving encouraged the development of conflict learning schemes in CP, resulting in numerous techniques such as g-nogoods [23], c-learning [29], nogoods from restarts [25] and Lazy Clause Generation (LCG) [14, 32], each with its own strengths and weaknesses. LCG is currently the most widespread scheme, used in several modern solvers such as Chuffed [7], OR-Tools [33], and Pumpkin [15].

While each learned nogood reduces the size of the search space left to explore, preventing the solver

from repeating some of its work, it also needs to be stored and constantly monitored to detect the moment when we can use it to perform propagations. Therefore, there is a trade-off – storing more nogoods provides the solver with more information but also increases the resources required. Moreover, even when nogoods do make propagations, these might not be particularly useful for advancing the solving process, resulting in wasted calculations. Thus, storing too many nogoods can be detrimental to the solver's performance. A common way of dealing with this problem is to measure the quality of each nogood using a specific heuristic and periodically remove those judged less useful.

In SAT solving, there were numerous techniques proposed for determining the quality of learned clauses, such as size [28], activity [11], Literal Block Distance (LBD) [3] or nbSAT [26]. Each of these metrics measures different aspects of the clauses, focusing on their structure, relations between components or earlier usage during the solving process, thus utilising different information. When introduced, each metric was extensively compared with the previous ones in terms of performance. Some of them were then imported into other domains, for example, in Answer Set Programming, the solver clasp [18] used activity, while in the context of Constraint Programming, Pumpkin [15] and Chuffed [7] respectively adopted LBD and activity. For CP, there was also a new metric proposed – NACRE [19] measured the number of integer variables in a nogood. However, there was no research into comparing the performance of these metrics specifically in the setting of Constraint Programming.

Even when we choose a given heuristic for measuring the quality of nogoods, there are several schemes of how and when the nogoods are removed. Some approaches involve deleting a nogood as soon as a specific condition (e.g. related to size) is satisfied [28, 30]. In contrast, others prune the database in bulk, periodically removing half of all the stored nogoods [3, 7, 19]. There are also more complex approaches that divide the nogoods into tiers based on predefined heuristics and manage each tier independently [24, 31]. Therefore, when combined with the many possible measures of nogood usefulness, there are numerous ways of managing the nogood database.

In this thesis, we investigate the influence of nogood quality metrics and nogood management schemes on the performance of CP solvers. Firstly, we propose and examine several heuristics for ranking nogoods. Then, we evaluate solver performance in case we use these metrics to guide the nogood database reduction process, combining them in different ways. We primarily focus on the number of conflicts encountered as a proxy for the amount of work required, but we also investigate the anytime behaviour of solvers. Lastly, we measure the impact of several parameters related to nogood management and investigate whether differences in performance stem from differing abilities to identify nogoods that cause few propagations.

We therefore aim to answer the following questions:

1. Which nogood quality heuristics can predict whether a nogood will be useful?
2. How does the choice of nogood quality metrics affect solver performance in terms of conflict count?
3. Which nogood management schemes offer the best anytime behaviour?
4. How does the allowed database size impact the number of solved problem instances?
5. Do differences in performance between nogood management schemes originate from differences in the ability to remove nogoods that cause few propagations?

Our experimental results demonstrate that all evaluated metrics can provide meaningful information about the future usefulness of learned nogoods. Some of the metrics are highly correlated with each other, suggesting they encode similar characteristics of the nogoods. When combining metrics for database management, schemes that retain nogoods excelling in at least one measure outperform those requiring nogoods to score well on all included metrics, with the combination of LBD and activity being most effective at minimising conflict counts in both optimisation and satisfaction settings. However, the differences between the nogood removal schemes are rather small unless compared with a specifically designed poor-performing one, indicating that it is more important to avoid exceptionally bad schemes than to choose the best one. When looking at the anytime performance, we found that incorporating the number of variables alongside LBD and activity leads to finding good solutions more quickly. We further found that periodic nogood removal is essential when solving optimisation instances, as solvers without it deteriorate significantly on larger instances, while the choice of database

size parameters governs a trade-off between solution-finding speed and the ability to prove optimality. Lastly, although we hypothesised that performance differences between schemes stem from their ability to identify and remove nogoods that cause few propagations, our analysis does not reveal any clear characteristic that fully explains these differences, pointing to open questions for future work.

This thesis is structured as follows. First, in Chapter 2, we present necessary background information about Constraint Programming and related techniques. In Chapter 3, we discuss previous work on nogood quality metrics and database reduction schemes. Next, in Chapter 4, we investigate the correlation between nogood's metrics and its usefulness. Chapter 5 focuses on differences in the solver's performance when using different metrics for database reductions. Lastly, the thesis concludes in Chapter 6 with a survey of the main findings and an outline of potential future research directions.

2

Background

This chapter introduces the concepts required for the remainder of the thesis. We first discuss the key ideas of SAT solving and Constraint Programming (CP), followed by an overview of CP solving techniques. Afterwards, we present Lazy Clause Generation (LCG), a fundamental component of modern CP solvers. Finally, we discuss proof logging in Constraint Programming.

2.1. SAT solving

The *Boolean Satisfiability Problem* (SAT) is the problem of determining whether there exists an assignment of truth values to a set of Boolean variables $x_1, x_2, \ldots, x_n$ such that a given propositional formula using those variables evaluates to true. SAT was the first problem shown to be NP-complete by Cook [8], and because of that, it became one of the most extensively studied problems in computer science.

SAT instances are represented in *Conjunctive Normal Form* (CNF), where a formula is expressed as a conjunction of *clauses*, each clause being a disjunction of *literals*. A literal is either a Boolean variable x_i or its negation $\neg x_i$. Thus, a literal can be assigned to one of the $\{\top, \bot, unassigned\}$ based on the truth value of the underlying variable. A clause is *satisfied* if at least one of its literals evaluates to $\top$, and the whole CNF formula is *satisfiable* if there exists an assignment of truth values to the variables such that all its clauses are satisfied simultaneously.

Solving SAT instances relies on using the *unit rule*, which states that if in a given clause $C : (l_1 \vee l_2 \vee \ldots \vee l_k)$ all but one literals (say $l_1, l_2, \ldots, l_{k-1}$) are assigned to $\bot$, then the remaining one (l_k) needs to be assigned to $\top$ to satisfy the clause. Such an assignment might allow further applications of the unit rule in other clauses, and this procedure is repeated as long as possible.

In 1961, Davis, Logemann, and Loveland [9] introduced the DPLL algorithm, a simple approach to solving SAT problems that, given a CNF formula, first tries to use the unit rule, and when this rule does not provide new information, it recursively evaluates two subproblems. These subproblems are created by making a *decision* – heuristically choosing a variable x_i and splitting into two cases, one where x_1 is assigned to $\top$, and the other where it is assigned to $\bot$ (while keeping the previous assignments of other variables). In this depth-first manner, the algorithm is guaranteed to explore the whole search space and find an assignment of truth values to variables that satisfies the given formula, if such an assignment exists.

During solving, whenever any of the recursive calls encounters a clause with all of its literals assigned to $\bot$, the solver determines that the current assignment of variables is infeasible and concludes that the last decision must have been incorrect. It then *backtracks*, undoing the last decision, and proceeds with evaluating further subproblems. Later, the CDCL algorithm [28] was introduced, which used *conflict analysis* to examine the underlying reason for the falsified clause, allowing backtracking several decisions at once and creation of new clauses that efficiently decrease the size of the search space. We discuss conflict analysis in more detail in the context of CP in Section 2.3.

2.2. Constraint Programming

Constraint Programming (CP) is a general-purpose paradigm for solving combinatorial optimisation and satisfaction problems. In CP, a problem is first modelled using variables and constraints. Then, a specialised solver is responsible for exploring the search space and reasoning about the problem structure. CP has been successfully applied in domains such as scheduling, resource allocation, verification, planning, and logistics.

One of the key advantages of CP is the availability of high-level global constraints together with specialised propagators. Rather than expressing a problem purely through low-level logical formulas (such as Boolean variables), CP allows the modeller to represent structure directly. This often makes the modelling process much easier, while also enabling stronger inference during search.

2.2.1. Constraint Satisfaction Problems

A *Constraint Satisfaction Problem* (CSP) is defined as a tuple

$$\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}),$$

where:

- $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ is a set of *decision variables*,
- $\mathcal{D}(x_i)$ denotes the *domain* of variable x_i, with $\mathcal{D}(x_i) \subset \mathbb{Z}$, defining the set of values that can be assigned to x_i, and
- $\mathcal{C}$ is a set of *constraints*, with each constraint $C(X) \in \mathcal{C}$ restricting the allowed values for variables $X \subseteq \mathcal{X}$. A constraint can be defined as the set of all allowed tuples of values for these variables, which is a subset of the Cartesian product of domains of $x_i \in X$, namely $C(X) \subseteq \mathcal{D}(x_1) \times \mathcal{D}(x_2) \times \ldots \times \mathcal{D}(x_k)$.

An *assignment* ϕ is a mapping from variables to a set of values in their respective domains, namely, it assigns each variable $x_i \in \mathcal{X}$ to a set of values $V \subseteq \mathcal{D}(x_i)$. If a variable is mapped to a singleton set, we say it is *assigned*. When all variables are assigned, the corresponding assignment is a *complete assignment*, and otherwise it is a *partial assignment*.

With the definition of assignments, we can also define constraints in another way – a constraint C is a predicate $\phi \to \{0, 1\}$ reporting whether a given assignment is *feasible* or *infeasible*. Using this definition of constraints, we can cleanly define a *solution* for a CSP as a complete assignment ϕ_{sol} for which $\forall C \in \mathcal{C}, C(\phi_{sol}) = 1$.

Example 1. *As an example of how problems can be expressed as CSPs, consider the N-Queens problem. It involves placing N chess queens on an $N \times N$ chessboard, such that no two queens share the same column, row or diagonal. We start modelling this problem by creating N variables, $\mathcal{X} = \{x_1, x_2, \ldots, x_N\}$, with each variable corresponding to one queen. We can already assume that the queens will be placed in distinct columns, meaning that the value a given variable is assigned denotes the row in which the corresponding queen is placed. Since all queens can initially be placed in any row, we have that $\mathcal{D}(x_i) = \{1, 2, \ldots, N\}$. We now need to ensure that the queens do not share rows and diagonals. In CSP, this can be efficiently modelled using just three* `all-different` *constraints $\mathcal{C} = \{c_1, c_2, c_3\}$, with c_1 : `all-different`$(x_1, x_2, \ldots, x_N)$ ensuring the rows are distinct, c_2 : `all-different`$(x_1 + 1, x_2 + 2, \ldots, x_N + N)$ and c_3 : `all-different`$(x_1 - 1, x_2 - 2, \ldots, x_N - N)$ ensuring respectively that there are no two queens on the same descending or ascending diagonal.*

A CSP can be extended to a *Constraint Optimisation Problem* (COP) by providing a function $f : \phi \to \mathbb{Z}$ which assigns an objective value to each complete assignment. The goal in a COP is then to find a complete assignment that satisfies all the constraints, while (without loss of generality) minimising the objective function.

Atomic constraints are the basic building blocks used to express the state of variables during solving. An atomic constraint is a simple constraint on a single variable $x \in \mathcal{X}$, in the form of $[x \oplus v]$, where $\oplus \in \{=, \leq, \geq, \neq\}$ and $v \in \mathbb{Z}$, such as $[x \geq 5]$ or $[x \neq 3]$. Collections of atomic constraints can be used, for example, to describe the current partial assignments of variables throughout the search process.

2.2.2. Solving algorithm

Modern CP solvers operate through a combination of *propagation* and *exploration*. During propagation, the solver applies *propagators* to reduce domains of variables by removing values that cannot participate in any valid solution given the current partial assignment. Once propagation alone cannot remove any further values, the solver branches into multiple subproblems by making a *decision*.

Formally, a *propagator* is a function $p_C : \mathcal{D} \to \mathcal{D}'$ associated with a constraint C that removes unsupported values from variable domains ($\mathcal{D}'(x_i) \subseteq \mathcal{D}(x_i)$ for every variable x_i), while also preserving the solutions of the CSP. Propagation is repeatedly applied until a *fixpoint* is reached, meaning that no propagator can further reduce any domain. If a variable domain becomes empty during propagation, a *conflict* arises, meaning that the current partial assignment cannot be extended to any solution.

Example 2. *Consider variables x and y, with domains $\mathcal{D}(x) = \{4, 5, 6, 7\}$ and $\mathcal{D}(y) = \{0, 1, 2\}$. Suppose that there is a constraint imposed on them:*

$$x + y \geq 7$$

Since $[y \leq 2]$, we can see that the constraint will be satisfied only when $[x \geq 5]$ holds. Thus, the corresponding propagator would remove 4 from the domain of x, resulting in $\mathcal{D}'(x) = \{5, 6, 7\}$.

If propagation does not lead to either a solution or a conflict, the solver selects a branching decision. While in SAT solving a decision assigns a truth value to a variable, in CP, decisions are expressed with atomic constraints, meaning they either assign a variable to a specific value or split its domain into subdomains. The resulting subproblems are then explored recursively.

Algorithm 1 presents a simplified depth-first CP solving algorithm.

Algorithm 1 A simplified depth-first solving algorithm for Constraint Satisfaction Problems

Input: A CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$

```
1:  procedure Solve(P)
2:      P' ← propagate(P)
3:      if ∃x_1 ∈ X', D'(x_i) = ∅ then
4:          return false                          ▷ Conflict occurred
5:      end if
6:      if all variables are assigned then
7:          return true                           ▷ A solution was found
8:      end if
9:      Generate subproblems P_1, P_2, ..., P_k
10:     for i ← 1 to k do
11:         if Solve(P_i) then
12:             return true
13:         end if
14:     end for
15:     return false
16: end procedure
```

The effectiveness of a CP solver heavily depends on the strength of propagation and the quality of the branching strategy. There are numerous heuristics for choosing variables and values to branch on, such as *smallest domain*, *smallest / largest value*, *random*, or activity-based heuristics inspired by *VSIDS* [30] from SAT solving. The order in which variables are considered can also be provided by the specific CSP instance.

Solving a COP can be viewed as solving several CSPs in succession. Each time the solver reports a solution that satisfies the constraints, we add a new constraint requiring the next solution to have a lower objective value. Once the solver states that the problem is infeasible, we know there is no better solution, thereby proving that the last solution found has the lowest possible objective value and is the optimal one.

2.2.3. Decision Levels

During the search, every branching decision introduces a new *decision level*. The root node of the search tree corresponds to decision level 0. Whenever the solver branches, the decision level is incremented, and all propagations caused by that decision (directly or indirectly) are associated with the same level. When the solver backtracks, the decision level decreases accordingly.

Decision levels allow the solver to track how and when each atomic constraint became true during the search. In particular, this information is used during conflict analysis to identify the causes of conflicts.

2.3. Lazy Clause Generation

Lazy Clause Generation (LCG) [14, 32] combines finite-domain CP propagation with conflict analysis techniques from SAT solving. It can be viewed as a hybrid between the CP paradigm and Conflict-Driven Clause Learning (CDCL) [28].

Traditional CP solvers rely primarily on propagation, and they backtrack only one decision at a time. Although propagation can be very strong, pure backtracking may repeatedly revisit similar failing regions of the search space. LCG addresses this issue by learning *nogoods* from conflicts, preventing the solver from entering regions we know to contain no valid solutions.

There are two key ideas behind LCG. Firstly, each variable domain is represented by a set of Boolean literals corresponding to atomic constraints, e.g. $l_1 \leftrightarrow [x \leq v]$ or $l_2 \leftrightarrow \neg[x \leq k]$. Secondly, every propagation performed by a CP propagator must be explainable through a conjunction of atomic constraints. The incorporated SAT engine then utilises these explanations (via their corresponding Boolean literals) during conflict analysis.

2.3.1. Explanations

Whenever a propagator removes a value from a domain, the solver records an explanation describing the logical reason why the removal occurred. Explanations are expressed as conjunctions over literals representing atomic constraints, and are posted to the SAT engine, which uses them to build an *implication graph* over the propagated atomic constraints.

Example 3. *Recall Example 2. A CP propagator inferred* $[x \geq 5]$ *to satisfy a constraint* $x + y \geq 7$ *when* $\mathcal{D}(y) = \{0, 1, 2\}$*. The explanation for this propagation is*

$$[y \leq 2] \rightarrow [x \geq 5]$$

2.3.2. Conflict Analysis and 1UIP

When a conflict arises – that is, when a variable's domain is wiped out – the solver initiates *conflict analysis* to determine why the conflict occurred and to derive a *nogood*: a conjunction of atomic constraints, describing assignments that cannot be part of any solution. This nogood is then added to the *nogood database*, and it prevents the solver from revisiting similar conflicting states. It does so via *unit propagation*, a scheme similar to the *unit rule* from SAT solving, which triggers a propagation whenever all but one of its atomic constraints become true, propagating the negation of the remaining one.

Example 4. *Suppose that in the nogood database, there is a nogood*

$$[x \geq 5] \wedge [y \geq 7] \wedge [z \geq 4] \rightarrow \bot$$

and that the current domains are $\mathcal{D}(x) = \{5, 6\}$*,* $\mathcal{D}(y) = \{7\}$ *and* $\mathcal{D}(z) = \{3, 4\}$*, meaning that* $[x \geq 5]$ *and* $[y \geq 7]$ *are true. To prevent the solver from entering the infeasible part of the search space described by this nogood, we propagate the negation of the remaining atomic constraint, namely* $[z \leq 3]$*, obtaining* $\mathcal{D}'(z) = \{3\}$*.*

The most widely adopted conflict analysis procedure is based on the *First Unique Implication Point* (1UIP) scheme [28, 30]. It works by analysing the implication graph – a directed graph in which nodes correspond to atomic constraints and edges (a_1, a_2) mean that atomic constraint a_2 was in the explanation for the propagation of the atomic constraint a_1. The analysis traverses this graph from the

conflicting state, replacing constraints at the current decision level with their explanations one at a time, until it reaches a state in which exactly one atomic constraint in the derived nogood belongs to the current decision level. The nogood obtained in this way is called a *learned nogood*.

The learned nogood also determines the *backjump level* – the highest decision level among all atomic constraints in the new nogood, excluding the current decision level. The solver backtracks to this level, at which point the learned nogood immediately triggers a unit propagation. This non-chronological backjumping (backtracking multiple decision levels) allows the solver to avoid exploring large irrelevant parts of the search space.

2.3.3. Nogood Management

While each learned nogood provides the solver with additional information about the problem structure, maintaining a large database of nogoods has costs. There are efficient systems for detecting when a nogood can unit propagate, such as the *two-watcher scheme* [30], but the amount of work needed still grows with the database size. Similarly, storing more nogoods requires more memory, potentially leading to *out-of-memory* errors. In practice, an unbounded nogood database can significantly degrade solver performance on larger instances, as the overhead of managing and checking nogoods outweighs the benefit of the information they provide [20].

To address this, solvers employ *nogood management* – a process in which nogoods are periodically evaluated and those deemed less useful are removed. The typical approach is to assign each nogood a *quality score* according to some heuristic, sort them by this score, and delete the worse-ranked half [3, 7, 19]. There are other approaches as well; for example, the nogoods might be divided into *tiers*, with the removal procedure in each tier carried out independently of the others, and using differing deletion rules and allowed tier sizes [24, 31].

2.4. Proofs

Conflict analysis and learned nogoods are closely connected to *proof logging*. A *proof of infeasibility* consists of a sequence of reasoning steps demonstrating that the original problem has no solution. Similarly, a *proof of optimality* shows that there is no solution with a better objective value. Proof logging is particularly important for applications where solver correctness is critical, such as chip design [2] or combinatorial auctions [10].

A simple approach to proof logging is to record every nogood and propagation step that the solver performs [17]. However, this approach incurs significant overhead in both runtime and required disk space. Recently, Flippo et al. proposed a multi-stage proof logging approach [15], in which only the learned nogoods are recorded during solving, resulting in a *scaffold*. Afterwards, the scaffold is *trimmed* by removing all the nogoods that were not strictly necessary to derive the final conclusion. Lastly, necessary propagation steps are reintroduced, resulting in a *full proof*. This approach reduces the burden on the solver and yields smaller proofs.

3

Related work

This chapter provides an overview of different heuristics and techniques related to nogood management used in various solvers. It examines both metrics for measuring the quality of learned clauses and nogoods, as well as approaches to limiting the size of the nogood (or clause) databases, showing how the state-of-the-art schemes have evolved over time.

3.1. Nogood quality metrics

Research on heuristics for assessing the quality of learned clauses was primarily conducted by the SAT-solving community. Many of the metrics were initially proposed in SAT solvers; for example, GRASP [28] removes all clauses that are bigger than a specified threshold and contain more than one unassigned literal. Chaff [30] also monitors the number of unassigned literals in clauses and deletes those that exceed a given limit. MiniSat [11] took a different approach and measures the activity of clauses. Each time a learned clause participates in conflict analysis, its activity is increased. Additionally, after each conflict, the activity of all saved clauses is multiplied by a constant smaller than 1, meaning that the activity stays high only for the clauses that recently took part in conflict analysis. All other ones are periodically removed.

In 2009, Audemard and Simon introduced Literal Block Distance (LBD) in their solver Glucose [3]. The metric measures the number of distinct decision levels of literals in the clause. The intuition behind this approach is that literals from one decision level likely have dependencies among themselves and will probably be propagated together in the future. Therefore, we would like to minimise the number of such groups of variables in the clause, in an attempt to make it unit propagate as soon as possible. The authors empirically showed that Glucose clearly outperformed other solvers developed at the time.

LBD became a state-of-the-art metric used in many solvers and has been the subject of further research. For example, Oh [31] observed that Glucose tends to discard clauses with high LBD quickly and relies mostly on the low-LBD ones. The author then proposed to divide learned clauses into low-LBD *core* clauses that are never removed, *local* clauses with higher LBD that are stored only while their activity is high (to improve search in the current branch), and *mid-tier* that act as a bridge between the other two.

In addition to research on LBD, other metrics were developed, such as *nbSAT* [26]. It ranks clauses based on the number of literals satisfied by the current partial assignment or satisfied by the saved assignment in *phase saving*, a technique that remembers the last assigned value for a variable during backtracking and reuses this value if the variable is chosen again. The authors demonstrated experimentally that a solver combining nbSAT with LBD performs better than one using only LBD.

In the context of Constraint Programming, the metrics for ranking nogoods were mostly imported from SAT solving, for example, Pumpkin [15] uses the *tiered* variant of LBD, and Chuffed [7] uses activity. These heuristics, however, do not take advantage of additional information we have in CP – namely, that we can connect the literals to integer variables and values. Because of this, the authors of NACRE [19]

decided to use *number of variables* as a heuristic, with fewer variables implying better nogood, and to use activity as a secondary tie-breaking measure. However, they did not provide any empirical analysis of this choice.

3.2. Nogood removal schemes

The development process of schemes for the removal of nogoods was similar to that of nogood quality heuristics. The first systems used in SAT solving, like GRASP [28], involved deleting learned clauses as soon as they satisfied a specific condition. Later, more aggressive reduction schemes were developed, with a policy of periodically removing many clauses in bulk. MiniSat [11] and Glucose [3] removed half of all stored clauses once the size of the database exceeded a given threshold.

The notion of deleting half of the stored clauses was used by Oh [31] in their *tiered* database management system to remove clauses from the *local* tier. The *core* tier was never reduced, while the clauses from the *mid-tier* were moved to the *local* one if they had not participated in conflict analysis in the past 30000 conflicts. This tiered system was later improved by Kochemazov [24] by adding size limits for the *core* and *mid-* tiers, and removing half of the clauses stored in those tiers as soon as the limits were violated.

These schemes were later imported into Constraint Programming. NACRE [19] and Chuffed [7] reduce their nogood databases by half whenever their sizes reach a specific limit. Pumpkin [15] adopted the *tiered* system, including the improvements from [24], while slightly altering the specific values for tier size thresholds.

A caveat to consider in relation to removal schemes for clauses and nogoods is that each database reduction causes the solver to lose learned information. The removed nogoods may be necessary to solve the problem instance, so the solver tries to re-learn them, potentially triggering another database clean-up that again removes the freshly learned nogoods. This way, the solver might enter a loop and fail to solve a given instance. There are two main ways in which solvers attempt to deal with this problem. Some of them, like Chuffed or Pumpkin, set the size thresholds relatively high, hoping that it is enough to solve the problem. Others, including NACRE and Glucose, start with smaller size limits and increase them after each nogood removal, ensuring that the database eventually grows large enough to store all the required learned constraints.

4

Nogood quality metrics

The first aspect we discuss in this thesis is the usage of nogood quality metrics and their ability to distinguish good and bad nogoods. The goal is thus to find whether there are measures that can predict a given nogood's usefulness and determine if it should be kept in the database. To that end, we propose and describe several metrics (Section 4.1), define how to measure the usefulness of a nogood (Section 4.2), and evaluate the metrics on a wide range of optimisation and satisfaction problems. The results reported in Sections 4.4-4.6 indicate that each of the metrics we study can (to some extent) forecast whether a nogood is likely to trigger propagations, and that some of these metrics are strongly correlated with one another. Because of these correlations, in Section 4.7 we identify four metrics that are likely to assess nogoods differently, meaning they are the most promising candidates for further research.

4.1. Considered metrics

Firstly, we take four metrics as proposed in the literature. While this list is not exhaustive, we tried to pick the most common and influential ones. *Size*, *activity*, and *Literal Block Distance* were introduced in some of the most prominent research papers in the field, and thus got adopted by most of the popular SAT and CP solvers. *Number of variables*, to our knowledge, is the only metric used in practice that utilises information available only in CP but not in SAT solving.

1. **Size** [28] – counts number of atomic constraints in a nogood. Shorter nogoods are generally more desirable, as they require fewer atomic constraints to be satisfied before they can unit propagate. Nogoods of size equal to one (consisting of a single atomic constraint, also called *unit nogoods*) are highly preferable, as they can propagate already at the root level.
2. **Activity** [11] – measures how often a given nogood has recently participated in conflict analysis. The intuition is that if a nogood is repeatedly used to derive conflicts, then it will likely be helpful in future conflicts in the current part of the search space. Each time a nogood participates in conflict analysis, its activity is increased by a fixed amount d. Additionally, the activities are decayed to reduce the values for nogoods that were active in a distant part of the search tree, but are no longer relevant. In our case, this process of decay is implemented by increasing the amount d after each conflict, and when the activity of any nogood crosses a predefined threshold, the activities of all nogoods are divided by a large constant to avoid overflow errors.
3. **Literal Block Distance** (LBD) [3] – counts how many distinct decision levels are there among the atomic constraints of the nogood. The reasoning behind this metric is that atomic constraints from one decision level were all propagated after a single decision, meaning there might be direct dependencies between them, and they are likely to be propagated together again in the future. It is therefore useful to have such groups of linked atomic constraints in nogoods, as one decision can assign a truth value to many of them, bringing the nogood closer to unit propagation. We thus prefer nogoods with low LBD, as they have few groups of such interconnected atomic constraints, meaning they are likely to trigger unit propagation often. Since atomic constraints can become

assigned at different decision levels during the search, LBD of a single nogood can change over time. We recompute this metric for a given nogood each time it participates in conflict analysis, and update it only if we obtain a lower value than before.

4. **Number of variables** [19] – counts the number of distinct integer variables occurring in atomic constraints of the nogood, with the assumption that nogoods related to few variables are more powerful and unit propagate more often.

In addition, we propose four new metrics. Given the widespread success of LBD, *decision levels span* is our attempt to augment that metric with additional information and improve its performance. The next one, *search space size*, is designed to use information available only in CP, other than *number of variables* – it takes into account also the domains of variables. Lastly, the two versions of *constraints count* measure interactions between the constraints used in CP models.

5. **Decision levels span** – calculated as the number of decision levels between the 'oldest' atomic constraint from the nogood and the current decision level. This idea is an extension of LBD (the value of this metric is always greater than or equal to LBD), where, in addition to having the fewest distinct decision levels, we also want the nogood to propagate as soon as possible after some of its atomic constraints become true. Generally, the decisions can be completely independent of each other, meaning that even if a given nogood quickly propagates at one point, it does not necessarily need to do so again in the future. However, when the branching heuristic is specifically structured, such as the *fixed search* (which selects variables in a predetermined order), the information captured by this metric could become useful and applicable. Similar to LBD, we recalculate this metric each time the nogood participates in conflict analysis and update it only if it decreases.
6. **Search space size** – represents the size of the search space the nogood invalidates, calculated as a fraction of all possible *complete* assignments that the nogood rules out based on the initial domains of the problem's variables. Intuitively, we want our nogoods to be as general as possible and greatly reduce the size of the search space left to explore.
7. **Constraints count** – number of unique constraints whose propagations participated in the creation of a given nogood during the conflict analysis procedure. Here, we consider learned nogoods as separate constraints that can contribute to the count just as well as the CSP's initial constraints. One might expect that nogoods with high values for this metric can encapsulate information from many constraints, thus propagating more often and being more desirable. However, our preliminary experiments showed the opposite – nogoods with a low value of *constraints count* caused more propagations. This could be explained by the fact that nogoods combining information from many constraints are likely to be very specific and useful in limited parts of the search space. Therefore, in our analysis, we consider low values as preferable for this metric.
8. **Recursive constraints count** – similarly to *constraints count*, this metric also represents the number of constraints that participated in the creation of a given nogood. However, this time we consider only the problem's initial constraints. When constructing a nogood, we keep track of the set of constraints used, and if during conflict analysis we encounter a propagation made by another nogood, we recursively expand this set with constraints used to derive that propagating nogood. By doing so, we ensure that this metric takes into account only the CSP's original constraints.

4.2. Nogood usefulness

Before we can attempt to find which metrics can predict which nogoods should be kept in the database, we first need a way of distinguishing good from bad nogoods. While this is impossible to do while the solver is running, as at that point we only have partial data on nogood usage, one possible approach is to analyse the performance of nogoods across the entire solving process after the solver has finished running. In other words, given the path that the solver took through the search space, we want to define a measure of how useful a given nogood was.

4.2.1. Weighing propagations

A straightforward way of determining usefulness would be to check whether a given nogood ever made any unit propagations, and how many of those. After all, nogoods' job is to mark the infeasible parts of the search space and prevent the solver from repeatedly entering them. Each time a nogood performs its task, it does so by making a propagation pruning the corresponding invalid assignments. Therefore, we can use the number of propagations as a proxy for the amount of work that a given nogood performed.

However, some of these propagations might not be highly influential. We are most interested in those that lead to further conflicts, as they allow us to learn new information about the problem. Thus, in an attempt to prioritise nogoods that cause further conflicts, we can weigh the propagations from nogoods by the number of times they were used during the conflict analysis procedure.

Apart from examining conflicts, we can also consider proofs of optimality and unsatisfiability. It could be that some conflicts and learned nogoods were not strictly necessary to solve the problem. We thus consider the nogoods appearing in the trimmed proof. Although this is not necessarily the smallest possible set of nogoods required to solve the problem, such filtering is likely to yield the most impactful nogoods. Therefore, we also weigh nogood propagations by the number of times they were used to derive nogoods that appeared in the trimmed proof.

Lastly, we can further restrict our notion of usefulness by slightly modifying the approach from [34] and defining the *useful nogoods* as those whose propagations are used in the trimmed proof more often than the average of all nogoods from the proof. Then we consider only these *useful nogoods* and their propagations, weighing them, as previously, by the number of times they were used in conflict analysis.

4.2.2. Measuring and comparing nogood metrics

Above, we described four ways of determining which nogoods are useful, each more restrictive than the previous. Before we can move on to finding which metrics predict well which nogoods will be useful, we need to discuss two caveats.

Firstly, some of these metrics are updated over time, such as activity or LBD – for example, suppose we have a nogood that makes two propagations when its LBD is equal to 10, then its LBD is reduced to 5, and it makes one more propagation. It would then be wrong to count all three propagations as made by nogood with LBD 5. Instead, we should take into account the specific metric value at the time of propagation. Therefore, each time a nogood performs a propagation, we record its metrics at that moment, obtaining one data point. We then weigh these data points as described above, e.g. how many times the corresponding propagation was used in conflict analysis, or how many times it was used to derive nogoods in the trimmed proof. Using these weighted data points, we compute their distributions (separately for each metric), aggregate them across instances, and examine their shapes, means, standard deviations, and skewnesses.

Secondly, across different instances, the ranges of values that the nogoods can have for a given metric might vary drastically, depending on the problem structure and instance size. We attempt to solve it by computing and comparing *rankings* in addition to raw metric values. We define *ranking* of a nogood for a metric as the position where the given nogood would be placed if we were to order all the nogoods from the database by that metric. In this way, we can see if the current metric value of a nogood is relatively low or high compared to other stored nogoods. Specifically, we count the number of nogoods with a smaller metric value and the number of nogoods with a smaller or equal metric value, and calculate the ranking as

$$\text{ranking} = \frac{\#smaller + \#smallerOrEqual}{2 \cdot \#allNogoods}$$

This resulting ranking is a fraction in $(0, 1)$, with low values meaning that the nogood had a relatively small metric among all nogoods, while values close to 1 mean that the metric value was among the highest in the database.

Rankings computed in this way provide a convenient way of relatively comparing metric values in a given problem, swiftly deal with outliers, and can be useful for generalising our findings across many instances. However, they are not without their drawbacks. Example 5 describes a case in which

the way we calculate rankings can suggest that nogoods with a given ranking are more useful than they are in reality, simply because many nogoods have a particular metric value, resulting in the same ranking. We have observed this phenomenon in our experiments, especially in small instances in which discrete metrics (such as LBD or size) take only a handful of values. However, we also noted that this dominant behaviour diminishes when aggregating results across numerous instances, and thus believe that rankings can be used to draw valid conclusions.

Example 5. *Consider five nogoods with LBDs of* $[3, 6, 7, 8, 10]$. *If we now calculate their corresponding rankings, we obtain* $[0.1, 0.3, 0.5, 0.7, 0.9]$*, which are equally spaced, as the nogoods have unique LBD values. However, if some of the nogoods had the same LBD, e.g.* $[6, 6, 6, 8, 10]$*, then their corresponding rankings would be* $[0.3, 0.3, 0.3, 0.7, 0.9]$. *The first three nogoods, even though they have the smallest LBD of all, have a ranking of* 0.3*, suggesting that there might be other nogoods with smaller LBD. Moreover, if all of these nogoods were to make one propagation each (meaning they are all equally useful), and we looked at the distribution of rankings for these propagations without knowing the actual metrics the nogoods had, we might come to the misleading conclusion that nogoods with a ranking of* 0.3 *are more useful than the other ones.*

4.3. Experimental setup

We conducted a wide range of experiments to measure metrics of the learned nogoods and their relationship to the perceived usefulness of nogoods. We have run a Constraint Programming solver on numerous instances, 584 optimisation and 203 satisfaction ones, logging information about nogoods, their propagations, and the whole proof (in instances where this was possible). In these experiments, we turned off the nogood database reductions to avoid bias introduced in terms of preserved nogoods.

In this section, we describe the setup of our experiments – the software and hardware used, details of the data collection process, and problem instances we evaluated.

4.3.1. Software and hardware

We collect nogood metrics using an open-source CP solver Pumpkin[1] on commit `6020771`. Apart from recording and logging additional metrics, as well as disabling nogood database reductions, we did not make any further changes to the solver and used its default parameters. Our code is freely available online[2]. The solver was built using Rust 1.94.0. All problem instances were decomposed to FlatZinc using MiniZinc 2.9.7 and Pumpkin's custom configuration rules.

The experiments were conducted on the DelftBlue supercomputer [1] using the *compute-p2* partition, which is built with Intel Xeon Gold 6448Y 32C 2.1 GHz processors. Each problem instance was run in a separate process using 4GB of memory. Due to considerable performance overhead from logging nogood metrics and the whole proof, the timeout per instance was set to 2 hours.

4.3.2. Collected data

To summarise Section 4.2, when running a problem instance, each time a nogood propagates, we record its metrics at that time, specifically the raw value of the metrics and their *rankings*. These propagations are then weighted according to progressively stricter notions of usefulness:

- **Unweighted** - considers all the nogood propagations that occur during solving, giving each the same weight.
- **Conflict** - a given propagation is weighted by the number of times it takes part in conflict analysis.
- **Proof** - propagations are weighted by the number of times they are used to derive nogoods from the trimmed proof.
- **Useful proof** - propagations are weighted by the number of times they are used to derive nogoods from the trimmed proof, but we only assign this weight to propagations made by **useful proof nogoods** (where we defined *useful proof nogoods* as those whose propagations are used in the proof more often than the average of all nogoods in the proof).

[1] `https://github.com/ConSol-Lab/Pumpkin`
[2] `https://github.com/Hubert1913/Pumpkin/`

Having such (weighted) data points, we compute their density distributions and average them across instances. We look at averages, standard deviations, and skewnesses of these distributions to find which metrics best indicate whether a given nogood is likely to perform (useful) propagations. Ideally, we would like to find a metric whose distribution is highly skewed, such that most propagations are made by nogoods with the corresponding rankings close to 0 or 1.

An important aspect to consider regarding the last two weighing methods is that, when obtaining the trimmed proof, we did not use Pumpkin's proof processor, which reiterates parts of the solver's reasoning. Instead, we analysed the full proof produced during solving in a breadth-first manner, strictly following the path the solver took through the search space. While both approaches remove nogoods that were unnecessary to obtain the final contradiction, we observed that our method often produced a longer trimmed proof containing more nogoods. However, since it exactly follows the solver's steps, we believe it is better suited for analysing the performance of nogoods.

4.3.3. Selected instances

We measured nogood metrics on a wide range of optimisation and satisfaction problems by using all problem classes from all years of the MiniZinc Challenge [35, 36] available online in the archive repository[3]. As mentioned in Subsection 4.3.1, the time limit for each instance was set to 2 hours. Additionally, we filtered out all instances solved in under 20 seconds, as they are likely too simple to provide us with valuable generalisable information or contain a specific structure that the solver manages to exploit.

There were 135 unique optimisation problems. From each, we randomly picked four or five instances, resulting in 584 instances. 81 of those were solved within the time bounds.

As Constraint Programming is mostly used in practice for solving problems to optimality, the MiniZinc Challenge repository contains far fewer satisfaction problems. There were only 27 problem classes, and from each we randomly picked 10 instances (if fewer were available, we selected all of them). This way, we obtained 203 instances. After filtering based on time bounds, 33 instances remained, all of which were satisfiable.

Apart from aggregating many problem classes and generalising our findings, we also check whether the observations differ when we look at a specific type of problems, for example, the *scheduling* family, which is a major application area of CP. We picked models from the MiniZinc Benchmarks[4] repository, primarily focusing on *rcpsp* and *jobshop*, as well as their modifications, such as *cyclic-rcpsp* or *flexible-jobshop*. Additionally, we included several related scheduling problems, like *bus-scheduling*, *roster*, and *ship-schedule*. From most of the problem classes, we randomly selected 10 instances (if there were fewer available, we picked all of them). Since *rcpsp* and *jobshop* can be considered as the fundamental problems in this family, we selected 25 instances from each of them. In total, this resulted in 145 optimisation instances. Only 14 of them were solved within the time bounds.

4.4. Optimisation

We begin our analysis by looking at the density distributions of the metrics across optimisation instances. We will not discuss all the metrics here, as their distribution shapes are very similar to each other, but rather focus only on LBD and activity, as these two already allow us to make several observations that are generally applicable to other metrics as well. The density distribution plots for all the metrics can be found in Appendix A.

LBD

Already by looking at Figure 4.1, we can see a feature common to all metrics – the density distribution of ranking is highly skewed, with a clear peak at low values (in the case of activity and search space size, the peak is at high values). There is a tail, meaning that nogoods with relatively high LBD can also cause some propagations, but significantly fewer than the low-LBD ones. Moreover, as we use stricter notions of nogood usefulness, the size of this tail gets smaller, with almost no nogoods with LBD ranking higher than 0.8 making propagations in the *useful proof* category.

[3] https://github.com/MiniZinc/mzn-challenge
[4] https://github.com/MiniZinc/minizinc-benchmarks

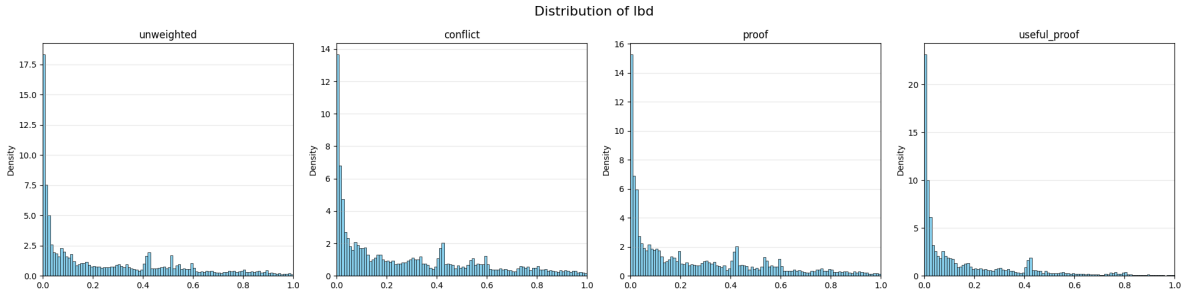

Figure 4.1: Density distribution of rankings for LBD, split into four approaches of weighing propagations.

Figure 4.2 presents the density distribution of raw LBD values among the nogoods performing propagations. While across all instances, there were some propagations made by nogoods with LBD of up to 1000, we can see that already the ones with LBD of 20 perform hardly any propagations. Additionally, around 70-75% (depending on the weighing approach) of all propagations are performed by nogoods with LBD smaller than or equal to 5. This aligns with Oh's [31] observation for SAT solving that learned clauses do not contribute greatly in terms of solving the problem, unless they have a very low LBD.

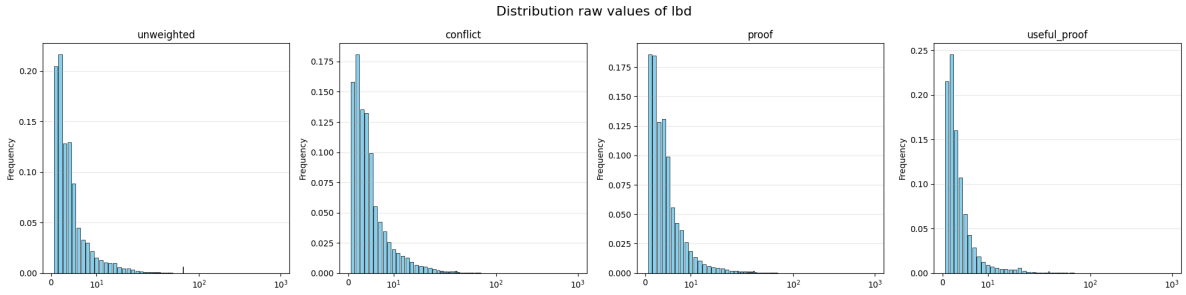

Figure 4.2: Density distribution of raw values for LBD, split into four approaches of weighing propagations. X-axis is in *symlog* scale, with the linear threshold at 20.

Activity

In contrast to LBD, most propagations are done by nogoods with high ranking for activity, as can be seen in Figure 4.3, with an even more pronounced peak than in the case of LBD. Decreasing the ranking value, we can observe that there are almost no propagations already at the ranking of 0.8. However, there is a sudden spike in density at 0.5, and a non-negligible increase around the ranking of 0.

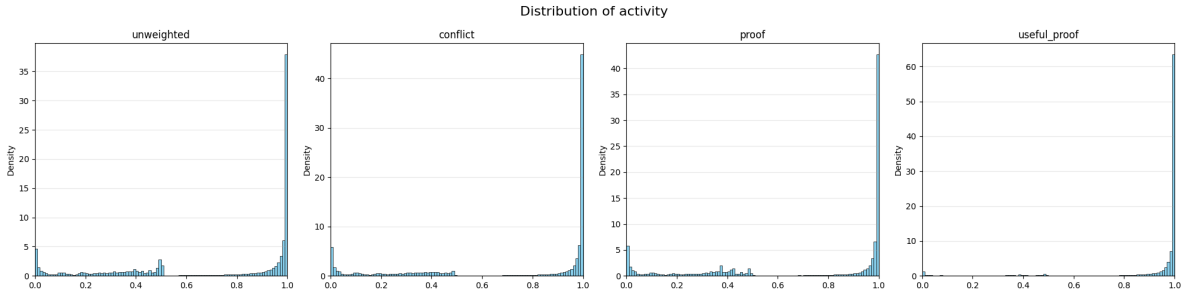

Figure 4.3: Density distribution of rankings for activity, split into four approaches of weighing propagations.

There is a similar phenomenon in the plots with raw values depicted in Figure 4.4. A significant number of propagations were made by nogoods with activity of 10^{-35}. Since this plot uses a logarithmic scale and the values are clamped to 10^{-35}, we believe that most of them, in reality, have activity of 0 (as confirmed by the results from several instances we inspected manually). This happens for propagations

made before their corresponding nogoods participated in conflict analysis for the first time, or when the nogood was used so rarely that its activity was heavily reduced and truncated to 0.

As mentioned in Example 5, when there are numerous nogoods with the same metric value, their corresponding ranking can be artificially inflated. This fact is the likely explanation for the density mass in the plots in Figure 4.3 for rankings below 0.5, as in the extreme case, when all the nogoods in the database have the same metric value, their ranking is equal to 0.5. This also explains why there are seemingly no propagations made by nogoods with activity ranking between 0.5 and 0.8.

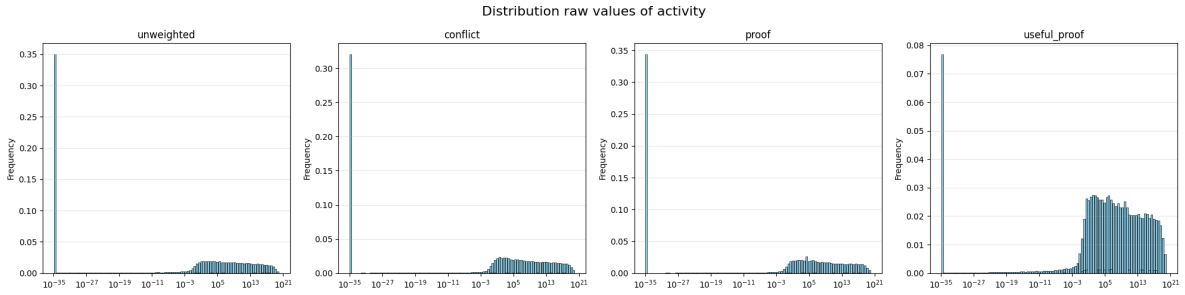

Figure 4.4: Density distribution of raw values for activity, split into four approaches of weighing propagations. X-axis is in log scale, with values clamped at 10^{-35}.

While the plots with rankings show that nogoods with relatively highest activity perform the most propagations, there are no corresponding peaks in Figure 4.4 – the distribution is much more uniform compared to the one for LBD in Figure 4.2. This is likely because of the way we calculate activity of nogoods in our implementation – the activity bump for participation in conflict analysis is constantly increasing, and reductions occur by dividing activities of all nogoods by a large constant, meaning the activity value at a given time point is meaningful only in comparison to other nogoods. A particular value might, in one moment, correspond to low-activity nogoods, but in another, a reduction occurs, and the same value can be considered high activity.

In the *useful proof* plots of Figure 4.3 and Figure 4.4, it can be seen that the tail in the rankings plot and the spike at 10^{-35} in the raw values plot are significantly reduced. It shows that this propagations weighing type filters out many nogoods that perform only a single inference upon their creation and do not participate further in the solving process, or are active very rarely. This means that activity can be a valuable metric for finding the core nogoods that are the most impactful.

Rankings statistics

Apart from inspecting the distributions visually, we can also attempt to summarise them using their statistics. Table 4.1 presents the mean, standard deviation and skewness of distributions of rankings of the metrics. We are looking for a metric for which the data points are strongly concentrated on one side, so for ease of comparison, we transformed activity and search space size using $1 - x$. This way, a good metric should have a low mean, low standard deviation and a high skewness. In the table, the two best values in each category were bolded.

There are three main observations we can make based on Table 4.1. Firstly, LBD performs well overall, consistently being one of the top two metrics across all weighing approaches. Secondly, the lowest mean and the highest skewness overall are for activity in the *useful proof* case. However, activity does not perform that well in other cases, likely due to the tail in its distribution, as we have discussed previously. Lastly, some groups of metrics, such as size and number of variables, or the two types of constraints count, have similar values for the three statistics across all propagation weighing schemes. This suggests that some metrics are highly correlated, assigning nogoods very similar rankings.

Table 4.1: Summary statistics derived from density distribution data. Bolded are the two best values in each category (lowest for Mean/Std, highest for Skewness).

Type	Metric	Mean	Std	Skewness
unweighted	size	**0.2591**	**0.2855**	**0.9749**
	activity	0.2909	0.3548	0.7795
	lbd	**0.2526**	**0.2677**	0.9309
	num_variables	0.2607	0.2865	0.9536
	decision_levels_span	0.2750	0.2852	0.8525
	search_space_size	0.2621	0.2907	**1.0037**
	constraints_count	0.3259	0.2937	0.7094
	constraints_count_recursive	0.3117	0.2928	0.7970
conflict	size	0.3093	0.3049	0.7558
	activity	**0.2767**	0.3709	**0.8901**
	lbd	**0.2776**	**0.2733**	**0.8271**
	num_variables	0.3070	0.3029	0.7535
	decision_levels_span	0.2863	**0.2807**	0.8143
	search_space_size	0.3091	0.3060	0.7540
	constraints_count	0.3659	0.3052	0.5486
	constraints_count_recursive	0.3614	0.3091	0.5821
proof	size	0.3121	0.3106	0.7615
	activity	0.2885	0.3703	0.8093
	lbd	**0.2525**	**0.2611**	**0.9498**
	num_variables	0.3116	0.3101	0.7590
	decision_levels_span	**0.2653**	**0.2709**	**0.9280**
	search_space_size	0.2996	0.3045	0.8084
	constraints_count	0.3854	0.3213	0.5248
	constraints_count_recursive	0.3764	0.3201	0.5350
useful_proof	size	0.1853	0.2502	1.5688
	activity	**0.1189**	0.2729	**2.4627**
	lbd	**0.1619**	**0.2164**	**1.6177**
	num_variables	0.1829	0.2469	1.5464
	decision_levels_span	0.1874	**0.2435**	1.4824
	search_space_size	0.2159	0.2880	1.4970
	constraints_count	0.2743	0.2837	1.0125
	constraints_count_recursive	0.2542	0.2714	1.1484

Correlations

To verify the observation about connected metrics, we calculated correlations between the metric rankings, measuring the similarity in how different metrics rank the nogoods. These correlations are presented in Figure 4.5. The rankings for activity and search space size were again transformed with $1-x$, so that, in all cases, a high correlation means the metrics agree on whether a given nogood is valuable.

We can immediately see that the correlations are all positive, meaning there are no two metrics judging nogoods in completely different ways. This fact could be somewhat expected – a smaller nogood has fewer atomic constraints, so fewer occasions to introduce new variables, and fewer restrictions on the invalidated part of the search space. Fewer atomic constraints also mean there are fewer opportunities to introduce new decision levels (increasing LBD) or to increase the decision levels span. Shorter nogoods likely required fewer conflict analysis steps during their creation, so there were fewer opportunities for different constraints to participate. However, we cannot make such a simple connection between activity and the other metrics. And indeed, activity shows a lower correlation, most noticeably in the *useful proof* case.

At the same time, the correlations are not all equal to 1. This means that there are nogoods which

get divergent rankings from different metrics. Thus, we could potentially use these differences and combine two or more metrics, in the hope that they provide new information about nogoods.

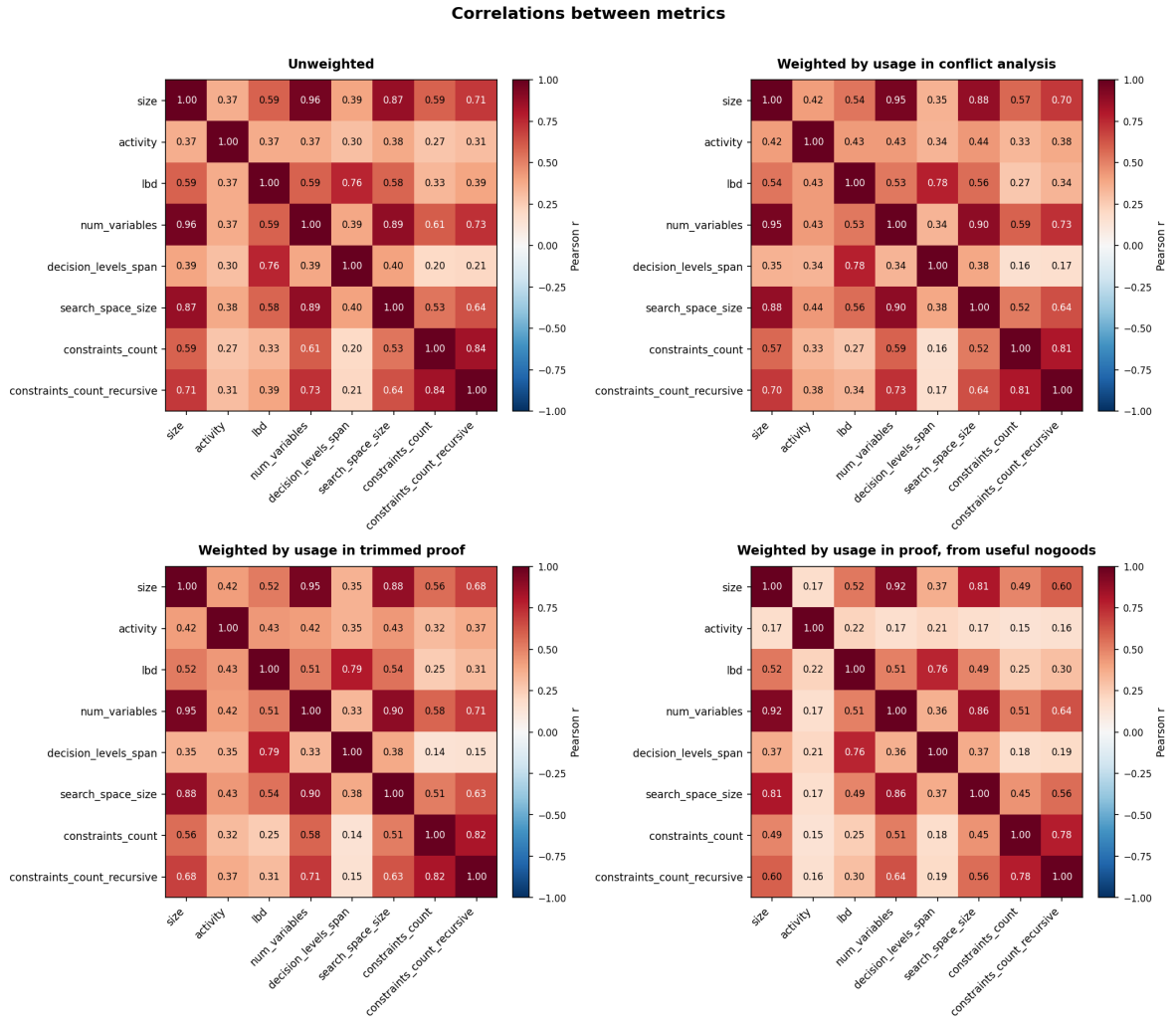

Figure 4.5: Correlations between the rankings assigned to nogoods by the metrics, split by approaches to weighing propagations.

As we have observed previously, some of the metrics are highly correlated with each other. We can distinguish four groups:

1. Size, number of variables, search space size
2. LBD, decision levels span
3. Constraints count, recursive constraint count
4. Activity

Propagations from the ‘better’ half

Lastly, in an attempt to quantify the quality of the metrics with a single number, we calculate the percentage of propagations made by nogoods with ranking smaller than or equal to 0.5 (correspondingly greater than or equal to 0.5 for activity and search space size). Many of the common nogood database reduction schemes work by ordering nogoods and removing the ‘worse’ half; thus, by calculating this percentage, we estimate how many propagations were made by nogoods that would be retained after a hypothetical database reduction. Figure 4.6 presents these values for all metrics, split by the weighing approaches.

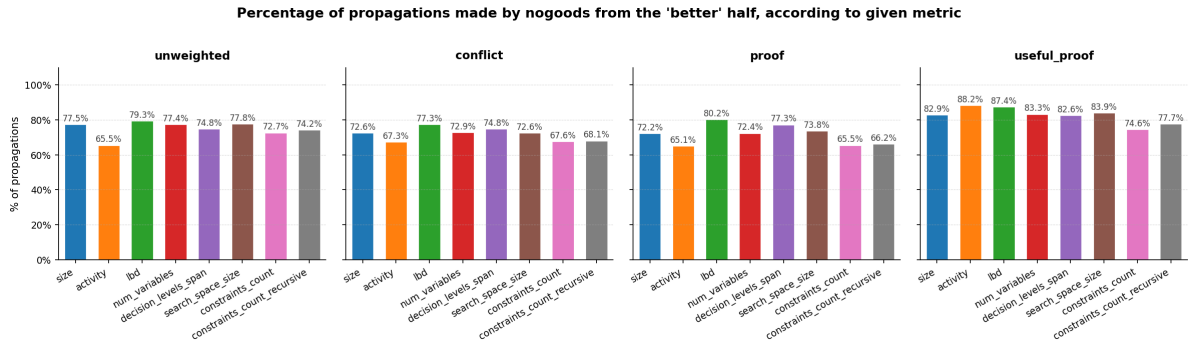

Figure 4.6: Percentages of propagations made by nogoods with a ranking of less than or equal to 0.5 (in case of activity and search_space_size – more than or equal to 0.5).

The results align with our previous observations in Table 4.1 – LBD performs overall well, only giving way to activity in the *useful proof* case. Activity, on the other hand, in the three remaining weighing approaches correctly classifies the fewest nogoods. Additionally, within the clusters of highly correlated metrics we have identified, the corresponding metrics exhibit very similar performance, regardless of how the propagations are weighted.

4.5. Satisfaction

Solving optimisation instances can be viewed as repeatedly finding a series of solutions that get progressively harder to uncover, and proving that no better solution exists at the end. In contrast, when solving a satisfaction instance, we only need to find a single solution, and do not pay attention to its objective value. This means that the solver's profile and characteristics might need to be completely different to excel in satisfaction instances, compared to optimisation ones.

In the following section, we repeat our analysis of nogood quality metrics from Section 4.4, this time using satisfaction instances. As mentioned in Subsection 4.3.3, all the instances solved within the time bounds were satisfiable, which means we could not produce a proof for them. Because of that, we only consider here the *unweighted* and *conflict* approaches of weighing propagations.

LBD

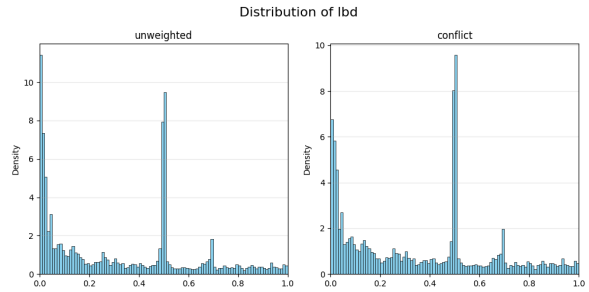

Figure 4.7: Density distribution of rankings for LBD, split into two approaches of weighing propagations.

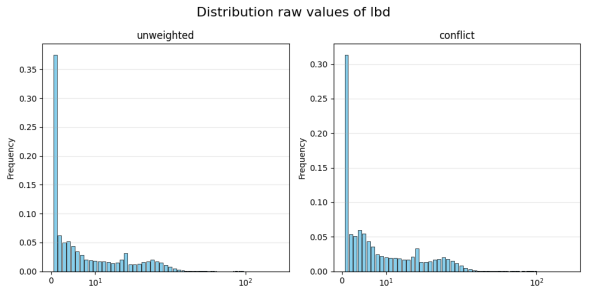

Figure 4.8: Density distribution of raw values for LBD, split into four approaches of weighing propagations. X-axis is in *symlog* scale, with the linear threshold at 20.

As we can see in Figure 4.7 and Figure 4.8, nogoods with low LBD values again tend to make more propagations. However, compared to optimisation, there seem to be many more nogoods with an LBD of 1. At the same time, there is a significant spike in the density distribution of rankings around the value of 0.5. Recalling Example 5, these results suggest that several instances had a specific structure that resulted in (almost) all nogoods having an LBD of 1, thereby skewing the distributions.

Activity

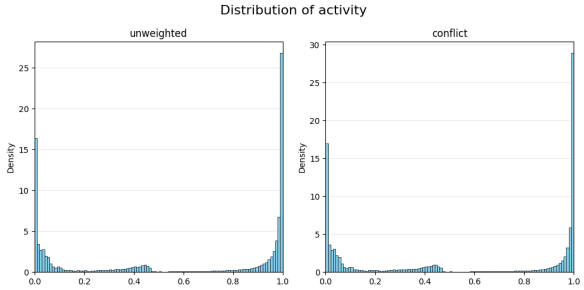

Figure 4.9: Density distribution of rankings for activity, split into two approaches of weighing propagations.

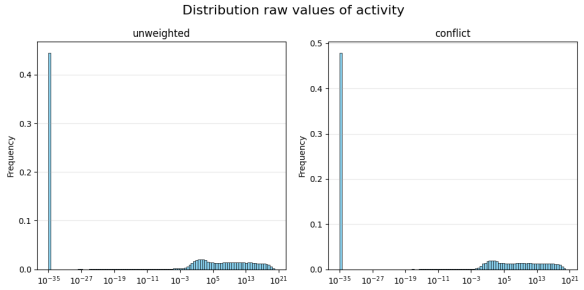

Figure 4.10: Density distribution of raw values for activity, split into two approaches of weighing propagations. X-axis is in log scale, with values clamped at 10^{-35}.

The distributions of rankings and raw values of activity presented in Figure 4.9 and Figure 4.10 are generally similar to the ones from optimisation instances. However, there are noticeably more propagations made by nogoods with very low rankings. This could be because the solver does not need to explore the whole search space, as in the case of optimisation, but instead stops after the first solution, likely leaving a significant part of the search tree unexplored. This means there was no chance to run into areas invalidated by many of the stored nogoods, so they had no opportunities to make unit propagations that would boost their activity score.

Rankings statistics

Table 4.2 presents the mean, standard deviation, and skewness of rankings of metrics. Compared to optimisation instances, LBD does not have the best statistics, with the trio of size, number of variables and search space size outperforming it. This also shows that the correlated metrics again have very similar performance to each other. Surprisingly, activity performs significantly worse compared to optimisation, having the highest mean and standard deviation and the lowest skewness among all metrics.

Table 4.2: Summary statistics derived from the density distribution data. Bolded are the two best values in each category (lowest for Mean/Std, highest for Skewness).

Type	Metric	Mean	Std	Skewness
unweighted	size	**0.2403**	**0.2871**	**1.0928**
	activity	0.4278	0.4352	0.2794
	lbd	0.3181	**0.2855**	0.5373
	num_variables	**0.2440**	0.2898	**1.0765**
	decision_levels_span	0.3434	0.2934	0.4433
	search_space_size	0.2515	0.2947	1.0371
	constraints_count	0.3696	0.3484	0.6677
	constraints_count_recursive	0.3523	0.3490	0.7378
conflict	size	**0.2814**	0.2981	**0.8615**
	activity	0.4500	0.4392	0.1677
	lbd	0.3549	**0.2842**	0.4005
	num_variables	**0.2850**	0.2998	**0.8502**
	decision_levels_span	0.3766	**0.2899**	0.3298
	search_space_size	0.2925	0.3037	0.8108
	constraints_count	0.4084	0.3433	0.5044
	constraints_count_recursive	0.3924	0.3463	0.5678

Correlations

Figure 4.11 shows the correlations between rankings of metrics. The values are very similar to those for optimisation, with all correlations being positive. Moreover, we can clearly identify the same four groups of highly correlated metrics.

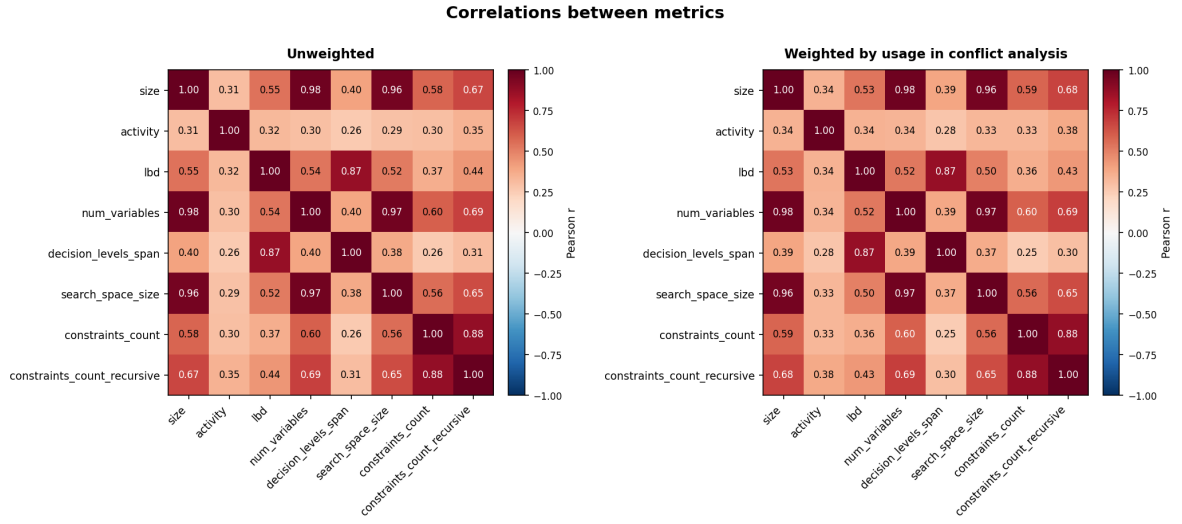

Figure 4.11: Correlations between the rankings assigned to nogoods by the metrics, split by approaches to weighing propagations.

Propagations from the 'better' half

Figure 4.12 confirms the observations from Table 4.2. Activity performs very poorly in predicting which nogoods cause the most propagations. This situation was similar in the *unweighted* and *conflict* cases in optimisation, but there the difference between activity and other metrics was not that stark. LBD also performs worse here, barely improving over the two types of constraints count, which were consistently the two worst-performing metrics in optimisation instances.

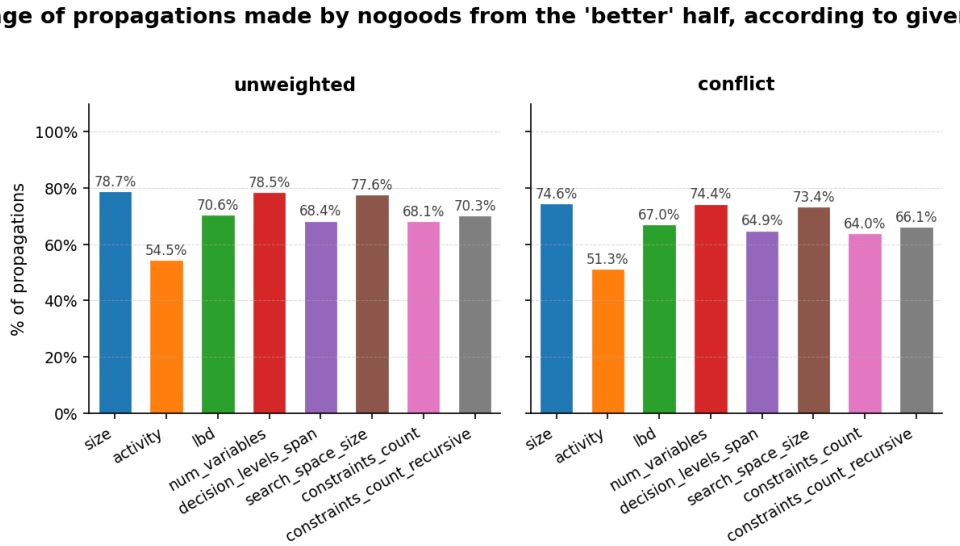

Figure 4.12: Percentages of propagations made by nogoods with a ranking of less than or equal to 0.5 (in case of activity and search_space_size – more than or equal to 0.5).

4.6. Scheduling

The analysis in Section 4.4 and Section 4.5 was based on a wide range of problem classes, with the aim of examining the performance of metrics in general. In this section, we verify whether the same observations hold when we consider only one particular type of problems. We chose to examine the *scheduling* family of problems.

The main observation in this regard is – yes, the remarks we made when studying the general optimisation instances largely hold here as well. There are several small differences, which we point out next. However, it is worth noting that they could largely be explained by the fact that this analysis is based on only 14 instances that were solved within the time bounds. It is likely that if there were more instances, the results could differ.

Firstly, Figure 4.13, which presents the density distribution of rankings of LBD, shows that most propagations were made by nogoods with a ranking of around 0.04, while in Figure 4.1 the peak was much closer to 0. This suggests that the nogoods with the strictly lowest LBD were not necessarily the most useful. However, this behaviour could also be an artefact of the artificial increase in ranking stemming from multiple nogoods having the same metric value (as described in Example 5).

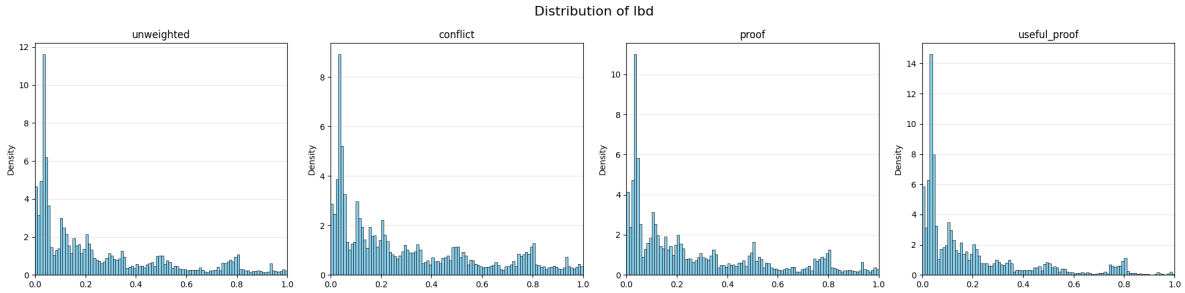

Figure 4.13: Density distribution of rankings for LBD, split into four approaches of weighing propagations.

Secondly, activity is better at identifying useful nogoods compared to the general optimisation instances. Its density distribution of rankings consistently has the lowest mean and the highest skewness among all metrics, as demonstrated in Table 4.3. Moreover, Figure 4.14 shows that activity does not have a significantly lower percentage of propagations made by nogoods with a ranking greater than 0.5 in the *unweighted*, *conflict*, and *proof* weighing approaches. This is in contrast with the results on all optimisation instances from Figure 4.6, where activity stood out for its poor performance.

Table 4.3: Summary statistics derived from the density distributions data. Bolded are the two best values in each category (lowest for Mean/Std, highest for Skewness).

Type	Metric	Mean	Std	Skewness
unweighted	size	0.2319	0.2753	1.3100
	activity	**0.2048**	0.3125	**1.4287**
	lbd	0.2532	**0.2615**	1.1468
	num_variables	0.2343	0.2725	1.3168
	decision_levels_span	0.2807	0.2820	0.9144
	search_space_size	**0.2230**	0.2627	**1.3546**
	constraints_count	0.2927	**0.2591**	0.9955
	constraints_count_recursive	0.2701	0.2707	1.1547
conflict	size	0.3008	0.3030	0.8903
	activity	**0.2154**	0.3376	**1.3016**
	lbd	0.3044	**0.2802**	0.8483
	num_variables	0.3025	0.3006	0.9023
	decision_levels_span	0.3226	0.2919	0.6930
	search_space_size	**0.2889**	0.2925	**0.9502**
	constraints_count	0.3397	**0.2741**	0.7446
	constraints_count_recursive	0.3331	0.2930	0.8179
proof	size	0.2719	0.2948	1.0617
	activity	**0.2115**	0.3306	**1.3095**
	lbd	0.2765	**0.2713**	0.9847
	num_variables	0.2731	0.2929	1.0799
	decision_levels_span	0.2947	0.2828	0.8096
	search_space_size	**0.2634**	0.2923	**1.1498**
	constraints_count	0.3262	**0.2692**	0.8209
	constraints_count_recursive	0.3129	0.2879	0.9389
useful_proof	size	0.1722	0.2224	1.7744
	activity	**0.0884**	**0.2170**	**2.9938**
	lbd	0.1979	0.2272	1.5522
	num_variables	0.1759	0.2223	**1.7793**
	decision_levels_span	0.2278	0.2563	1.2234
	search_space_size	**0.1657**	**0.2133**	1.7704
	constraints_count	0.2667	0.2377	1.1375
	constraints_count_recursive	0.2333	0.2386	1.4170

Similarly to activity, search space size also shows improvement, compared to the general optimisation instances. It is among the top three metrics in terms of means and skewness of the ranking distribution across all weighing approaches. While it is still closely correlated with size and number of variables, it slightly outperforms them, as presented in Figure 4.14.

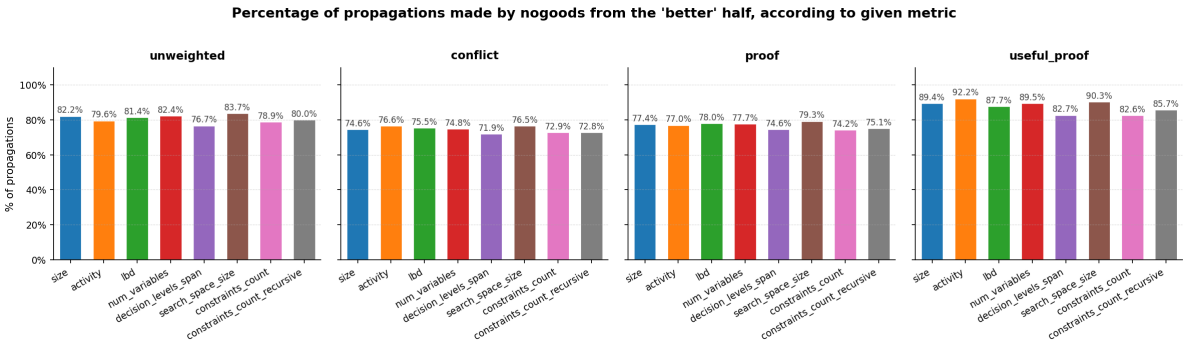

Figure 4.14: Percentages of propagations made by nogoods with a ranking of less than or equal to 0.5 (in case of activity and search_space_size – more than or equal to 0.5).

Lastly, correlations between the metrics, as shown in Figure 4.15, are largely the same as those observed across the wide range of optimisation problems. The same groups of highly correlated metrics can be identified. Interestingly, recursive constraints count is closely correlated with size, number of variables and search space size, even more tightly than with the plain version of constraints count, which is surprising given the similarities in the definitions of these two metrics.

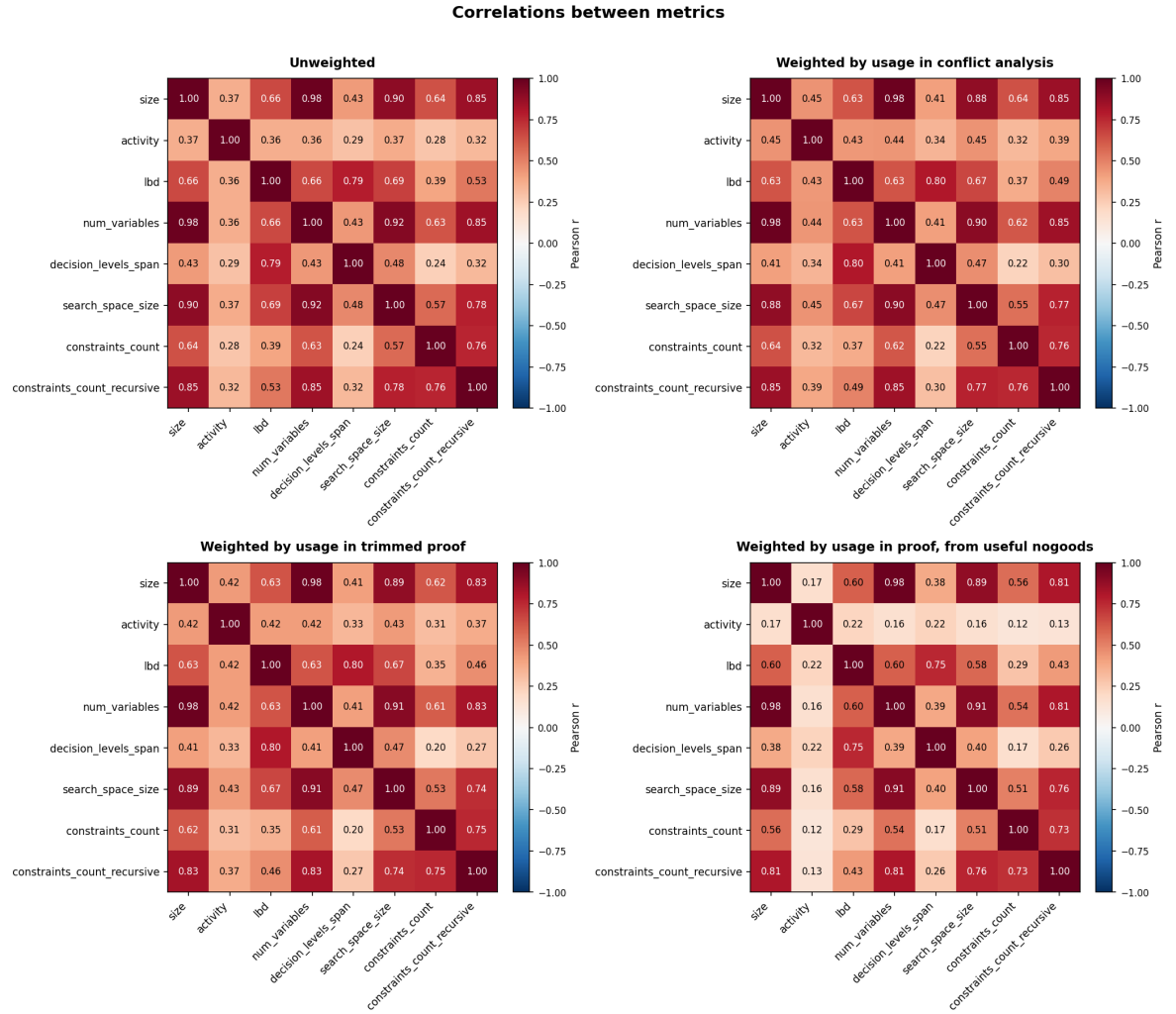

Figure 4.15: Correlations between the rankings assigned to nogoods by the metrics, split by approaches to weighing propagations.

4.7. Summary

In Section 4.4, Section 4.5 and Section 4.6, we have seen that all eight of our evaluated metrics can, to some extent, predict whether a given nogood is likely to cause propagations. Some of them have highly skewed distributions of rankings, such as LBD or activity, suggesting better performance, but other metrics, for example, search space size, could also be useful in specific cases. To verify the exact usefulness of the metrics, we used several of them to guide the nogood database reduction process and compared the performance of the resulting solver versions in Chapter 5.

However, before we proceed to this detailed evaluation, we can make use of one of the observations we made here. Namely, some of the metrics are highly correlated with each other, suggesting they yield similar usefulness estimates for nogoods and would perform the same actions when used in nogood management. To recall, the groups we identified were:

1. LBD, decision levels span
2. Activity
3. Size, number of variables, search space size
4. Constraints count, recursive constraints count

Since metrics in a group are likely to offer the same performance, we pick one metric per cluster:

1. **LBD** – when compared to decision levels span, LBD consistently had a lower mean and standard deviation, as well as a higher skewness of its ranking distributions.
2. **Activity** – it was the only metric in its cluster; it had relatively low correlation with all other metrics, especially when using the *useful proof* weighing approach for propagations.
3. **Number of variables** – it covers its cluster the best, that is, it had the highest correlation with the other two metrics (e.g. its correlation with size was higher than the correlation between size and search space size).
4. **Recursive constraints count** – the performance of the two types of constraints count was close, with the recursive version being slightly better. Moreover, the recursive one counts only the original constraints, so it does not depend on the solver's previous actions and better captures the structure of the problem.

We thus identified four nogood quality metrics that are likely to offer differing performance when used to remove nogoods. While they all can predict nogood usefulness on their own, we believe they provide different information about the nogoods, and it could be beneficial to combine metrics to judge nogood's usefulness - we evaluate this claim in Chapter 5.

5

Database reductions

In Chapter 4, we identified four nogood quality metrics for further analysis of their performance:

- Literal Block Distance
- Activity
- Number of variables
- Recursive constraints count (for simplicity, we will refer to it as just *constraints count*)

In this chapter, we propose several nogood removal schemes that make use of one or more of the above metrics. We experimentally evaluate these schemes across a wide range of optimisation and satisfaction problems, focusing on conflict counts and anytime performance. Additionally, we measure the impact of other parameters related to nogood management, such as the maximum database size and the rate of size increase. We also compare our best-performing schemes with the approach from Pumpkin, an open-source solver that won a bronze medal at MiniZinc Challenge 2025. Lastly, we attempt to explain the differences in performance between the solver versions in terms of the number of propagations that the retained nogoods perform. Our results show that:

1. Schemes retaining nogoods with extreme values for the metrics generally perform the best, with the one targeting low LBD or high activity requiring the fewest conflicts for both optimisation and satisfaction instances (Section 5.3 and Section 5.4).
2. Differences in numbers of conflicts between schemes are modest, around 5%, but increase to $> 20\%$ when compared to a specifically designed poor-performing nogood removal approach (Section 5.3 and Section 5.4).
3. Considering number of variables in addition to LBD and activity improves the solver's anytime behaviour, allowing it to find good solutions quickly (Section 5.5).
4. Limiting the maximal size of the nogood database can be advantageous in satisfaction instances. On the other hand, optimisation problems benefit from larger databases, but periodic nogood removals are still essential (Section 5.6).
5. The differences in performance among the database reduction schemes cannot be fully attributed to the solver's ability to remove the nogoods that cause few propagations (Section 5.7).

5.1. Nogood management schemes

Most solvers periodically remove nogoods by sorting them according to a specific heuristic and then removing the ones that are deemed worse. We adopt this idea and construct several schemes that attempt to remove half of the stored nogoods, using either only a single metric or multiple combined.

Single The first approach is to simply use a single metric to order the nogoods and remove half of them. However, there is a caveat – in our implementation, nogoods currently involved in propagation cannot be deleted, so that they can later provide propagation explanations if needed. Because of this

requirement, if such a nogood is to be removed according to the given metric, we skip it and remove other non-propagating nogood instead. The detailed algorithm is shown in Algorithm 2. It always attempts to remove half of the stored nogoods, but due to propagating nogoods, it could occur that fewer get deleted. For the same reason, it could happen that we do not strictly remove all the nogoods from the worse half – we might keep a propagating nogood but remove another one with a better value for the metric.

Algorithm 2 *Single* nogood removal scheme

Input: a list of all learned nogoods N and a nogood quality metric M

$N \leftarrow N.sortOn(M)$ ▷ Sorted descendingly on perceived usefulness
$numToRemove \leftarrow N.length/2$
$idx \leftarrow N.length - 1$
while $numToRemove > 0$ **do**
  while $idx \geq 0$ and $isPropagating(N[idx])$ **do**
    $idx \leftarrow idx - 1$
  end while
  if $idx \geq 0$ **then**
    $markForRemoval(N[idx])$
    $numToRemove \leftarrow numToRemove - 1$
    $idx \leftarrow idx - 1$
  else
    break ▷ All remaining nogoods are propagating
  end if
end while
$removeMarkedNogoods()$

Both The second approach is to combine two metrics and keep the nogoods that have good values for both of them. This is equivalent to removing nogoods that have a poor value for at least one of the metrics. The details of this approach are presented in Algorithm 3. As in the *single* approach, we aim to remove half of the nogoods and skip those currently propagating. Note that we attempt to remove the same number of nogoods for each metric, which could mean that more than half of the database is deleted. Additionally, while the execution of this algorithm can vary slightly depending on the order in which metrics are provided, we believe that, in practice, the potential differences are negligible, and we use a single ordering for each pair of metrics in our experiments.

Either The last approach also combines pairs of metrics, keeping the nogoods for which at least one metric has a good value. While the *both* scheme might favour nogoods that have good-but-not-the-best values, this approach clearly prefers the nogoods with extreme metrics. The detailed procedure is shown in Algorithm 4. Similarly to the previous ones, it skips the propagating nogoods and tries to keep the same number of nogoods for each metric. The order in which the metrics are provided can again slightly alter the output, but we believe the difference in this case is negligible as well.

Algorithm 3 *Both* nogood removal scheme

Input: a list of all learned nogoods N and two nogood quality metric M_1 and M_2

$N_1 \leftarrow N.sortOn(M_1)$ ▷ Sorted descendingly on perceived usefulness
$N_2 \leftarrow N.sortOn(M_2)$
$numToRemove \leftarrow N.length/2$
$idx_1 \leftarrow N.length - 1$
$idx_2 \leftarrow N.length - 1$
while $numToRemove > 0$ **do**
  while $idx_1 \geq 0$ and ($isPropagating(N_1[idx_1])$ or $isMarked(N_1[idx_1])$) **do**
    $idx_1 \leftarrow idx_1 - 1$
  end while
  if $idx_1 \geq 0$ **then**
    $markForRemoval(N_1[idx_1])$
    $numToRemove \leftarrow numToRemove - 1$
    $idx_1 \leftarrow idx_1 - 1$
  end if
  while $idx_2 \geq 0$ and ($isPropagating(N_2[idx_2])$ or $isMarked(N_2[idx_2])$) **do**
    $idx_2 \leftarrow idx_2 - 1$
  end while
  if $idx_2 \geq 0$ **then**
    $markForRemoval(N_2[idx_2])$
    $numToRemove \leftarrow numToRemove - 1$
    $idx_2 \leftarrow idx_2 - 1$
  end if
  if $idx_1 < 0$ and $idx_2 < 0$ **then**
    break ▷ All remaining nogoods are propagating
  end if
end while
$removeMarkedNogoods()$

Algorithm 4 *Either* nogood removal scheme

Input: a list of all learned nogoods N and two nogood quality metric M_1 and M_2

$N_1 \leftarrow N.sortOn(M_1)$ ▷ Sorted descendingly on perceived usefulness
$N_2 \leftarrow N.sortOn(M_2)$
$keep \leftarrow \{n \in N \mid isPropagating(n)\}$
$numToKeep \leftarrow N - (N.size/2) - keep.size$
$idx_1 \leftarrow 0$
$idx_2 \leftarrow 0$
$L \leftarrow N.size$
while $numToKeep > 0$ **do**
  while $idx_1 < L$ and $keep$ contains $N_1[idx_1]$) **do**
    $idx_1 \leftarrow idx_1 + 1$
  end while
  if $idx_1 < L$ **then**
    $keep \leftarrow keep \cup \{N_1[idx_1]$
    $numToKeep \leftarrow numToKeep - 1$
    $idx_1 \leftarrow idx_1 + 1$
  end if
  while $idx_2 < L$ and $keep$ contains $N_2[idx_2]$) **do**
    $idx_2 \leftarrow idx_2 + 1$
  end while
  if $idx_2 < L$ **then**
    $keep \leftarrow keep \cup \{N_2[idx_2]$
    $numToKeep \leftarrow numToKeep - 1$
    $idx_2 \leftarrow idx_2 + 1$
  end if
end while
$removeNogoods(\{n \in N \mid n \notin keep\})$

5.2. Experimental setup

Software We implemented the nogood management schemes in Pumpkin on commit `6020771`. The solver was built using Rust 1.94.0. All problem instances were decomposed to FlatZinc using MiniZinc 2.9.7 and Pumpkin's custom configuration rules.

Hardware The experiments were carried out on the DelftBlue supercomputer [1] using the *compute-p2* partition with Intel Xeon Gold 6448Y 32C 2.1 GHz processors. Each problem instance was run in a separate process with 4GB of memory and a timeout of 20 minutes.

Solver parameters An important aspect of nogood management is how often the database reduction is triggered. In most solvers, this is handled by setting a threshold on the number of stored nogoods, which, when exceeded, triggers the nogood removal process. Various solvers use different values for this threshold. For example, in Chuffed [7] it is fixed at 100000, in NACRE [19] it starts at 4000 and is increased by 500 after each reduction, Pumpkin [15] uses the *tiered* system with 100000, 7000 and 20000 as fixed thresholds for its low, medium and high tiers. In our implementation, we initially set this threshold to 40000 and increase it by 10% after each database reduction. These values were chosen as a compromise between those from other solvers, and we measure the impact of this choice in Section 5.6.

Database reduction schemes Using our four nogood quality metrics and the nogood management schemes described in Section 5.1, we build the following solver versions for evaluation:

- Using each of the metrics and the *single* scheme, denoted *single_{metric}*
- Using each of the six pairs of metrics and the *both* scheme, denoted *both_{metric}_{metric}*
- Using each of the six pairs of metrics and the *either* scheme, denoted *either_{metric}_{metric}*
- Adapted versions of the *both* and *either* schemes that use all four metrics, denoted *all* and *all_either* respectively. We also added a version of the *either* scheme that uses three of the metrics – LBD, activity, and number of variables – denoted *all_either_except_cc*. The reason for this particular version will be explained later.
- As a baseline, we used a *random* version that does not use any metric, but rather removes the nogoods randomly (it follows Algorithm 2, but instead of sorting, it shuffles the nogoods). We ran this solver version three times and averaged the collected statistics.
- Lastly, our preliminary experiments suggested that *either_lbd_activity* performs notably well, so we added a version *opposite* to it – it removes the nogoods for which either LBD or activity has a good value in a manner similar to Algorithm 3.

In this way, we obtained 21 different solver versions for evaluation.

The benchmarks We evaluate the different nogood removal approaches on a wide range of optimisation and satisfaction instances. For optimisation, we used all problem instances from the last 10 years of the MiniZinc Challenge (2016-2025), resulting in 862 instances across 118 problem classes. However, some of them failed during decomposition to FlatZinc, and we additionally removed models larger than 100 MB (to improve the performance of our experimental setup). This left us with 783 instances.

As there are fewer satisfaction instances available in the MiniZinc Challenge repository, we used all such instances from the last 15 years of the Challenge (2011-2025). There were 14 problem classes with 188 instances, 5 of which failed during decomposition to FlatZinc, resulting in 183 instances.

5.3. Optimisation

We begin our analysis by considering optimisation instances. We examine the number of solved instances for each solver, and analyse in detail the number of conflicts the versions required to reach optimal solutions, both in general and on a per-instance basis.

5.3.1. Solved instances and conflict counts

Table 5.1 shows the performance of each solver version. The solvers are grouped based on the reduction scheme they use – *single*, *both*, or *either* – as well as those that use three or four metrics, and two

baselines: *random* and *opposite*. Within each such group, the best values for each column are bolded (apart from the *Satisfiable* column, as here solvers' goal is to show optimality, not that a solution exists; and the *Unsatisfiable* column, as there are almost no differences between solvers in this matter).

Firstly, the table presents the number of instances solved to optimality, the number of problems for which a solution was found without proving the optimal bound, and the number for which the solver showed that no solution exists. The parenthesised numbers in the *Optimal* and *Unsatisfiable* columns count how many of those instances needed more than 40000 conflicts to be solved (thus triggering at least one database reduction). For the *Satisfiable* column, the parenthesised number indicates how many instances the given solver found a solution for, while at least one other solver did not find any (before the timeout). Column *Total* sums these counts, respectively parenthesised and non-parenthesised.

Apart from counting solved instances, we also measure the average number of conflicts and nodes (decisions) the solver needed to reach optimality. To be able to compare these metrics across solvers, we consider only the instances solved by all solver versions. Additionally, in cases when an instance required fewer than 40000 conflicts, there were no database reductions performed, meaning that all solvers behaved in the exact same manner. Because of that, we also exclude such instances and consider only the ones in which nogood removal occurred at least once. There were 245 instances solved to optimality by all solvers, 162 of which required fewer than 40000 conflicts, leaving 83 instances. The conflict and node counts for these instances are aggregated using the arithmetic mean and presented respectively in the *Avg Conflicts* and *Avg Nodes* columns.

Lastly, we consider the time it took the solvers to solve instances. We calculate the average PAR-2 score, a metric used in SAT Competitions [16], which assigns a solver a number of points equal to the solve time in seconds if the instance was solved, and to double the time limit otherwise (with a timeout of 1200 seconds in our case).

It should be noted that the runtime on DelftBlue is highly volatile – we observed that the differences in solve time when running a given solver version on a particular instance multiple times could be as high as 50%, depending on the load on the supercomputer. For the same reason, it could be that the solver does not manage to completely solve an instance or find a solution for it, simply because the hardware used was under more stress. Because of that, in our analysis, we focus on counts of conflicts and nodes for solved instances, as they objectively represent the amount of work the solver had to perform to solve a given instance. Nonetheless, we still provide counts of solved instances and the PAR-2 metrics for completeness.

Table 5.1: Solver comparison. Parenthesised counts for Optimal/Unsatisfiable: instances above $40\,000$ conflicts threshold; for Satisfiable: instances for which at least one other solver did not find any solution in the time limit. Avg Conflicts/Nodes computed over 83 instances solved to optimality by all solvers in more than $40\,000$ conflicts. PAR-2 uses a timeout of 1200 seconds.

Version	Optimal	Satisfiable	Unsatisfiable	Total	Avg Conflicts	Avg Nodes	PAR-2
single_lbd	265 (103)	382 (9)	5 (1)	652 (113)	**528475.89**	695626.63	1690.55
single_activity	**266** (104)	383 (11)	4 (0)	**653** (115)	530998.72	**676963.16**	**1688.22**
single_var	265 (103)	382 (9)	5 (1)	652 (113)	539106.33	697395.12	1689.51
single_cc	256 (94)	391 (9)	4 (0)	651 (103)	562411.29	738501.55	1710.84
both_lbd_activity	260 (98)	387 (9)	4 (0)	651 (107)	526644.84	697839.87	1709.00
both_lbd_var	**261** (99)	388 (11)	4 (0)	**653** (110)	**525316.81**	**688654.04**	**1702.92**
both_lbd_cc	255 (93)	391 (8)	5 (1)	651 (102)	539115.45	704841.20	1712.02
both_activity_var	260 (98)	388 (10)	4 (0)	652 (108)	542483.16	721702.08	1706.86
both_activity_cc	257 (95)	392 (11)	4 (0)	**653** (106)	549029.40	723815.05	1712.37
both_var_cc	257 (95)	388 (7)	4 (0)	649 (102)	574673.08	741481.06	1712.87
either_lbd_activity	**269** (107)	378 (9)	4 (0)	651 (116)	**506491.55**	**671042.00**	1685.86
either_lbd_var	264 (102)	383 (9)	4 (0)	651 (111)	536365.36	706320.61	1696.44
either_lbd_cc	268 (106)	377 (7)	4 (0)	649 (113)	533882.99	711321.25	1684.91
either_activity_var	**269** (107)	380 (11)	4 (0)	**653** (118)	527789.19	703810.93	**1680.85**
either_activity_cc	**269** (107)	378 (9)	4 (0)	651 (116)	531474.17	691612.42	1683.45
either_var_cc	261 (99)	386 (9)	4 (0)	651 (108)	532282.75	704195.76	1699.06
all	260 (98)	389 (11)	4 (0)	653 (109)	547040.14	719734.27	1703.24
all_either	265 (103)	382 (9)	4 (0)	651 (112)	**513054.95**	**672070.81**	1691.16
all_either_except_cc	**269** (107)	381 (12)	5 (1)	**655** (120)	521460.07	683191.07	**1676.72**
random	264 (102)	384 (10)	4 (0)	652 (112)	542824.82	728237.55	1693.90
opposite	253 (91)	390 (5)	4 (0)	647 (96)	642500.19	892312.00	1721.22

The first observation we can make based on Table 5.1 is that when using only a single metric, constraints count performs poorly compared to the other three metrics. LBD and activity are close to each other in terms of conflict count, with number of variables not much worse. This does not align with the observation made in [20] about SAT solving, which notes that activity performs much worse than LBD. However, the authors also observed that using LBD and size offers very similar performance – in Chapter 4 we saw that size and number of variables are closely correlated, and here we see that *single_var* results in only roughly 2% more conflicts compared to *single_lbd*, seemingly confirming that observation from previous research.

Combining metrics using *both* schemes decreases the number of instances solved to optimality. In terms of conflict count, most approaches result in more conflicts compared to using just the better one of the included metrics. In the two schemes, *both_lbd_activity* and *both_lbd_var*, for which the conflict count decreases, this difference is very small (less than 1%).

On the other hand, combining metrics using *either* schemes improves performance, increasing the number of solved instances, and decreases or keeps the conflict count roughly the same compared to single metrics. This improvement is most notable in *either_lbd_activity*, which, on average, has the fewest conflicts and nodes across all the tested solver versions.

Using all four metrics with *both* scheme decreases performance, solving fewer instances and requiring more conflicts than any of the *either* approaches or using the metrics on their own (apart from constraint count). In contrast, combining all four metrics with *either* scheme outperforms almost all solvers, only having more conflicts than *either_lbd_activity*.

These results suggest that nogoods with extreme metric values are especially useful. The *both* approach might remove a nogood if one of the metrics has a very good value, but the other has a poor one; thus, it is more likely to keep nogoods with mediocre metric values. On the other hand, the *either* approach will strictly keep the nogoods with the best metrics. The improved performance of the *either* schemes shows that, when combining metrics, their role is not necessarily to agree with each other, but rather to complement each other – when one metric misses a useful nogood, then the other one hopefully retains it.

As mentioned before, *either_lbd_activity* achieves the lowest average count of conflicts and nodes, and solves the most instances to optimality (joint with several other solvers). However, *all_either_except_cc* solves the same number of instances to optimality, and manages to find solutions for three more, as well as certify that there is no solution in one more problem, thus solving or finding a solution in the biggest number of instances among all solvers. It also achieves the lowest PAR-2 score of all solver versions. At the same time, we need to remember that the varying runtimes on DelftBlue can highly influence the results in these categories.

When looking at our two baselines, we see that *random* performs surprisingly well, improving over *single_cc*, *all*, *both_activity_cc* and *both_var_cc*. It also has a conflict count only about 8% higher than the best-performing *either_lbd_activity*. However, the other baseline – *opposite* – performs much worse, solving the fewest instances, requiring the most conflicts and decisions, as well as achieving the worst PAR-2 score, highlighting it as the clearly poorest-performing solver version.

Just comparing the arithmetic means of conflict counts in Table 5.1 can be misleading, as the resulting average can be dominated by a handful of instances with many conflicts. Therefore, we look at conflict counts in more detail in Table 5.2. There, we calculate, for each instance, the solver's performance relative to the average solver, the virtual best and worst, as well as the random baseline, and aggregate the results using the geometric mean. In the table, the best values in each column for every solver group are bolded.

Table 5.2: Conflict counts in detail over 83 instances (solved by all solvers with $>40\,000$ conflicts). 'vs. avg solver' compares each solver against the per-instance mean; 'vs. virtual best/worst' compares against the per-instance minimum/maximum, 'vs. random' compares against that solver's conflict count. Percentage differences are aggregated with the geometric mean.

Version	Avg Conflicts	vs. avg solver	vs. virtual best	vs. virtual worst	vs. random
single_lbd	**528475.89**	-2.4%	+11.8%	-25.5%	-5.4%
single_activity	530998.72	**-4.1%**	**+9.9%**	**-26.8%**	**-7.0%**
single_var	539106.33	-1.7%	+12.6%	-25.0%	-4.7%
single_cc	562411.29	+2.7%	+17.7%	-21.6%	-0.4%
both_lbd_activity	526644.84	-3.0%	+11.1%	-26.0%	-6.0%
both_lbd_var	**525316.81**	**-3.5%**	**+10.6%**	**-26.3%**	**-6.4%**
both_lbd_cc	539115.45	-0.9%	+13.6%	-24.3%	-3.9%
both_activity_var	542483.16	-2.9%	+11.2%	-25.9%	-5.9%
both_activity_cc	549029.40	+1.7%	+16.5%	-22.4%	-1.4%
both_var_cc	574673.08	+2.2%	+17.1%	-22.0%	-0.9%
either_lbd_activity	**506491.55**	**-6.8%**	**+6.8%**	**-28.8%**	**-9.6%**
either_lbd_var	536365.36	-4.0%	+9.9%	-26.8%	-7.0%
either_lbd_cc	533882.99	-2.7%	+11.4%	-25.8%	-5.7%
either_activity_var	527789.19	-4.6%	+9.3%	-27.2%	-7.5%
either_activity_cc	531474.17	-3.6%	+10.5%	-26.4%	-6.5%
either_var_cc	532282.75	-1.9%	+12.4%	-25.1%	-4.9%
all	547040.14	-1.5%	+12.9%	-24.8%	-4.5%
all_either	**513054.95**	-5.1%	+8.7%	-27.6%	-8.0%
all_either_except_cc	521460.07	**-5.5%**	**+8.3%**	**-27.8%**	**-8.3%**
random	542824.82	+3.1%	+18.2%	-21.3%	+0.0%
opposite	642500.19	+27.8%	+46.4%	-2.4%	+23.9%

In Table 5.2, we clearly see the reason for inclusion of *all_either_except_cc* – LBD, activity and number of variables, when used alone, all improve compared to the average solver, and significantly outperform the *random* solver, while constraints count is worse than the average and only slightly better than *random*. Therefore, this poor performance of constraints count could be the reason for *all_either* being worse than *either_lbd_activity*. To test this hypothesis, we included *all_either_except_cc*, which combines LBD, activity and number of variables using the *either* approach. It does not have a lower arithmetic mean of conflicts, but it does improve over *all_either* when using other aggregation types. However, it still performs worse than *either_lbd_activity* in every category.

The new types of aggregation mostly agree with the plain average of conflicts and confirm our observations based on Table 5.1, with two notable exceptions – *single_activity* outperforming *single_lbd* and the aforementioned *all_either_except_cc* outperforming *all_either*. Interestingly, all non-baseline solvers have fewer conflicts than the *random* solver when aggregated using geometric mean, even the ones with a greater arithmetic mean of conflicts.

either_lbd_activity is still clearly the best performing solver, having on average almost 10% fewer conflicts than the *random* baseline. Its contrary version, *opposite*, is clearly the worst, being close to the virtual worst solver, requiring on average 24% more conflicts than the random solver, and having around 27% more average conflicts than *either_lbd_activity*.

The fact that there is no single solver version that is significantly close to the virtual best, while the *opposite* is clearly the virtual worst in many instances, suggests that it is less important to choose the best nogood management scheme, but rather we need to avoid the worst approaches. Most of the solvers have similar conflict counts, within 5% of each other. However, choosing a bad approach – the *opposite* – can considerably decrease performance, increasing the conflict count by more than 20%, even when compared to an average-performing database reduction scheme.

5.3.2. Conflicts per instance

In Figure 5.1, we plot per instance the number of conflicts for each solver. For the majority of instances, all versions have nearly the same number of conflicts, with only the *opposite* and *random* having noticeably more. Only about a quarter of all problems show a significant difference. Interestingly, these outliers come from various problem classes, with different solvers performing best and worst in each of them. We were unable to pinpoint any specific reason for this discrepancy in performance.

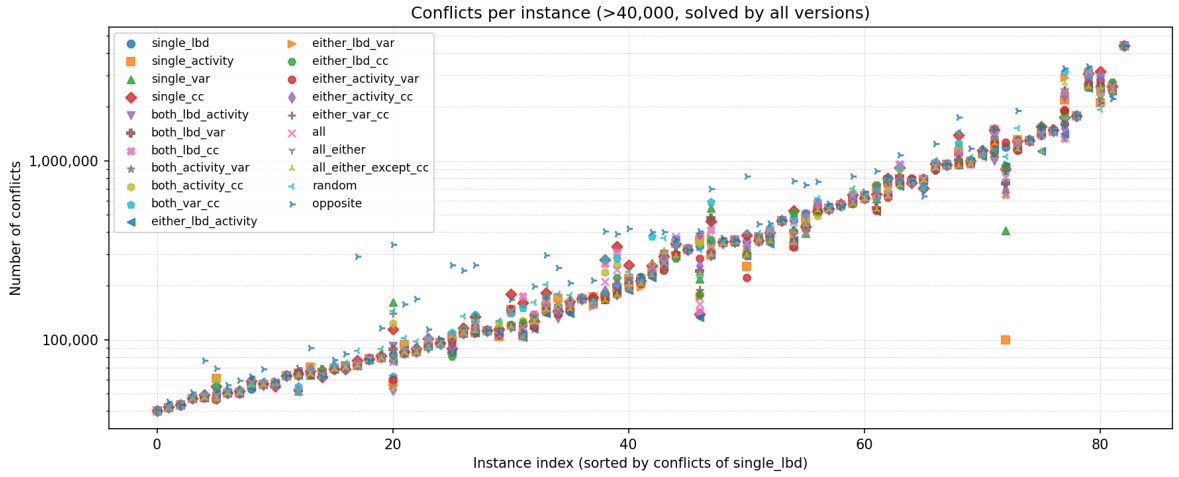

Figure 5.1: Number of conflicts per instance for all solvers, in instances solved to optimality by all solvers in at least $40\,000$ conflicts.

While the overall high number of conflicts of *opposite* stems from the fact that it is often the worst performing version for a given instance, the reason for an overall low conflict count for *either_lbd_activity* is not visible so clearly. It is not the best-performing approach in many instances, and there is also no particular instance in which it would significantly outperform other solvers (as is the case for *single_activity*). Instead, this version likely consistently achieves slightly lower numbers of conflicts, which, when aggregated, result in the best overall performance.

5.3.3. Best performing schemes

In Table 5.1 and Table 5.2, the conflict counts were calculated only on instances solved to optimality by all solvers, meaning that poor versions, like *single_cc* or *opposite*, considerably reduce the number of instances we consider and thus influence the results. Therefore, we repeat our analysis of the number of conflicts using only the solvers that solved the most instances – 269 – and present the results in

Table 5.3. We additionally included *all_either*, as it was the second-best performing solver in terms of conflict count. This way, we increased the number of considered instances (solved by all solvers using $>$ 40000 conflicts) from 83 to 98 (if we were to exclude *all_either*, the resulting count would be 100).

Table 5.3: Conflict counts in detail over 98 instances (solved by all solvers with $>40\,000$ conflicts). 'vs. avg solver' compares each solver against the per-instance mean; 'vs. virtual best/worst' compares against the per-instance minimum/maximum. Percentage differences are aggregated with the geometric mean.

Version	Avg Conflicts	vs. avg solver	vs. virtual best	vs. virtual worst
either_lbd_activity	**661807.14**	**-3.4%**	**+2.8%**	**-10.6%**
either_activity_var	713257.63	+0.5%	+7.0%	-6.9%
either_activity_cc	754766.04	+3.6%	+10.3%	-4.0%
all_either	**665953.24**	**-1.9%**	**+4.4%**	**-9.1%**
all_either_except_cc	693737.41	-1.2%	+5.2%	-8.5%

With the two best values in each column of Table 5.3 bolded, we can see that *either_lbd_activity* still performs the best, achieving the fewest conflicts. *all_either* is slightly worse, though the relative difference in the average number of conflicts between these two solvers became smaller. Additionally, this time *all_either* outperforms *all_either_except_cc* in all aggregation types. Overall, the relative performance of these five solvers remained roughly the same, showing that inclusion of more instances does not significantly alter the results.

5.4. Satisfaction

Our analysis of satisfaction instances follows the same pattern as the optimisation ones, where we first examine the number of solved instances and compare the number of conflicts, nodes, and solve time. The results are presented in Table 5.4. Out of the initial 183 instances, 91 were found to be satisfiable by all solvers, 66 of which required fewer than 40000 conflicts, leaving 25 instances for computing the average numbers of conflicts and nodes. There were also 8 unsatisfiable instances that all solvers solved. However, they required fewer than 40000 conflicts, so these instances were excluded from the computation of conflicts and nodes.

Table 5.4: Solver comparison. Parenthesised counts: instances above $40\,000$ conflicts threshold. Avg Conflicts/Nodes were computed over 25 instances solved by all solvers in more than $40\,000$ conflicts (all instances were satisfiable). PAR-2 uses a timeout of 1200 seconds.

Version	Satisfiable	Unsatisfiable	Total Solved	Avg Conflicts	Avg Nodes	PAR-2
single_lbd	**95** (29)	8 (0)	**103** (29)	**376719.52**	**408690.04**	**1140.79**
single_activity	**95** (29)	8 (0)	**103** (29)	384380.12	415872.16	1152.67
single_var	93 (27)	8 (0)	101 (27)	395937.28	430127.04	1173.61
single_cc	92 (26)	8 (0)	100 (26)	397238.80	427678.12	1190.41
both_lbd_activity	94 (28)	8 (0)	102 (28)	**379256.44**	**411970.84**	1155.47
both_lbd_var	94 (28)	8 (0)	102 (28)	381618.28	412508.84	1168.80
both_lbd_cc	91 (25)	8 (0)	99 (25)	387634.96	419871.96	1183.01
both_activity_var	93 (27)	8 (0)	101 (27)	390571.16	422433.56	1163.65
both_activity_cc	**95** (29)	8 (0)	**103** (29)	395146.48	429341.76	**1151.73**
both_var_cc	92 (26)	8 (0)	100 (26)	388522.24	419701.88	1183.64
either_lbd_activity	93 (27)	8 (0)	101 (27)	**368848.44**	**397730.08**	1177.35
either_lbd_var	**94** (28)	8 (0)	**102** (28)	379939.76	412304.84	**1170.11**
either_lbd_cc	92 (26)	8 (0)	100 (26)	371021.96	400900.64	1179.81
either_activity_var	92 (26)	8 (0)	100 (26)	377887.88	406700.88	1184.69
either_activity_cc	93 (27)	8 (0)	101 (27)	380297.92	408113.60	1176.71
either_var_cc	93 (27)	8 (0)	101 (27)	383497.52	413151.48	1170.36
all	**94** (28)	8 (0)	**102** (28)	393065.36	426237.20	1153.23
all_either	93 (27)	8 (0)	101 (27)	**371269.40**	**400228.08**	1176.98
all_either_except_cc	**94** (28)	8 (0)	**102** (28)	375767.44	404082.16	**1148.11**
random	93 (27)	8 (0)	101 (27)	397268.80	432774.84	1171.04
opposite	92 (26)	8 (0)	100 (26)	469817.92	519213.64	1182.97

The results are overall similar to the ones for optimisation. LBD and activity outperform number of variables and constraint count when used on their own. Solvers using *both* schemes in almost all cases result in worse performance compared to using only the better of their metrics, while solvers using *either* approaches almost always improve over both metrics. Combining all four metrics does not improve performance over *either_lbd_activity*, which is again the solver with the fewest conflicts and nodes.

Random does not perform that well anymore, with all versions achieving lower average conflict and node counts. However, *opposite* is still clearly the worst solver, requiring around 25% more conflicts than *either_lbd_activity*. Relatively, the differences between solver versions are smaller than those in optimisation, with conflict counts differing by around 2-6% (whereas in optimisation they were around 4-10%). This suggests that nogoods are less important when solving satisfaction instances for which a solution exists.

We compare conflict counts in more detail in Table 5.5. Unlike in optimisation, all aggregation types agree with each other, showing that *single_lbd* improves over *single_activity*, and that *all_either* is better than *all_either_except_cc*. Notably, in the *both* category, *both_lbd_activity* performs the best, compared to *both_lbd_var* in optimisation. However, the differences between these two schemes are rather small.

Table 5.5: Conflict counts in detail over 25 instances (solved by all solvers with $>40\,000$ conflicts). 'vs. avg solver' compares each solver against the per-instance mean; 'vs. virtual best/worst' compares against the per-instance minimum/maximum; 'vs. random' compares against that solver's conflict count. Percentage differences are aggregated with the geometric mean.

Version	Avg Conflicts	vs. avg solver	vs. virtual best	vs. virtual worst	vs. random
single_lbd	**376719.52**	**-2.8%**	**+7.2%**	**-23.0%**	**-6.8%**
single_activity	384380.12	-2.5%	+7.6%	-22.7%	-6.5%
single_var	395937.28	+1.0%	+11.5%	-20.0%	-3.1%
single_cc	397238.80	+0.9%	+11.3%	-20.1%	-3.2%
both_lbd_activity	**379256.44**	**-3.3%**	**+6.6%**	**-23.4%**	**-7.3%**
both_lbd_var	381618.28	-2.7%	+7.4%	-22.9%	-6.7%
both_lbd_cc	387634.96	-1.7%	+8.4%	-22.1%	-5.7%
both_activity_var	390571.16	-0.4%	+9.9%	-21.1%	-4.4%
both_activity_cc	395146.48	-0.1%	+10.2%	-20.8%	-4.1%
both_var_cc	388522.24	-0.9%	+9.3%	-21.5%	-5.0%
either_lbd_activity	**368848.44**	**-5.3%**	**+4.5%**	**-24.9%**	**-9.1%**
either_lbd_var	379939.76	-2.0%	+8.1%	-22.4%	-6.0%
either_lbd_cc	371021.96	-4.5%	+5.4%	-24.3%	-8.3%
either_activity_var	377887.88	-2.3%	+7.8%	-22.6%	-6.3%
either_activity_cc	380297.92	-1.7%	+8.5%	-22.1%	-5.7%
either_var_cc	383497.52	-2.9%	+7.1%	-23.1%	-6.9%
all	393065.36	-0.7%	+9.6%	-21.3%	-4.7%
all_either	**371269.40**	**-4.1%**	**+5.8%**	**-24.0%**	**-8.0%**
all_either_except_cc	375767.44	-2.9%	+7.1%	-23.1%	-6.9%
random	397268.80	+4.2%	+15.0%	-17.4%	+0.0%
opposite	469817.92	+22.8%	+35.5%	-2.7%	+17.8%

For completeness, we also include the plot with the number of conflicts per instance for all solvers in Figure 5.2. As in optimisation, in most instances the differences between solver versions are negligible, with only *opposite* or *random* performing worse, though surprisingly, there is also one instance in which *opposite* has the fewest conflicts. The best-performing *either_lbd_activity* again does not dominate in any instances, but rather consistently performs well, improving by small amounts.

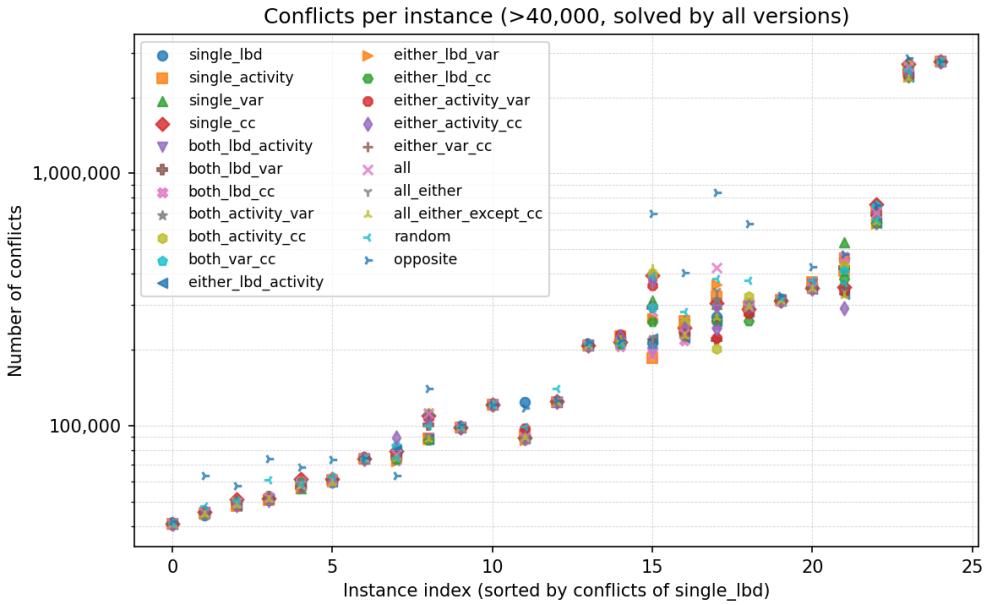

Figure 5.2: Number of conflicts per instance, in instances solved by all solvers in at least $40\,000$ conflicts.

5.5. Primal integral

In Section 5.3, we focused on comparing performance in cases where the instances get solved to optimality. However, there is another approach to evaluating solvers – *anytime performance*. In many real-world applications, we do not necessarily need to find the optimal solution, but rather we are interested in obtaining a *good*, close-to-optimal solution quickly. *Primal integral* (PI) [6] is a metric that estimates exactly this behaviour. It uses a notion of *primal gap* – a value in $[0, 1]$ that estimates, for a given time point, how far the solver's last found solution is from the best solution found by any solver over the whole search process. The primal integral is computed as the integral of the primal gap function over the time interval from 0 to the timeout T.

Note that the primal integral depends on the *time* when solutions were found, meaning that the variable runtime on DelftBlue can influence these results. The hope is that our results remain reliable thanks to aggregating numerous instances solved on different nodes and spread over time, thereby hopefully capturing the underlying properties of the solver versions.

Since in satisfaction instances we only have a single solution, and for primal integral we need to have several ones that continuously improve the objective value over time, in Table 5.6 we consider only the 83 optimisation instances that were solved to optimality by all solvers (in at least 40000 conflicts). The results for *random* are not averaged over several runs (as was the case in the preceding analysis of conflicts count), but we arbitrarily picked one of the runs to compute this solver's primal integral.

Table 5.6: Primal integral statistics over the 83 instances solved to optimality (timeout 1200 s). Columns: average primal integral; geometric-mean % difference vs per-instance mean, virtual best, virtual worst, and the random baseline.

Solver	Avg PI	vs. avg solver	vs. virtual best	vs. virtual worst	vs. random
single_lbd	32.4517	-0.0%	+52.8%	**-29.8%**	+4.0%
single_activity	**27.3106**	**-14.2%**	**+31.2%**	-39.7%	**-10.7%**
single_var	30.7657	-8.1%	+40.4%	-35.5%	-4.4%
single_cc	36.1693	-0.6%	+51.9%	-30.2%	+3.4%
both_lbd_activity	38.1628	+12.0%	+71.2%	-21.3%	+16.6%
both_lbd_var	34.4546	**+5.0%**	**+60.6%**	**-26.2%**	**+9.3%**
both_lbd_cc	33.6561	+6.3%	+62.5%	-25.3%	+10.6%
both_activity_var	37.5691	+8.4%	+65.7%	-23.8%	+12.8%
both_activity_cc	35.7569	+8.0%	+65.1%	-24.1%	+12.4%
both_var_cc	**33.2793**	+7.1%	+63.7%	-24.8%	+11.4%
either_lbd_activity	36.5148	+4.7%	+60.0%	-26.5%	+8.9%
either_lbd_var	34.1827	+7.0%	+63.5%	-24.9%	+11.3%
either_lbd_cc	32.5326	-10.4%	+36.9%	-37.1%	-6.8%
either_activity_var	**26.0610**	**-13.8%**	**+31.7%**	**-39.5%**	**-10.3%**
either_activity_cc	27.3055	-13.2%	+32.7%	-39.0%	-9.7%
either_var_cc	27.8122	-10.5%	+36.8%	-37.1%	-6.9%
all	33.4441	-1.1%	+51.1%	-30.5%	+2.9%
all_either	29.3246	-4.8%	+45.5%	-33.1%	-0.9%
all_either_except_cc	**27.4353**	**-22.8%**	**+18.0%**	**-45.8%**	**-19.7%**
random	31.3002	-3.9%	+46.9%	-32.5%	+0.0%
opposite	36.7889	+2.7%	+56.9%	-27.9%	+6.8%

The results here differ quite significantly compared to our analysis of the conflict count. The best performing solver in terms of the arithmetic mean of primal integrals is *either_activity_var*. However, in all other aggregation types, it is considerably surpassed by *all_either_except_cc*. The solver, which had the fewest conflicts, *either_lbd_activity*, has one of the highest primal integrals, and performs poorly compared to the average solver and the *random* baseline. At the same time, the solver which makes completely different decisions – the *opposite* – does not perform much better (although it is also not the worst one, as was the case in terms of conflicts).

The relative differences between different solvers are bigger than when looking at conflict counts, and the performances are more varied. At the same time, no single version is close to the virtual best or

worst, suggesting that performance depends heavily on the specific problem instance. This can also be seen in Figure 5.3, which plots the average primal gap over time. The primal gap curves of all solvers are close to each other, with no single one clearly finding better solutions earlier.

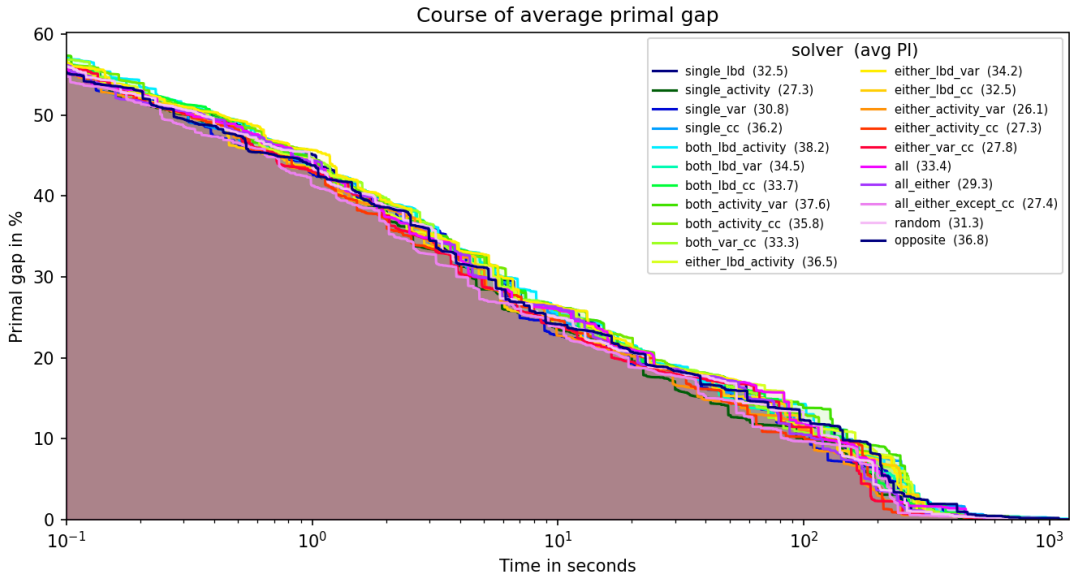

Figure 5.3: Course of the average primal gap on instances solved to optimality, x-axis is in log-scale.

Figure 5.4 presents the primal integrals per instance. The differences between solvers are more explicit than when we compared the conflict count. In all instances, the values are spread, with the best solver often having a primal integral several times smaller than that of the worst version. As we observed earlier, there is no solver close to the virtual best or worst, with the solver performance changing on an instance-by-instance basis.

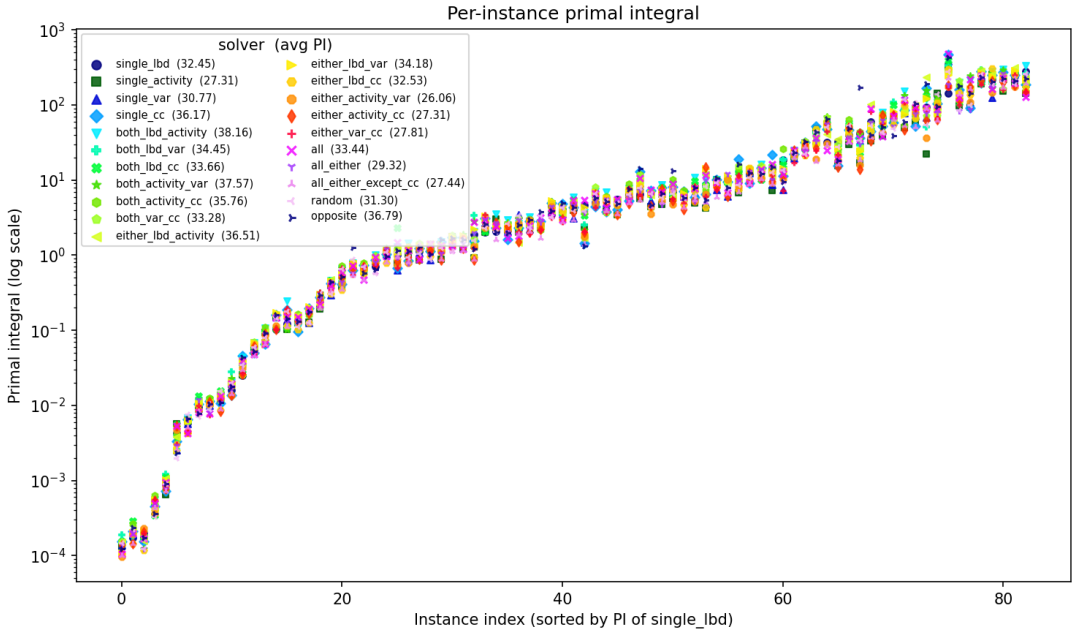

Figure 5.4: Primal integrals per instance, in instances solved to optimality by all solvers in at least $40\,000$ conflicts.

Since the primal integral is used to estimate the quality of solutions found before the solver reaches optimality, we can also add to our analysis all the instances that were not necessarily fully solved, but for which all solver versions found at least one solution. This results in a considerable increase to 435 instances, and we present the comparison of their primal integrals in Table 5.7.

Table 5.7: Primal integral statistics over 435 instances in which all solvers found at least one solution (timeout 1200 s). Columns: average primal integral; geometric-mean % difference vs per-instance mean, virtual best, and virtual worst, and the random baseline.

Solver	Avg PI	vs. avg solver	vs. virtual best	vs. virtual worst	vs. random
single_lbd	63.7122	-4.6%	+60.2%	-36.6%	+0.9%
single_activity	60.8969	-10.5%	+50.4%	-40.5%	-5.3%
single_var	**54.7590**	**-14.1%**	**+44.3%**	**-42.9%**	**-9.2%**
single_cc	68.9519	+1.2%	+70.1%	-32.7%	+7.1%
both_lbd_activity	72.5377	+12.9%	+89.7%	-25.0%	+19.4%
both_lbd_var	66.7660	+2.8%	+72.6%	-31.7%	+8.7%
both_lbd_cc	70.0688	+5.6%	+77.5%	-29.8%	+11.7%
both_activity_var	65.0098	+3.5%	+73.8%	-31.2%	+9.4%
both_activity_cc	**63.8336**	**+1.6%**	**+70.7%**	**-32.5%**	**+7.5%**
both_var_cc	67.7927	+6.9%	+79.5%	-29.0%	+13.0%
either_lbd_activity	65.3369	+0.4%	+68.7%	-33.2%	+6.2%
either_lbd_var	64.6726	+2.4%	+72.1%	-31.9%	+8.3%
either_lbd_cc	59.7073	-10.9%	+49.8%	-40.8%	-5.7%
either_activity_var	**54.0980**	**-17.4%**	**+38.8%**	**-45.1%**	**-12.6%**
either_activity_cc	56.8467	-15.7%	+41.7%	-43.9%	-10.8%
either_var_cc	60.3419	-6.8%	+56.6%	-38.0%	-1.4%
all	65.1491	-2.5%	+63.9%	-35.2%	+3.2%
all_either	62.1448	-3.6%	+61.9%	-36.0%	+1.9%
all_either_except_cc	**52.8385**	**-21.4%**	**+32.0%**	**-47.8%**	**-16.9%**
random	60.8957	-5.5%	+58.8%	-37.2%	+0.0%
opposite	69.5687	+4.2%	+75.0%	-30.8%	+10.2%

The results slightly change compared to Table 5.6. When using only one metric, *single_var* is the best. *both_activity_cc* now outperforms other solvers using *both* approach. *all_either_except_cc* has now a lower arithmetic mean of the primal integral than *either_activity_var*, thus becoming the best solver version according to all aggregation types. We can see that the three best performing approaches – *all_either_except_cc*, *either_activity_var* and *single_var* – all utilise the number of variables. While using this metric did not significantly reduce the number of conflicts among solved instances, these results suggest that it is vital to the solver's anytime performance.

In Figure 5.5 we plot the course of the average primal gap. The differences between solvers become even less noticeable than in Figure 5.3, with their performance varying over time. Unlike in the previous figure, the primal gap does not drop to 0 around the timeout, indicating that there are considerable differences in the objective values of solutions found by solvers within the time limit.

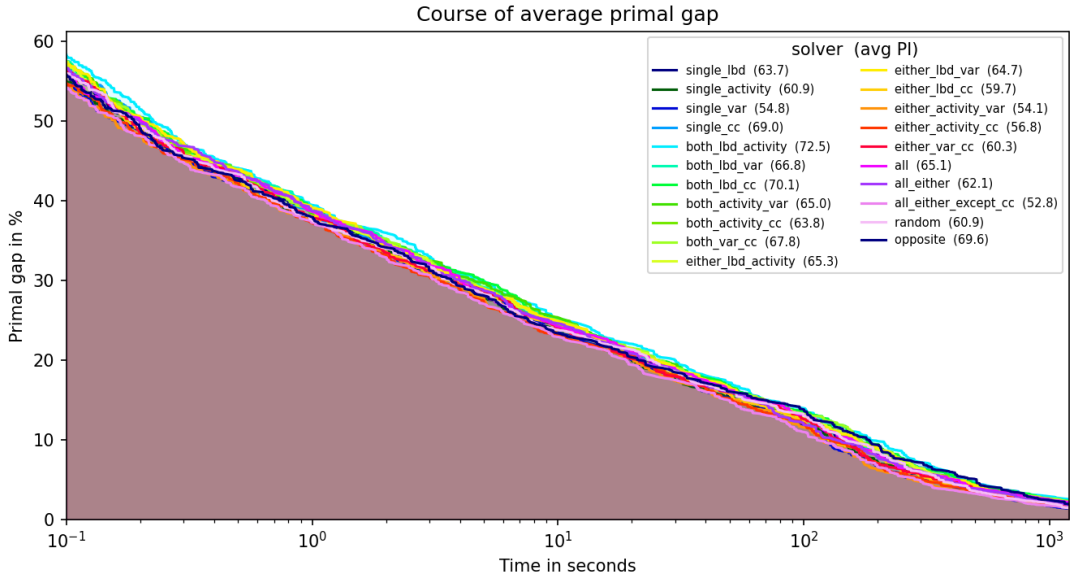

Figure 5.5: Course of the average primal gap in instances in which all solvers found a solution, x-axis is in log-scale.

Figure 5.6 presents the primal integrals for all instances. We can see that as the primal integrals grow, the divergence between solvers grows as well. This means that if a solver finds a good solution quickly, the others are likely to do so too, but when it takes more time to reach optimality, the solvers vary in performance. Additionally, for some of the newly included instances, there are substantial differences between the solvers, even up to two orders of magnitude. However, we should keep in mind that observations could be a result of the variable runtime on DelftBlue, with some versions finding a good solution more quickly than others simply because the hardware was under less stress.

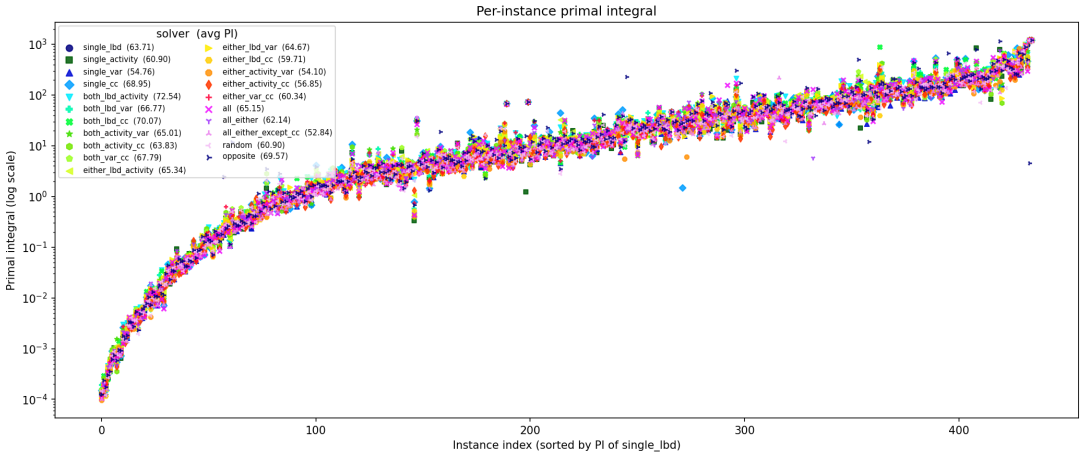

Figure 5.6: Primal integrals per instance, in instances in which all solvers found a solution, and at least one did so in more than $40\,000$ conflicts.

5.6. Impact of database size

As mentioned in Section 5.2, in the solver used to run the experiments in the previous sections, we made specific choices regarding the size of the nogood database and how it grows. Nogood removal occurred when the database size exceeded a specific threshold, initially set to 40000. After each database clean-up, the threshold was increased by 10%, with no upper limit on the maximum database size. Throughout the experiments, we noticed that using this scheme when solving some of the bigger

instances meant that the database grew up to 700000 nogoods, likely causing the solver to spend a lot of resources on storing and handling the nogoods.

In this section, we measure the impact of database size on solver performance. Specifically, we alter the parameters in three ways. Firstly, reducing the initial threshold for the first nogood removal to 10000 – the database is initially smaller, influencing the information the solver learns at the beginning of the solving process. Secondly, reducing the threshold increase factor to 2%, which causes the database to grow much more slowly, keeping the burden on the solver low for a longer period. Thirdly, introducing an upper limit on the threshold of 100000, thus preventing the database from ever growing too large and hindering the solver's performance.

We take the original solver, as well as mix and match these three changes to create 5 new solver versions – three by applying one of the changes, a fourth one by applying all changes to obtain a solver that aggressively reduces the database, and a fifth one by applying the last two changes, but keeping the initial threshold at 40000. All these versions used the best-performing *either_lbd_activity* approach for ranking nogoods. The solvers are named as *{initial threshold}_{threshold increase}_{upper limit}*.

Additionally, we included two more solvers as baselines:

1. *no_reductions* – A solver that does not perform any nogood removal, saving all the learned nogoods and letting the database grow freely.
2. *default_pumpkin* – A solver using the *tiered* system for nogood management used in the default implementation of Pumpkin. Developed based on [24, 31], it contains three tiers for nogoods: *low* of size 100000 for nogoods with LBD ≤ 3, *medium* of size 7000 for nogoods with LBD $\in [4, 6]$, and *high* of size 20000 for the remaining nogoods. Whenever the number of nogoods in one of the tiers exceeds the limit, they are sorted (by activity in the *high* and *medium* tiers; LBD, with size to break ties, is used in the *low* tier), and half of them get removed.

In Table 5.8 we present the performance of the solvers on optimisation instances. There were 243 instances solved to optimality by all solvers. Since almost all solvers (apart from *default_pumpkin*) definitely do not perform any database reductions within the first 10000 conflicts, when computing the average number of conflicts and nodes, we only consider 123 instances solved with more than 10000 conflicts.

When looking at the results, we can no longer use the conflict count as an unbiased metric of solver performance. The solvers reduce the database at different rates and have different allowed maximum sizes at a given time point. If a solver removes nogoods more aggressively, it loses information provided by the deleted learned nogoods, which may require it to re-learn them, thus inflating the conflict count. At the same time, it maintains a smaller database, spending less resources on nogood management, meaning it might be able to process more conflicts in a given time unit, and even solve more instances. DelftBlue's However, we also cannot compare solvers solely by the number of solved instances due to the unstable runtime. Therefore, we need to look at both these metrics simultaneously. If a solver can solve more instances with fewer conflicts, we can be fairly certain that it indeed performs better. However, if these metrics do not agree, we need to be cautious and reason on a case-by-case basis.

Table 5.8: Solver comparison on optimisation instances. Parenthesised counts for Optimal/Unsatisfiable: instances above $10\,000$ conflicts threshold; for Satisfiable: instances for which at least one other solver did not find any solution in the time limit. Avg Conflicts/Nodes computed over 123 instances solved to optimality by all solvers in more than $10\,000$ conflicts.

Version	Optimal	Satisfiable	Unsatisfiable	Total	Avg Conflicts	Avg Nodes	PAR-2
10k_10p_noLimit	**270** (150)	380 (9)	4 (1)	654 (160)	289983.37	457544.63	**1675.78**
10k_2p_100k	260 (140)	392 (11)	4 (1)	**656** (152)	316133.82	499724.03	1701.22
40k_10p_100k	263 (143)	381 (3)	4 (1)	648 (147)	293585.78	468865.72	1694.23
40k_10p_noLimit	**269** (149)	378 (6)	4 (1)	651 (156)	**279940.26**	**445342.42**	1685.86
40k_2p_100k	264 (144)	387 (10)	4 (1)	**655** (155)	287902.33	455497.17	1692.04
40k_2p_noLimit	267 (147)	382 (8)	4 (1)	653 (156)	287479.31	454875.25	**1682.13**
default_pumpkin	262 (142)	385 (5)	4 (1)	651 (148)	321592.56	496240.02	1697.69
no_reductions	249 (129)	396 (4)	5 (2)	650 (135)	**278765.93**	**444056.80**	1727.29

The first observation we can make based on Table 5.8 is that *no_reductions* managed to solve significantly fewer instances than other solvers. It does not remove any nogoods, so it does not have to re-learn any of them, meaning its conflict count is low (which matches observations from [20] for SAT solving), but its fast-growing database causes performance to deteriorate quickly as the search continues. This result confirms the importance of periodic nogood removals.

At the same time, the best three solvers by both the conflict count and the number of solved instances are the *_noLimit* ones. This suggests that optimisation instances do benefit from a larger database, or that the applied upper limit should be greater.

We also evaluate the solvers on satisfaction instances, with the results presented in Table 5.9. The average numbers of conflicts and nodes were calculated using 41 satisfiable instances solved by all solvers in more than 10000 conflicts.

Table 5.9: Solver comparison on satisfaction instances. Parenthesised counts: instances above $10\,000$ conflicts threshold. Avg Conflicts/Nodes computed over 41 satisfiable instances solved by all solvers in more than $10\,000$ conflicts.

Version	Satisfiable	Unsatisfiable	Total Solved	Avg Conflicts	Avg Nodes	PAR-2
10k_10p_noLimit	93 (46)	8 (3)	101 (49)	179545.27	200241.80	1182.54
10k_2p_100k	**94** (47)	8 (3)	**102** (50)	196773.85	228978.24	**1156.69**
40k_10p_100k	**94** (47)	8 (3)	**102** (50)	173663.51	191746.56	1158.79
40k_10p_noLimit	93 (46)	8 (3)	101 (49)	**173464.83**	**191476.39**	1177.35
40k_2p_100k	91 (44)	8 (3)	99 (47)	177918.63	196808.98	1188.83
40k_2p_noLimit	**95** (48)	8 (3)	**103** (51)	177918.63	196808.98	**1147.33**
default_pumpkin	91 (45)	8 (3)	99 (48)	183980.05	205977.20	1190.39
no_reductions	92 (45)	8 (3)	100 (48)	**170789.83**	**186841.44**	1198.22

Firstly, the *no_reductions* solver is not the worst one anymore, managing to solve more instances than *default_pumpkin* and *40k_2p_100k*. This contrasts with observations made in SAT solving that keeping more clauses is especially harmful in satisfiable instances [20]. The effect of the upper limit of database size is also not so clear, with *40k_10p_100k* outperforming the corresponding version without the limit, *40k_10p_noLimit*, but at the same time, a solver without the limit, *40k_2p_noLimit*, managed to solve the most instances.

Comparing the other parameter changes, we also cannot make any unambiguous conclusions. Decreasing the initial database clean-up threshold to 10000 conflicts allowed the *10k_10p_noLimit* version to solve the most optimisation instances. However, the same change between *40k_2p_100k* and *10k_2p_100k* resulted in a decrease in performance on optimisation problems but an increase on satisfaction ones. Similarly, reducing the rate of increase of the database size threshold from 10% to 2% resulted in fewer instances solved to optimality, but more instances were found to be satisfiable in both optimisation and satisfaction problems. This suggests that slowly growing databases help find solutions quickly, but struggle to prove optimality.

Looking at the solver version that removes nogoods the most aggressively, *10k_2p_100k*, we see it manages to fully solve very few optimisation instances, but it finds a solution in the largest number of instances among all solvers. Moreover, it performs well on satisfaction problems, solving nearly the largest number of instances. Its aggressive database reductions lead to the highest conflict count among all solvers, but the number of solved instances suggests that maintaining a small database was beneficial for finding solutions quickly.

Lastly, we can see that almost all solvers improve over *default_pumpkin*, both in the number of solved instances and in the number of conflicts. The *tiered* system of nogood management used by this solver has shown considerable performance improvements [31] in SAT solving, but our results suggest that it does not offer any benefits over *either_lbd_activity* in the context of Constraint Programming. However, there could be a simple explanation for this behaviour – the LBD thresholds and tier size limits used in Pumpkin were not specifically optimised. It could be that after further parameter tuning, performance would increase significantly.

5.7. Nogoods performing few propagations

In Chapter 4, we defined the usefulness of a nogood in terms of the number of propagations it performs. At the same time, in [4], the authors observed that many learned nogoods propagate only once, meaning that they could be deleted immediately after they stop being propagating for the first time, without any negative impact on the solver's performance. In that view, one could argue that the main reason for differences in performance between our proposed nogood management schemes is their ability to identify and remove these non-propagating nogoods.

With this hypothesis in mind, some of our previous results could be surprising. For example, activity should be a good identifier for these unhelpful nogoods – if a nogood does not make any propagations, then its activity should be generally low. However, in Table 5.2, we saw that while using only activity does outperform some of the other schemes, performance further improves when it is additionally paired with LBD or the number of variables. Similarly, if many nogoods are valueless, then limiting the maximum nogood database size should be beneficial – the opposite of what we saw in Table 5.8.

In this section, we investigate whether there is a correlation between a nogood removal scheme's performance and its ability to discard non-propagating nogoods while retaining the ones that have caused the most propagations so far. In Figure 5.1, we saw that only around a quarter of the solved instances had significant differences between the solvers, so we selected 20 of them with the biggest relative differences in the number of conflicts between the best and worst solver. We then used those instances on the solver versions from Section 5.3, as well as *40k_10p_100k* to examine why limiting the database reduces performance, and *10k_2p_100k* to assess the characteristics of an aggressive removal scheme. When running each instance, after each database reduction, we measure several metrics related to the retained and deleted nogoods, and aggregate the results across reductions and instances.

Firstly, we verify whether a significant portion of all learned nogoods indeed do not make more than one propagation. The results we obtained showed that for all solver versions, 53-56% of learned nogoods propagated only once, showing that a large part of the nogoods do not contribute to solving beyond the initial propagation upon backtracking. However, there was no clear connection between the solver's performance and this percentage metric.

Next, we measure metrics related to nogoods that cause few propagations. To that end, we define **SP** – *single propagation* – the nogoods that made only one propagation since their creation up to their deletion, or the end of solving; and **NNP** – *no new propagations* – the nogoods that have not made any propagations since the last database reduction. Note that a given nogood can belong to both of these groups at the same time, if it makes only one propagation and survives at least one database reduction. Using these definitions, at each reduction we measure *%-SP-Left* and *%-NNP-Left*, the percentage of the database that the SP and NNP nogoods constitute after the nogood removal. Similarly, we calculate *%-SP-Rem* and *%-NNP-Rem*, measuring how much of all SP and NNP nogoods get removed in each database reduction. The average values of these metrics for all solver versions are presented in Table 5.10.

The three of our nogood quality metrics, apart from activity, when used on their own, can identify SP nogoods well, removing more than 60% of them in each reduction, but are poor at distinguishing the NNP nogoods, deleting less than 20% of them. On the other hand, activity does not remove so many SP nogoods, with them taking up more than 50% of the database. It is much better at finding NNP nogoods that have not propagated for some time, meaning their activity is likely very low.

When combining multiple metrics, their ability to identify SP and NNP nogoods also gets combined. For example, *either_lbd_activity* retains more SP nogoods than just using LBD, but it keeps fewer NNP nogoods thanks to activity. In all the schemes, the SP and NNP nogoods occupy roughly 30-50% of the database. Similarly, all schemes manage to remove 50-65% of all SP nogoods in each reduction. The percentage of NNP nogoods removed is either below 20% or above 40%, depending on whether the scheme uses activity as one of its metrics.

Table 5.10: Average metrics related to retention and deletion of nogoods that cause few propagations. '%-SP-Left' and '%-NNP-Left' are the fractions of all stored nogoods after the database reduction that belong to the SP and NNP groups. '%-SP-Rem' and '%-NNP-Rem' are the average fractions of the corresponding nogood groups that get removed in each database reduction.

Solver	%-SP-Left	%-NNP-Left	%-SP-Rem	%-NNP-Rem
single_lbd	36.63%	51.43%	64.30%	18.01%
single_activity	50.58%	33.42%	52.53%	50.89%
single_var	35.31%	51.35%	63.85%	17.95%
single_cc	37.19%	55.37%	64.08%	11.29%
both_lbd_activity	44.46%	39.66%	57.46%	38.23%
both_lbd_var	34.63%	50.04%	65.25%	18.78%
both_lbd_cc	32.40%	49.96%	67.74%	16.46%
both_activity_var	45.08%	31.91%	56.32%	46.95%
both_activity_cc	43.22%	32.97%	57.97%	44.83%
both_var_cc	33.57%	50.88%	66.30%	15.75%
either_lbd_activity	40.34%	32.04%	60.57%	47.84%
either_lbd_var	35.66%	50.93%	64.39%	17.61%
either_lbd_cc	34.78%	51.90%	64.90%	15.34%
either_activity_var	39.68%	31.91%	59.65%	48.25%
either_activity_cc	41.76%	34.60%	59.28%	41.68%
either_var_cc	35.87%	52.17%	63.81%	16.34%
all	38.34%	35.76%	62.49%	40.21%
all_either	37.20%	40.92%	62.70%	31.46%
all_either_except_cc	37.22%	36.35%	62.08%	36.30%
random	52.73%	32.35%	50.06%	47.54%
opposite	63.28%	65.23%	43.07%	14.86%
40k_10p_100k	39.55%	33.18%	60.90%	48.29%
10k_2p_100k	33.58%	33.52%	64.73%	48.28%

However, there is no clear characteristic that would allow us to identify the schemes that solved most instances with the fewest conflicts. The well-performing schemes we saw before, *either_lbd_activity* and *all_either_except_cc*, both have rather high *%-SP-Rem* and *%-NNP-Rem*, but the same holds for the *all* scheme, which performed poorly. We can only clearly distinguish *random*, as it removes around 50% of both SP and NNP nogoods, and the poor-performing *opposite*, with it managing to remove the fewest of the less-useful nogoods, having both groups constitute more than 60% of the database after the reductions.

Similarly, there is no single feature that would stand out for *40k_10p_100k* and *10k_2p_100k*, with both having statistics similar to *either_lbd_activity*. Therefore, the fact that they keep the same ratio of less-useful nogoods, together with their limit on the total database size, could explain their decreased performance on optimisation instances, as the same ratio with a smaller database means that the solver stores fewer useful nogoods.

Apart from removing the nogoods that cause few propagations, the task of nogood management schemes is also to preserve those that propagate frequently. Therefore, we also measure the schemes' ability to do so. As the number of propagations can differ significantly between the nogoods, we record the total number of propagations made by each nogood, both since its creation and since the last reduction, and sum them across all nogoods in the database before and after the reduction. We then calculate the ratio of these sums, allowing us to see how well the schemes preserve the active nogoods (*%-Tot-Prop* and *%-New-Prop* for propagations since nogood creation and the last database reduction, respectively). We also measure the average number of propagations made by each nogood, and the results are presented in Table 5.11.

Firstly, we can observe that for all solvers the nogoods made on average around 60-70 propagations. This value is 4 times higher than the one reported in [4], but this difference may stem from our measurement being based on a very limited number of instances. Generally, we could expect the better-

performing schemes to have more propagations per nogood, as they create fewer nogoods overall, thereby increasing this ratio. However, we can see in Table 5.11 that there are no clear differences between the different types of removal schemes in this metric. Potentially, this is because even though the poor-performing approaches require more nogoods, they also take more time, allowing more propagations to be made, again increasing the ratio of propagations to nogoods.

Table 5.11: Average metrics related to the solver's ability to preserve nogoods that cause many propagations. 'Propagations per nogood' is the average number of propagations triggered per each nogood. '%-Tot-Prop' and '%-New-Prop' are the average fractions of propagations of nogoods that are retained after the reduction compared to all the nogoods before the reduction, counting respectively propagations made since nogood creation and since the last database reduction.

Solver	Propagations per nogood	%-Tot-Prop	%-New-Prop
single_lbd	64.6620	91.17%	81.54%
single_activity	59.9711	79.01%	78.00%
single_var	68.2898	89.38%	79.80%
single_cc	66.3544	89.68%	78.40%
both_lbd_activity	63.0563	82.17%	80.89%
both_lbd_var	64.8904	90.24%	81.70%
both_lbd_cc	69.5245	91.63%	83.03%
both_activity_var	64.6403	81.08%	80.56%
both_activity_cc	65.3346	82.20%	82.05%
both_var_cc	68.5810	89.72%	80.09%
either_lbd_activity	62.9352	90.26%	85.94%
either_lbd_var	66.4995	91.31%	82.79%
either_lbd_cc	65.8041	90.69%	81.86%
either_activity_var	65.7137	90.09%	85.83%
either_activity_cc	64.6392	88.64%	84.70%
either_var_cc	67.9662	90.75%	82.08%
all	65.7611	85.82%	83.67%
all_either	64.4693	90.87%	84.71%
all_either_except_cc	65.1813	90.90%	85.41%
random	61.3409	53.36%	52.68%
opposite	59.1393	24.38%	13.60%
40k_10p_100k	62.7879	90.66%	85.81%
10k_2p_100k	58.5375	92.63%	84.79%

In terms of *%-Tot-Prop* and *%-New-Prop*, all schemes, apart from *random* and *opposite*, manage to retain nogoods that made 80-90% of the propagations. The better performing *either* schemes tend to generally obtain higher values than the *both* schemes, but this observation does not generalise well, as *single_activity* has visibly lower values than other schemes, despite its good performance in terms of solved instances and low conflict count. As previously, the only significant difference can be observed for *random* and *opposite*. While for *opposite* this does correlate with its poor performance, recall that *random* outperformed many of the *both* schemes.

Lastly, we investigated the average number of reductions that nogoods survive (not presented in the tables). For nogoods that remained in the database, this average *age* was several times higher in all schemes than for nogoods that were removed, showing that generally the nogoods that are saved get retained in several next reductions as well. For schemes that involve activity, this difference was smaller than in other approaches, suggesting it often happens that nogoods are kept in the database but do not cause propagations afterwards, leading to a drop in their activity and a quick subsequent deletion. Therefore, activity is much more volatile compared to other nogood quality metrics.

To sum up, we did not find any single quality that would correlate well with and explain the performance differences we observed in Section 5.3. An obvious issue with the evaluation we conducted is that we looked only at past propagations, whereas the goal of nogood management schemes is to identify nogoods likely to be useful in the future. Perhaps there are other metrics and properties of the retained nogoods that could explain the differences in performance between our nogood management schemes.

6

Conclusions and Future Work

The introduction of conflict analysis significantly improved solver performance [28]. However, the resulting list of learned clauses (or nogoods) can significantly slow the solver down if not properly managed. To that end, most solvers employ nogood management, which involves periodically removing some of the nogoods according to a specific heuristic. Numerous such heuristics have been proposed in the domain of SAT solving and later adopted in Constraint Programming, often without explicit performance analysis. In this thesis, we first analysed 8 different metrics for estimating the quality of nogoods, later combined them in various ways to create a series of nogood management approaches, and finally measured their performance across a wide range of optimisation and satisfaction problems.

Experimental results showed that all metrics can, to some extent, predict whether a nogood is likely to cause useful propagations in the future. Some of the metrics are highly correlated to each other, with their performance being very similar in optimisation and satisfaction instances. Within each group of correlated metrics, we identified one that best captures the performance of the entire cluster and used these identified metrics as heuristics guiding the nogood management process.

Our experiments, based on 966 instances, showed that solvers that prioritise nogoods with extreme metric values perform the best, with a solver that focuses on LBD and activity having the fewest conflicts in both optimisation and satisfaction problems, even outperforming the approach used in a state-of-the-art solver Pumpkin [15]. At the same time, the differences between most schemes were rather small (around to 5%), with the results heavily depending on the specific problem instance. However, when compared with a particularly designed poor-performing approach, the efficiency differences became significant. This shows that it might not be that important to find the best nogood removal scheme, but rather we need to avoid especially bad ones. When looking at the *anytime behaviour*, a combination of LBD, activity, and number of variables enabled the rapid discovery of good solutions, indicating that different sets of metrics should be used, depending on whether we are interested only in the strictly best solutions, or if we want to obtain *acceptable* ones quickly.

The analysis of the impact of the nogood database size on performance demonstrated that keeping the database small enabled solvers to find solutions for more instances, but then they struggled to solve the problems to optimality. When the database was allowed to grow larger, the solvers fully solved more optimisation instances, but they still required periodic nogood removals – without them, performance quickly dropped on more complicated instances, confirming observations from the literature [20].

Lastly, we attempted to explain the differences in performance between solvers by their ability to remove nogoods that make few propagations. However, we were not able to clearly identify any feature that would point out these differences, neither in the rate of removal of the less-useful nogoods, nor in the rate and time of retention of the highly active ones. Future work is needed to potentially pinpoint these differences.

6.1. Future Work

There are numerous opportunities for further research, extending all parts of this thesis.

Firstly, more nogood quality metrics and heuristics can be considered. They can be both numerical, like the ones we used, which allow us to sort the nogoods, but other types can also be studied – for example, research in SAT solving [20] suggested that including a binary *used* flag is beneficial. It indicates whether a given clause participated in conflict analysis since the last database reduction, and the experiments have shown it was highly useful, consistently improving solvers' performance.

The various metrics can then be combined in numerous ways. We only explored *both* and *either* approaches, which essentially work like logical AND and OR operators. More sophisticated functions could be considered, such as introducing additional metrics to break ties during nogood sorting. Moreover, while our data collection approach did not allow us to do so, machine learning techniques could be applied to train a lightweight model deciding whether a nogood should be kept at various time points. Such an approach would also allow us to vary the fraction of nogoods deleted in each reduction, rather than always keeping it fixed at 50%. This method showed promising results when applied in SAT solving [34].

Constraint Programming solvers have many interacting components, so it would be interesting to see how an appropriate change to one of them would affect the performance of nogood management schemes. For example, in our analysis, we always used the search strategy provided by the problem instances. If we were to use the activity-based VSIDS branching heuristic instead, the nogood removal approaches involving *activity* might offer better performance.

Another example of a related component could involve the process of making nogood propagations. Recently, Marijnissen et al. [27] developed a framework that generalises CDCL for CP, allowing for stronger and earlier nogood propagations in cases when all the unassigned atomic constraints in a given nogood correspond to a single integer variable. Using this approach could mean that nogoods with fewer variables are more important, as it is easier for them to become propagating. Therefore, our database reduction schemes that involve the *number of variables* metric could potentially become more useful.

References

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

[2] Tobias Achterberg. “Constraint Integer Programming”. Doctoral Thesis. Institut für Mathematik, Fakultät II Mathematik und Naturwissenschaften, July 17, 2007. doi: 10.14279/depositonce-1634. url: https://depositonce.tu-berlin.de/items/9f46a10e-2f7b-4dea-8e27-9cae07de5258.

[3] Gilles Audemard and Laurent Simon. “Predicting Learnt Clauses Quality in Modern SAT Solvers”. In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence*. Vol. 9. IJCAI’09. Pasadena, California, USA: Morgan Kaufmann Publishers Inc., July 2009, pp. 399–404. url: https://inria.hal.science/inria-00433805.

[4] Robbin Baauw, Maarten Flippo, and Emir Demirović. “Conflict Analysis Based on Cutting-Planes for Constraint Programming”. In: *31st International Conference on Principles and Practice of Constraint Programming (CP 2025)*. Ed. by Maria Garcia de la Banda. Vol. 340. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 4:1–4:19. isbn: 978-3-95977-380-5. doi: 10.4230/LIPIcs.CP.2025.4. url: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2025.4.

[5] Nicholas Beaumont. “Scheduling staff using mixed integer programming”. In: *European journal of operational research* 98.3 (1997), pp. 473–484. issn: 0377-2217. doi: https://doi.org/10.1016/S0377-2217(97)00055-6. url: https://www.sciencedirect.com/science/article/pii/S0377221797000556.

[6] Timo Berthold. “Measuring the impact of primal heuristics”. en. In: *Operations Research Letters* 41.6 (Nov. 2013), pp. 611–614. issn: 01676377. doi: 10.1016/j.orl.2013.08.007. url: https://linkinghub.elsevier.com/retrieve/pii/S0167637713001181.

[7] Geoffrey Chu et al. *Chuffed: The Chuffed CP Solver*. 2018. url: https://github.com/chuffed/chuffed.

[8] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. isbn: 9781450374644. doi: 10.1145/800157.805047. url: https://doi.org/10.1145/800157.805047.

[9] Martin Davis, George Logemann, and Donald Loveland. “A machine program for theorem-proving”. In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. issn: 0001-0782. doi: 10.1145/368273.368557. url: https://doi.org/10.1145/368273.368557.

[10] Sven De Vries and Rakesh V Vohra. “Combinatorial auctions: A survey”. In: *INFORMS Journal on computing* 15.3 (Aug. 2003), pp. 284–309. doi: 10.1287/ijoc.15.3.284.16077.

[11] Niklas Eén and Niklas Sörensson. “An Extensible SAT-solver”. en. In: *Theory and Applications of Satisfiability Testing*. Ed. by Gerhard Goos et al. Vol. 2919. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518. isbn: 978-3-540-20851-8 978-3-540-24605-3. doi: 10.1007/978-3-540-24605-3_37. url: http://link.springer.com/10.1007/978-3-540-24605-3_37 (visited on 01/18/2026).

[12] Salah E Elmaghraby. “Resource allocation via dynamic programming in activity networks”. In: *European Journal of Operational Research* 64.2 (1993), pp. 199–215. issn: 0377-2217. doi: https://doi.org/10.1016/0377-2217(93)90177-O. url: https://www.sciencedirect.com/science/article/pii/037722179390177O.

[13] Thibaut Feydy and Peter J. Stuckey. “Interval Constraints with Learning: Application to Air Traffic Control”. In: *Principles and Practice of Constraint Programming*. Ed. by Michel Rueher. Cham: Springer International Publishing, 2016, pp. 224–232. isbn: 978-3-319-44953-1.

[14] Thibaut Feydy and Peter J. Stuckey. “Lazy Clause Generation Reengineered”. en. In: *Principles and Practice of Constraint Programming - CP 2009*. Ed. by Ian P. Gent. Vol. 5732. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 352–366. isbn: 978-3-642-04243-0. doi: 10.1007/978-3-642-04244-7_29. url: http://link.springer.com/10.1007/978-3-642-04244-7_29.

[15] Maarten Flippo et al. “A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers”. In: *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*. Ed. by Paul Shaw. Vol. 307. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 11:1–11:20. isbn: 978-3-95977-336-2. doi: 10.4230/LIPIcs.CP.2024.11. url: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2024.11.

[16] Nils Froleyks et al. “SAT Competition 2020”. In: *Artificial Intelligence* 301 (2021), p. 103572. issn: 0004-3702. doi: https://doi.org/10.1016/j.artint.2021.103572. url: https://www.sciencedirect.com/science/article/pii/S0004370221001235.

[17] Graeme Gange, Geoffrey Chu, and Peter J. Stuckey. “Certifying Optimality in Constraint Programming”. Manuscript. 2023. url: https://people.eng.unimelb.edu.au/pstuckey/papers/certified-cp.pdf.

[18] Martin Gebser et al. “clasp: A Conflict-Driven Answer Set Solver”. en. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by Chitta Baral, Gerhard Brewka, and John Schlipf. Berlin, Heidelberg: Springer, 2007, pp. 260–265. isbn: 978-3-540-72200-7. doi: 10.1007/978-3-540-72200-7_23.

[19] Gael Glorian, Jean-Marie Lagniez, and Christophe Lecoutre. “NACRE-A nogood and clause reasoning engine”. In: *23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR’23)*. Ed. by Elvira Albert and Laura Kovacs. Vol. 73. EPiC Series in Computing. 2020, pp. 249–259. doi: 10.29007/dxnb.

[20] Bernhard Gstrein et al. “Learn to Unlearn”. In: *28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025)*. Ed. by Jeremias Berg and Jakob Nordström. Vol. 341. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 14:1–14:12. isbn: 978-3-95977-381-2. doi: 10.4230/LIPIcs.SAT.2025.14. url: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2025.14.

[21] Fang He and Rong Qu. “A constraint programming based column generation approach to nurse rostering problems”. In: *Computers & Operations Research* 39.12 (2012), pp. 3331–3343. issn: 0305-0548. doi: https://doi.org/10.1016/j.cor.2012.04.018. url: https://www.sciencedirect.com/science/article/pii/S0305054812000986.

[22] Dongwon Kang, Jinhwan Jung, and Doo-Hwan Bae. “Constraint-based human resource allocation in software projects”. In: *Software: Practice and Experience* 41.5 (2011), pp. 551–577. doi: https://doi.org/10.1002/spe.1030. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.1030. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.1030.

[23] George Katsirelos and Fahiem Bacchus. “Generalized nogoods in CSPs”. In: *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*. Vol. 5. AAAI’05. Pittsburgh, Pennsylvania: AAAI Press, 2005, pp. 390–396. isbn: 157735236x.

[24] Stepan Kochemazov. “Improving Implementation of SAT Competitions 2017–2019 Winners”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Ed. by Luca Pulina and Martina Seidl. Springer. Springer International Publishing, 2020, pp. 139–148. isbn: 978-3-030-51825-7.

[25] Christophe Lecoutre et al. “Nogood Recording from Restarts”. In: *IJCAI*. Vol. 7. 2007, pp. 131–136.

[26] Chu Min Li, Fan Xiao, and Ruchu Xu. “On Reducing Clause DataBase in Glucose”. In: *IWIL-2015. 11th International Workshop on the Implementation of Logics*. Ed. by Boris Konev, Stephan Schulz, and Laurent Simon. Vol. 40. EPiC Series in Computing. EasyChair, 2016, pp. 67–77. doi: 10.29007/b8kb. url: /publications/paper/zpPk.

[27] Imko Marijnissen, Maarten Flippo, and Emir Demirović. “From Literals to Atomic Constraints: Generalising Conflict-Driven Clause Learning for Constraint Programming”. In: *Proceedings of the 32nd International Conference on Principles and Practice of Constraint Programming (CP 2026)*. To appear. 2026.

[28] João P. 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. issn: 0018-9340. doi: 10.1109/12.769433.

[29] Neil CA Moore. “C-learning: Further generalised g-nogood learning”. In: *ERCIM Workshop on Constraint Solving and Constraint Logic Programming, 2011*. 2011, p. 103.

[30] Matthew W. Moskewicz et al. “Chaff: engineering an efficient SAT solver”. In: *Proceedings of the 38th Annual Design Automation Conference*. DAC ’01. Las Vegas, Nevada, USA: Association for Computing Machinery, 2001, pp. 530–535. isbn: 1581132972. doi: 10.1145/378239.379017. url: https://doi.org/10.1145/378239.379017.

[31] Chanseok Oh. “Improving SAT solvers by exploiting empirical characteristics of CDCL”. PhD thesis. New York University, 2016.

[32] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. “Propagation = Lazy Clause Generation”. en. In: *Principles and Practice of Constraint Programming – CP 2007*. Ed. by Christian Bessière. Berlin, Heidelberg: Springer, 2007, pp. 544–558. isbn: 978-3-540-74970-7. doi: 10.1007/978-3-540-74970-7_39.

[33] Laurent Perron and Frédéric Didier. *CP-SAT*. Version v9.12. Google, Feb. 17, 2025. url: https://developers.google.com/optimization/cp/cp_solver/.

[34] Mate Soos, Raghav Kulkarni, and Kuldeep S Meel. “CrystalBall: Gazing in the Black Box of SAT Solving”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2019, pp. 371–387. isbn: 978-3-030-24258-9. doi: 10.1007/978-3-030-24258-9_26.

[35] Peter J Stuckey, Ralph Becket, and Julien Fischer. “Philosophy of the MiniZinc challenge”. In: *Constraints* 15.3 (July 2010), pp. 307–316. issn: 1383-7133. doi: 10.1007/s10601-010-9093-0. url: https://doi.org/10.1007/s10601-010-9093-0.

[36] Peter J. Stuckey et al. “The MiniZinc Challenge 2008–2013”. In: *AI Magazine* 35.2 (2014), pp. 55–60. doi: https://doi.org/10.1609/aimag.v35i2.2539. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1609/aimag.v35i2.2539. url: https://onlinelibrary.wiley.com/doi/abs/10.1609/aimag.v35i2.2539.

A

Nogood metrics plots

A.1. Optimisation

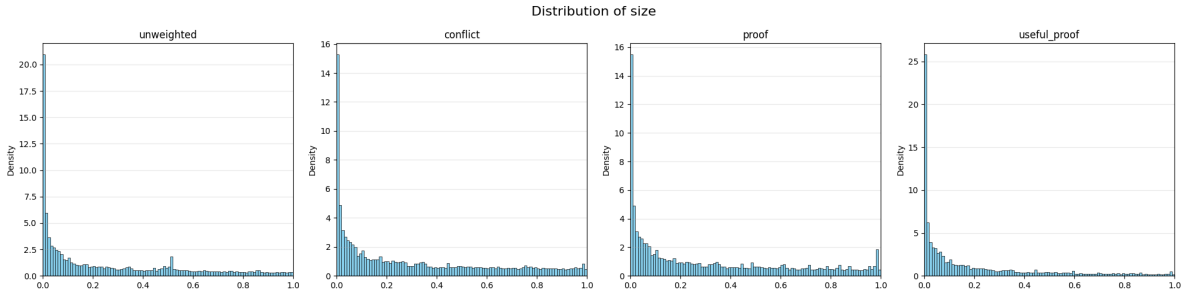

Figure A.1: Density distribution of rankings of *size* in optimisation instances.

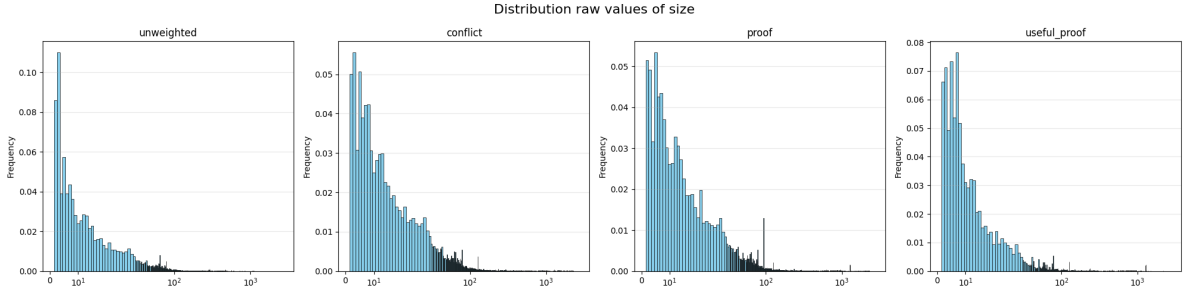

Figure A.2: Density distribution of raw values for *size*. X-axis is in *symlog* scale, with the linear threshold at 30.

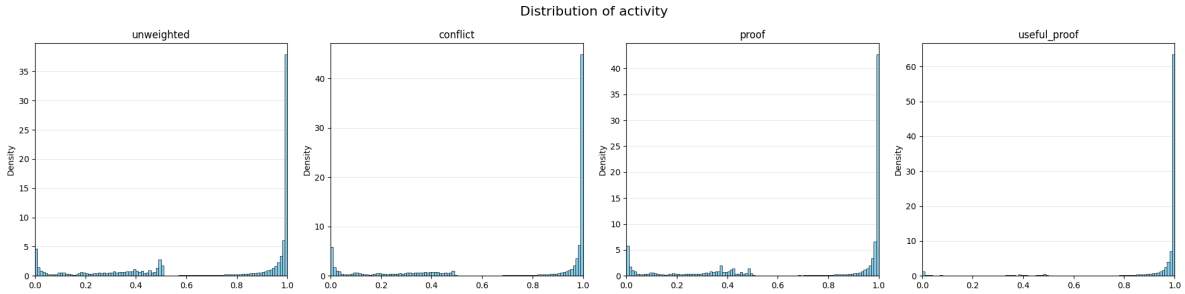

Figure A.3: Density distribution of rankings of *activity* in optimisation instances.

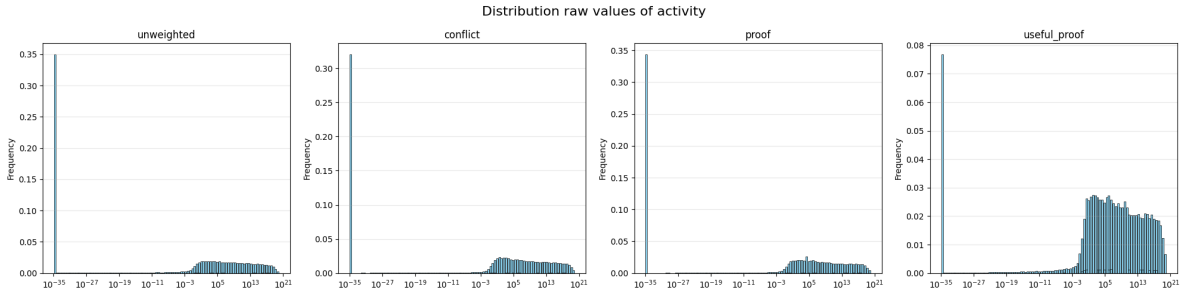

Figure A.4: Density distribution of raw values for *activity*. X-axis is in log scale, with values clamped at 10^{-35}.

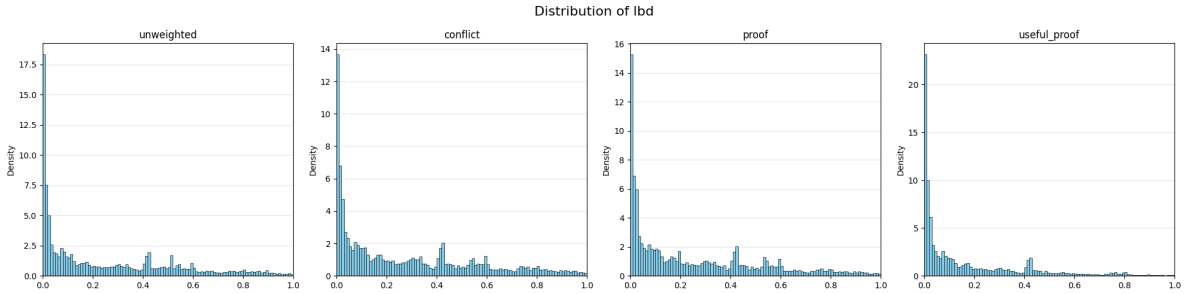

Figure A.5: Density distribution of rankings of *LBD* in optimisation instances.

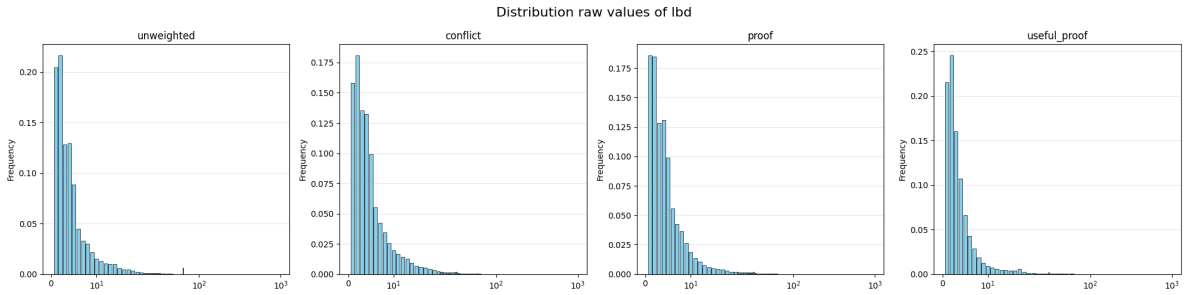

Figure A.6: Density distribution of raw values for *LBD*. X-axis is in *symlog* scale, with the linear threshold at 20.

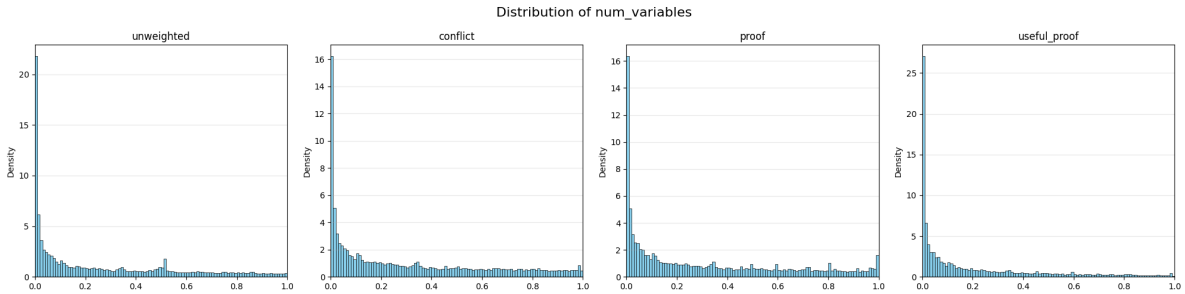

Figure A.7: Density distribution of rankings of *number of variables* in optimisation instances.

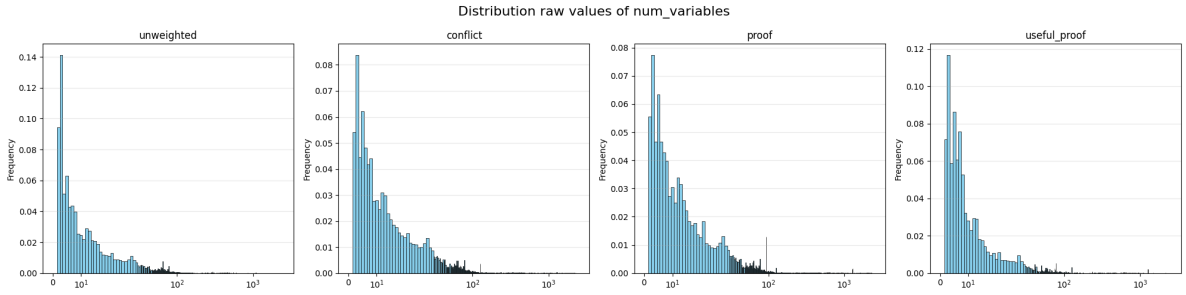

Figure A.8: Density distribution of raw values for *number of variables*. X-axis is in *symlog* scale, with the linear threshold at 30.

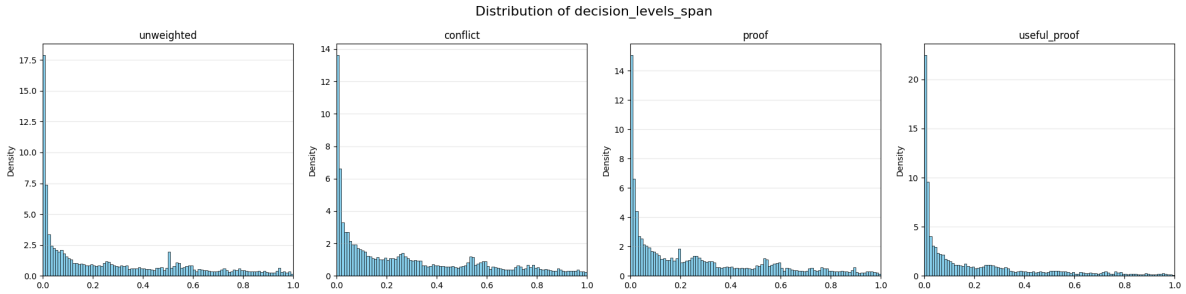

Figure A.9: Density distribution of rankings of *decision levels span* in optimisation instances.

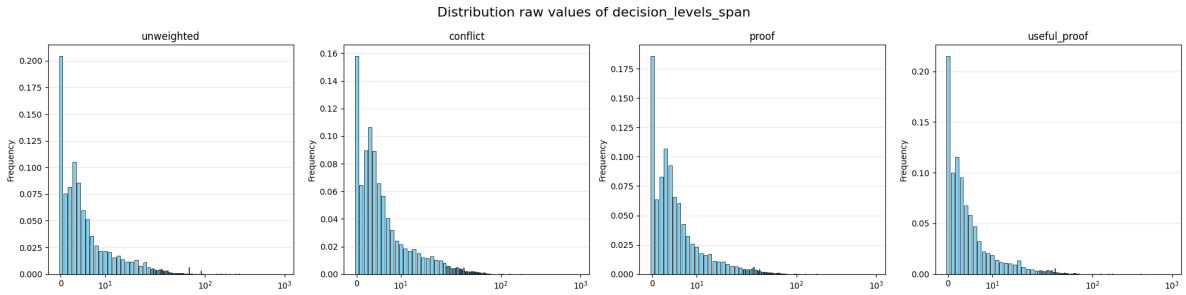

Figure A.10: Density distribution of raw values for *decision levels span*. X-axis is in *symlog* scale, with the linear threshold at 20.

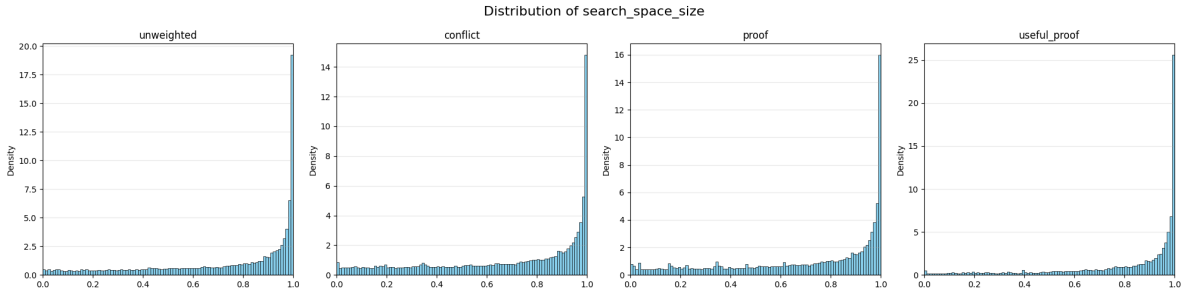

Figure A.11: Density distribution of rankings of *search space size* in optimisation instances.

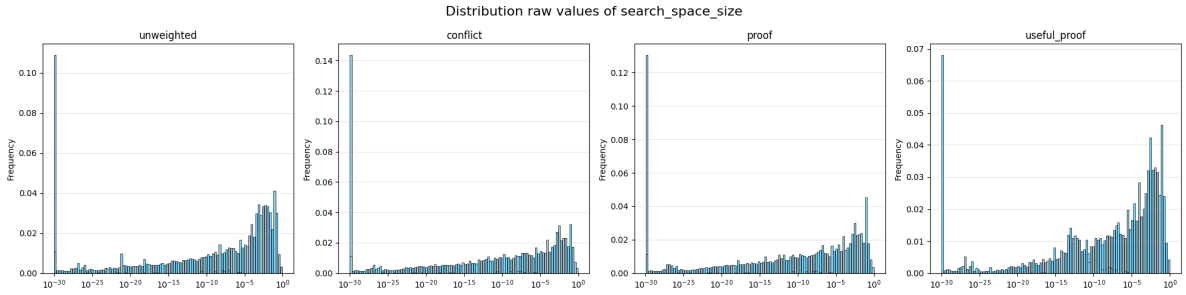

Figure A.12: Density distribution of raw values for *search space size*. X-axis is in log scale, with values clamped at 10^{-30}.

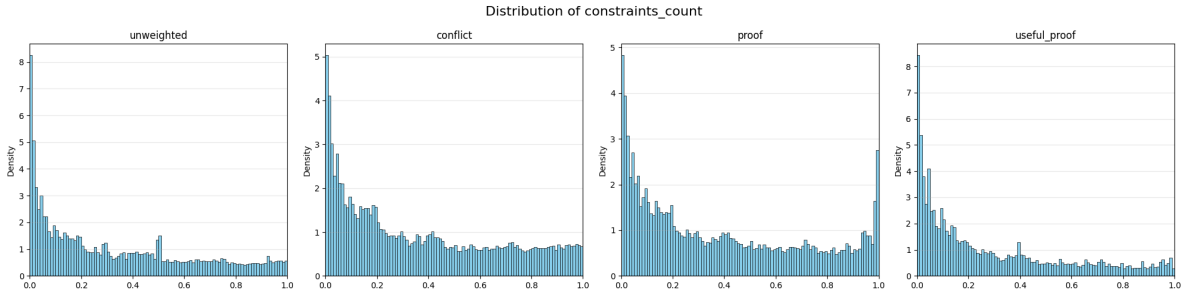

Figure A.13: Density distribution of rankings of *constraints count* in optimisation instances.

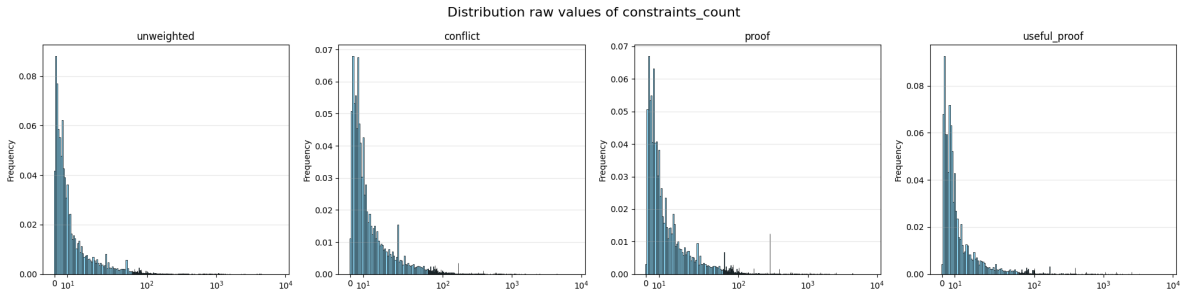

Figure A.14: Density distribution of raw values for *constraints count*. X-axis is in *symlog* scale, with the linear threshold at 60.

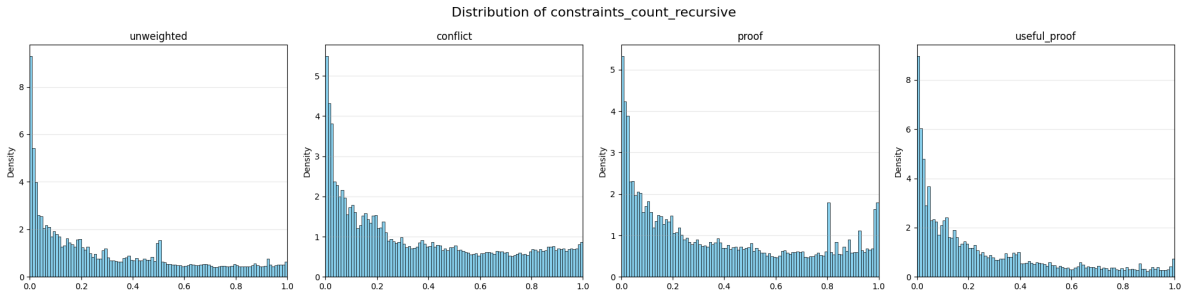

Figure A.15: Density distribution of rankings of *recursive constraints count* in optimisation instances.

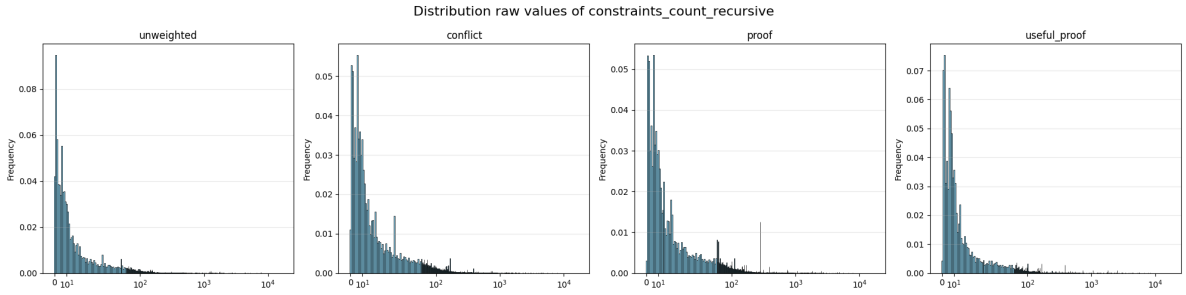

Figure A.16: Density distribution of raw values for *recursive constraints count*. X-axis is in *symlog* scale, with the linear threshold at 60.

A.2. Satisfaction

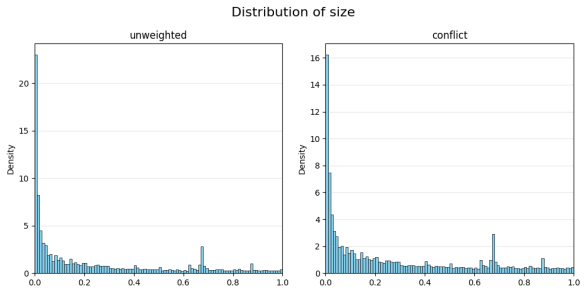

Figure A.17: Density distribution of rankings of *size* in satisfaction instances.

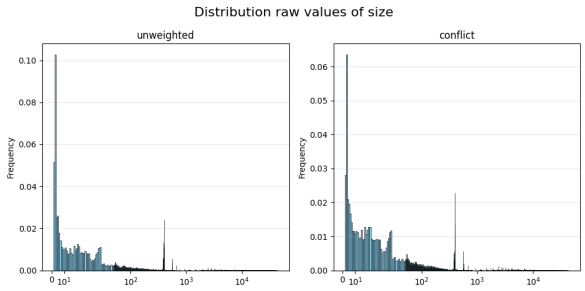

Figure A.18: Density distribution of raw values for *size*. X-axis is in *symlog* scale, with the linear threshold at 50.

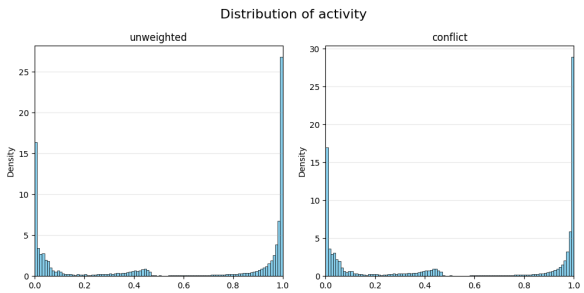

Figure A.19: Density distribution of rankings of *activity* in satisfaction instances.

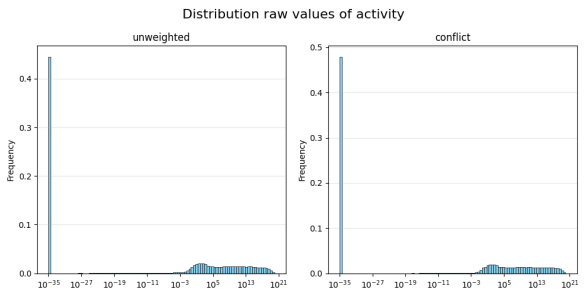

Figure A.20: Density distribution of raw values for *activity*. X-axis is in log scale, with values clamped at 10^{-35}.

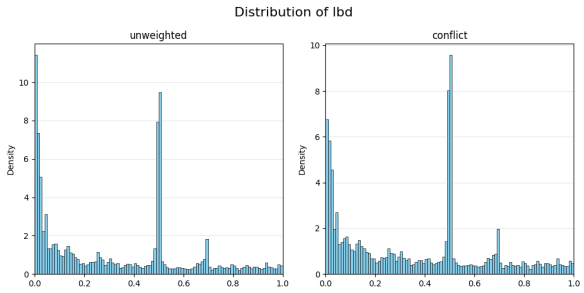

Figure A.21: Density distribution of rankings of *LBD* in satisfaction instances.

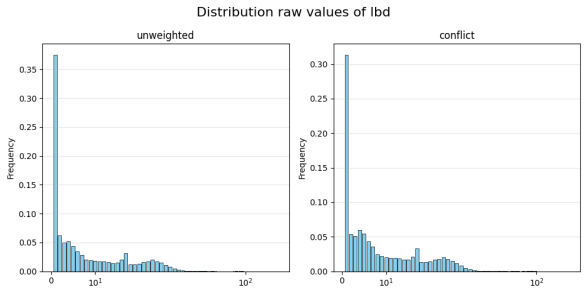

Figure A.22: Density distribution of raw values for *LBD*. X-axis is in *symlog* scale, with the linear threshold at 30.

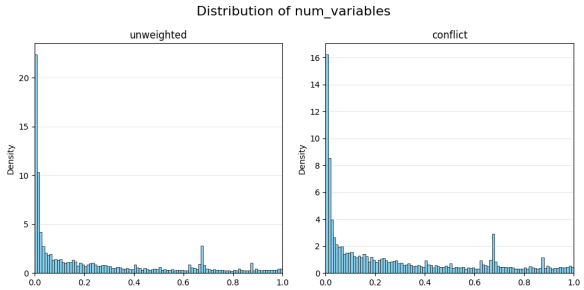

Figure A.23: Density distribution of rankings of *number of variables* in satisfaction instances.

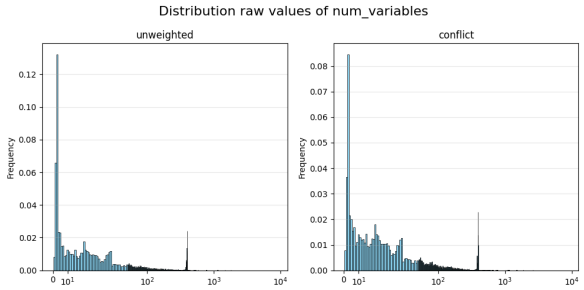

Figure A.24: Density distribution of raw values for *number of variables*. X-axis is in *symlog* scale, with the linear threshold at 50.

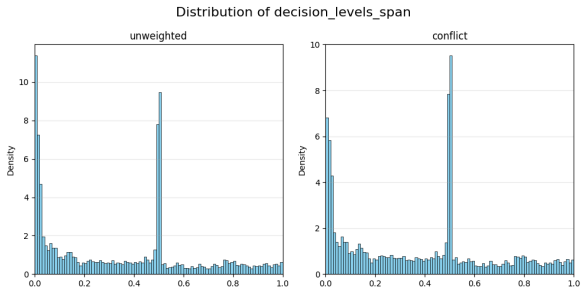

Figure A.25: Density distribution of rankings of *decision levels span* in satisfaction instances.

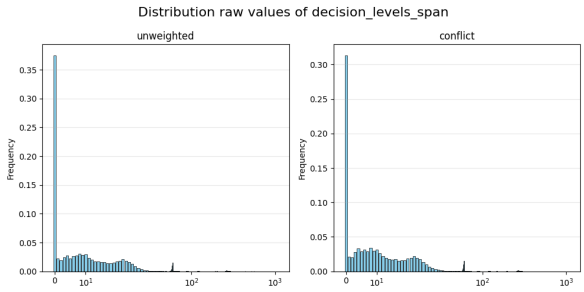

Figure A.26: Density distribution of raw values for *decision levels span*. X-axis is in *symlog* scale, with the linear threshold at 30.

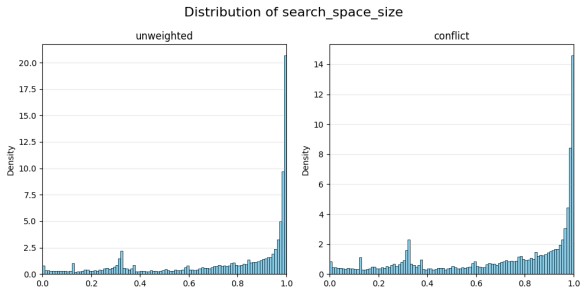

Figure A.27: Density distribution of rankings of *search space size* in satisfaction instances.

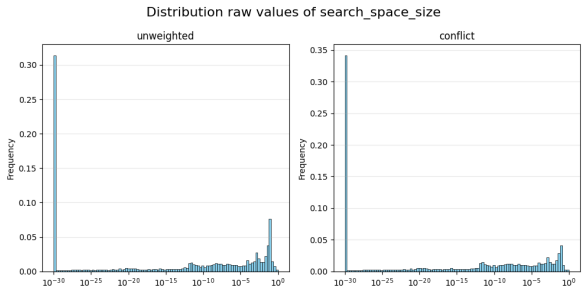

Figure A.28: Density distribution of raw values for *search space size*. X-axis is in log scale, with values clamped at 10^{-30}.

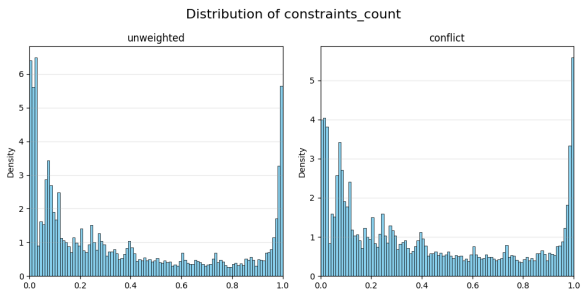

Figure A.29: Density distribution of rankings of *constraints count* in satisfaction instances.

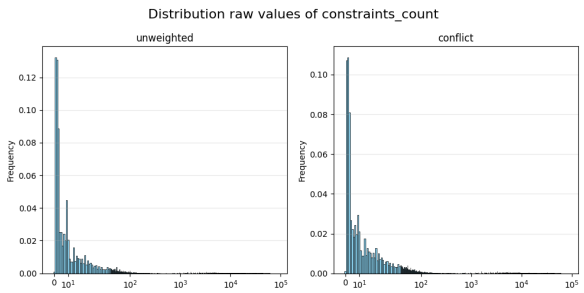

Figure A.30: Density distribution of raw values for *constraints count*. X-axis is in *symlog* scale, with the linear threshold at 40.

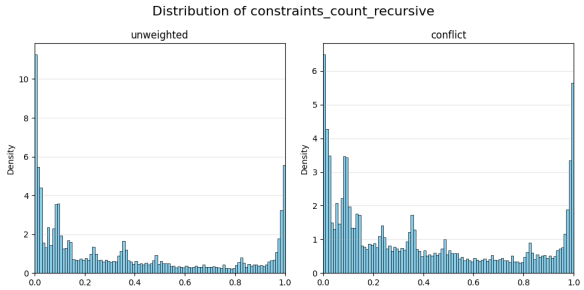

Figure A.31: Density distribution of rankings of *recursive constraints count* in satisfaction instances.

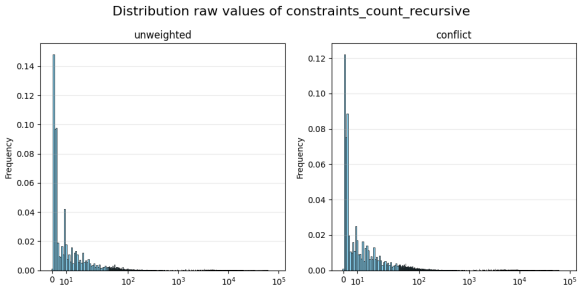

Figure A.32: Density distribution of raw values for *recursive constraints count*. X-axis is in *symlog* scale, with the linear threshold at 40.

A.3. Scheduling

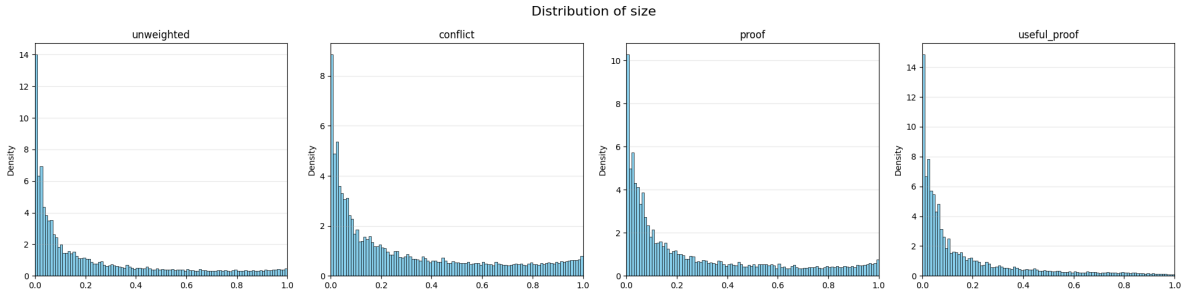

Figure A.33: Density distribution of rankings of *size* in scheduling instances.

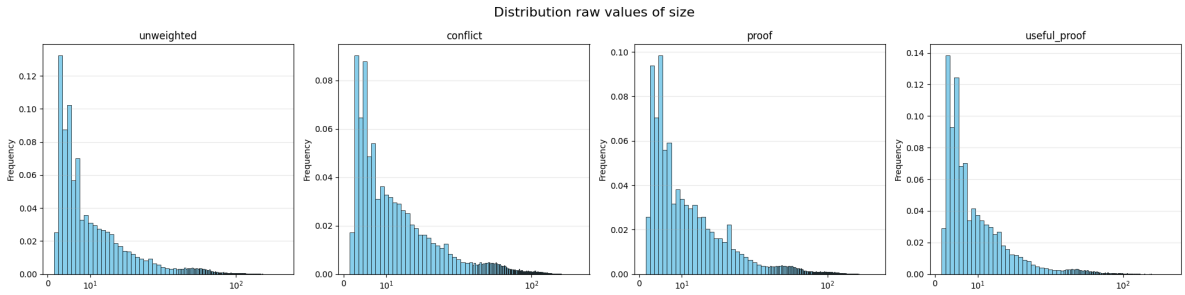

Figure A.34: Density distribution of raw values for *size*. X-axis is in *symlog* scale, with the linear threshold at 30.

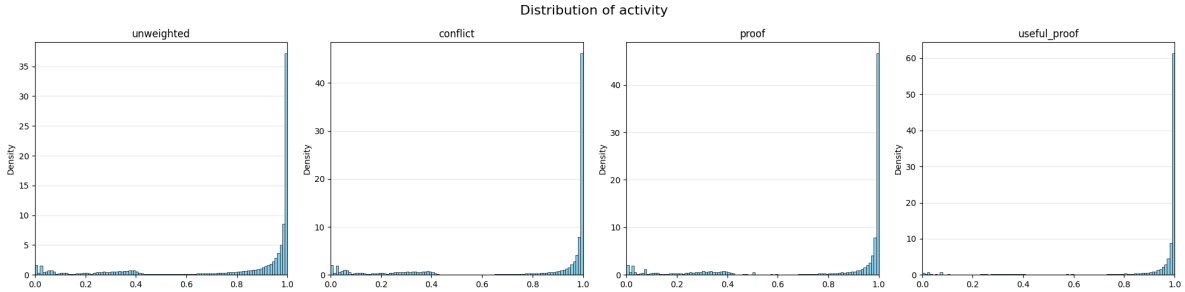

Figure A.35: Density distribution of rankings of *activity* in scheduling instances.

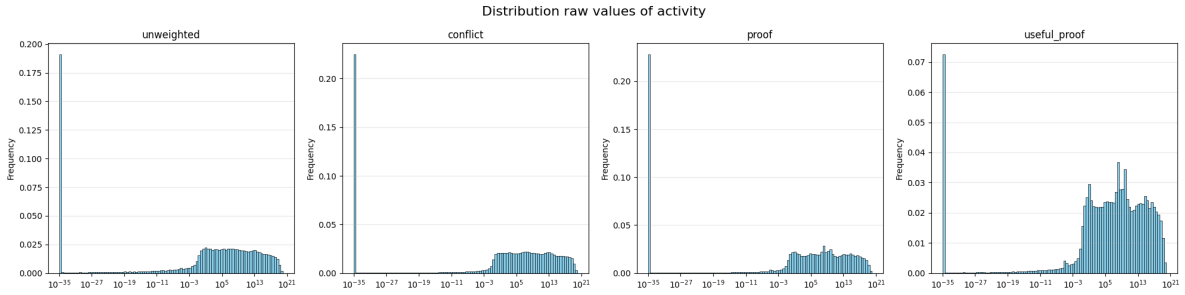

Figure A.36: Density distribution of raw values for *activity*. X-axis is in log scale, with values clamped at 10^{-35}.

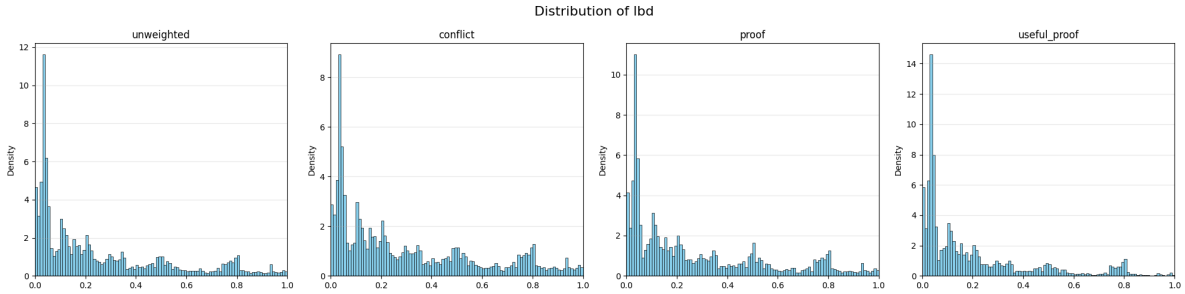

Figure A.37: Density distribution of rankings of *LBD* in scheduling instances.

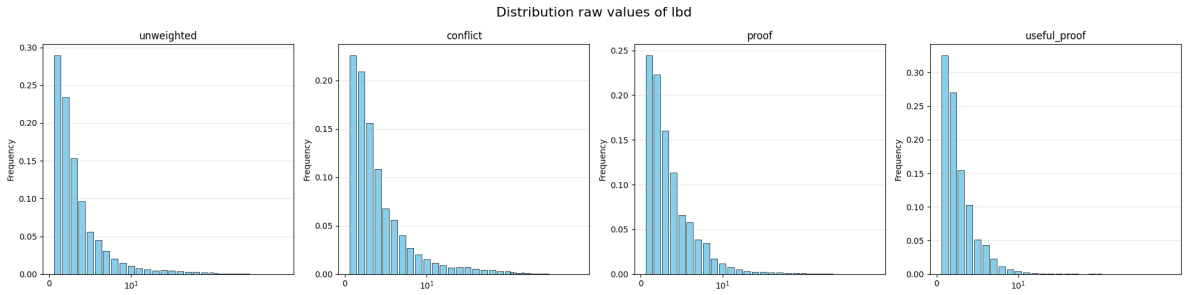

Figure A.38: Density distribution of raw values for *LBD*. X-axis is in *symlog* scale, with the linear threshold at 20.

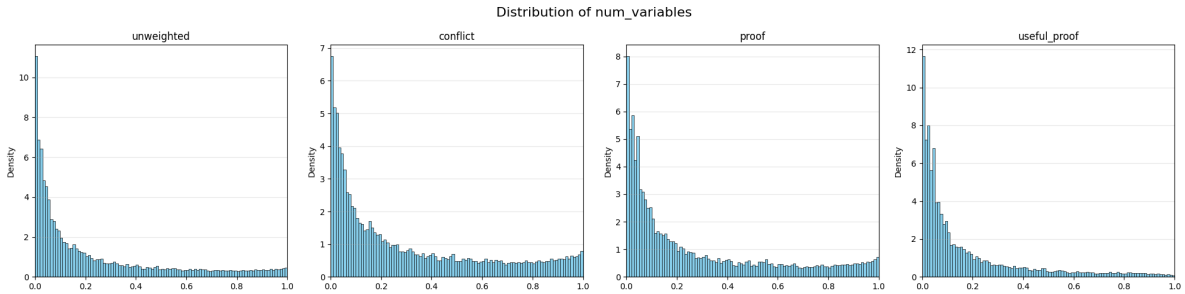

Figure A.39: Density distribution of rankings of *number of variables* in scheduling instances.

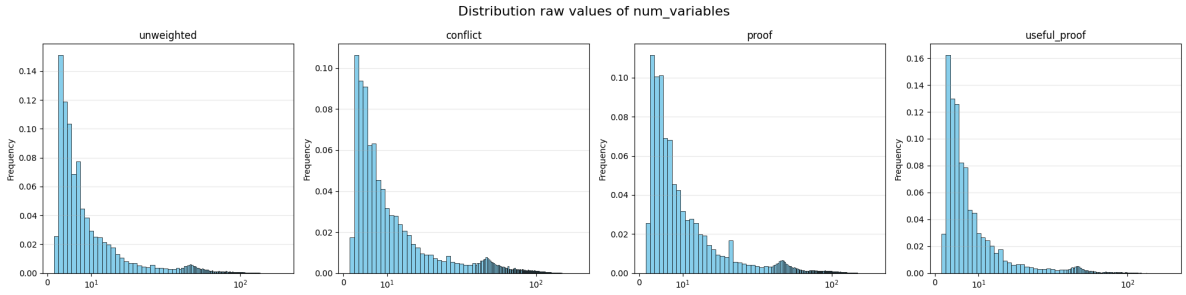

Figure A.40: Density distribution of raw values for *number of variables*. X-axis is in *symlog* scale, with the linear threshold at 30.

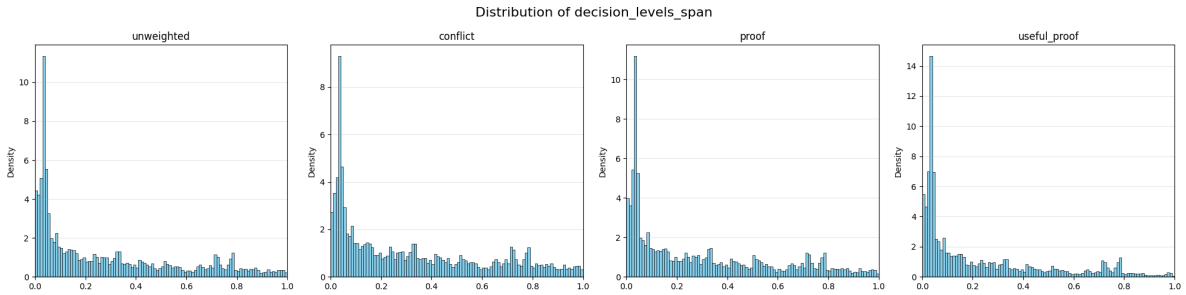

Figure A.41: Density distribution of rankings of *decision levels span* in scheduling instances.

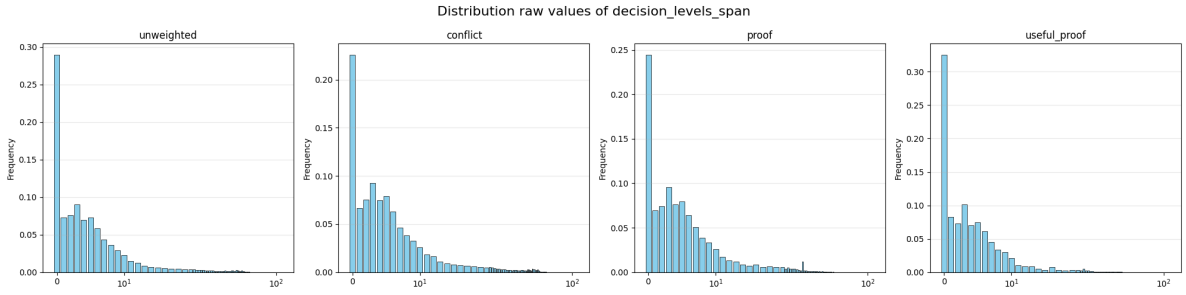

Figure A.42: Density distribution of raw values for *decision levels span*. X-axis is in *symlog* scale, with the linear threshold at 20.

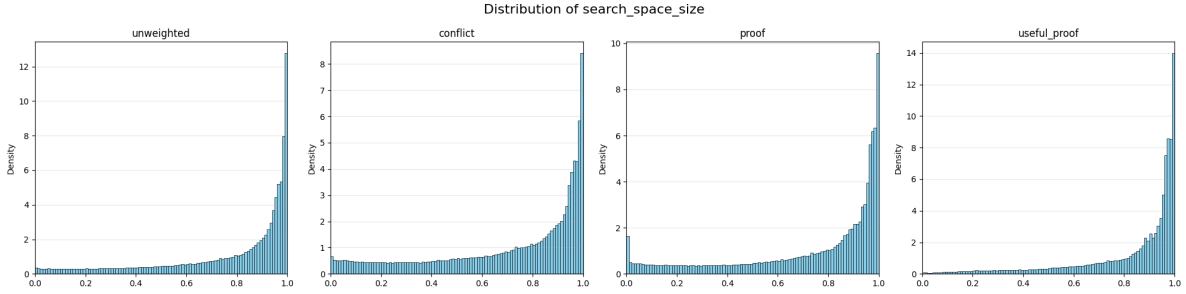

Figure A.43: Density distribution of rankings of *search space size* in scheduling instances.

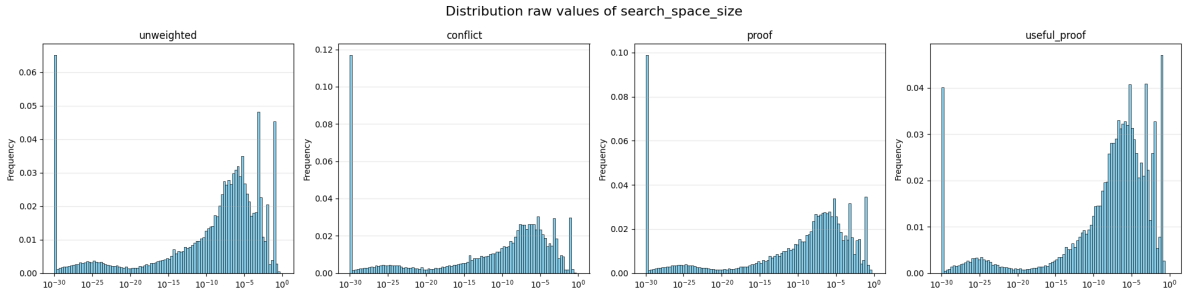

Figure A.44: Density distribution of raw values for *activity*. X-axis is in log scale, with values clamped at 10^{-30}.

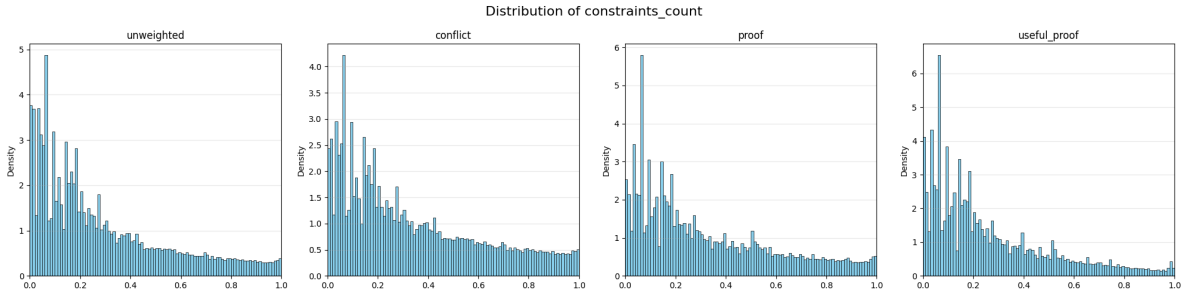

Figure A.45: Density distribution of rankings of *constraints count* in scheduling instances.

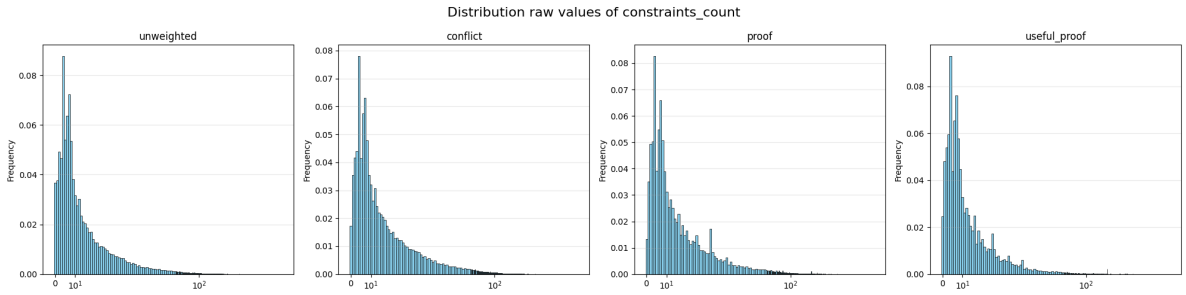

Figure A.46: Density distribution of raw values for *constraints count*. X-axis is in *symlog* scale, with the linear threshold at 60.

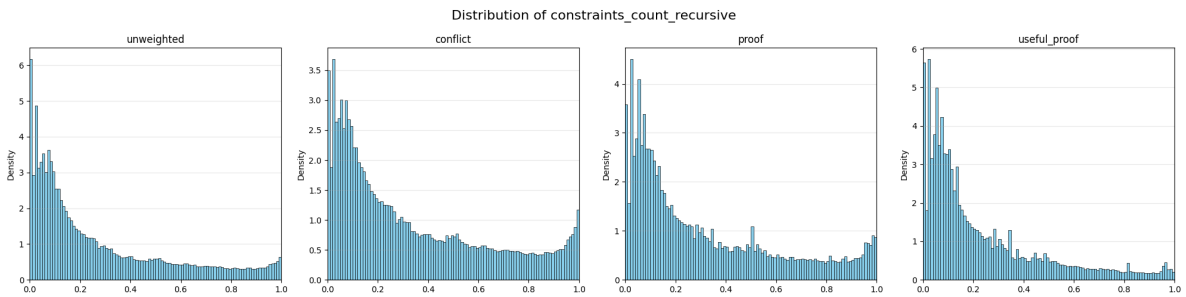

Figure A.47: Density distribution of rankings of *recursive constraints count* in scheduling instances.

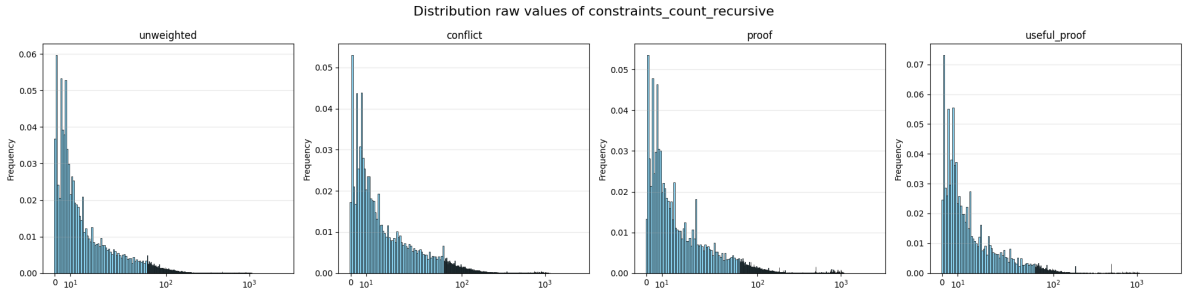

Figure A.48: Density distribution of raw values for *recursive constraints count*. X-axis is in *symlog* scale, with the linear threshold at 60.