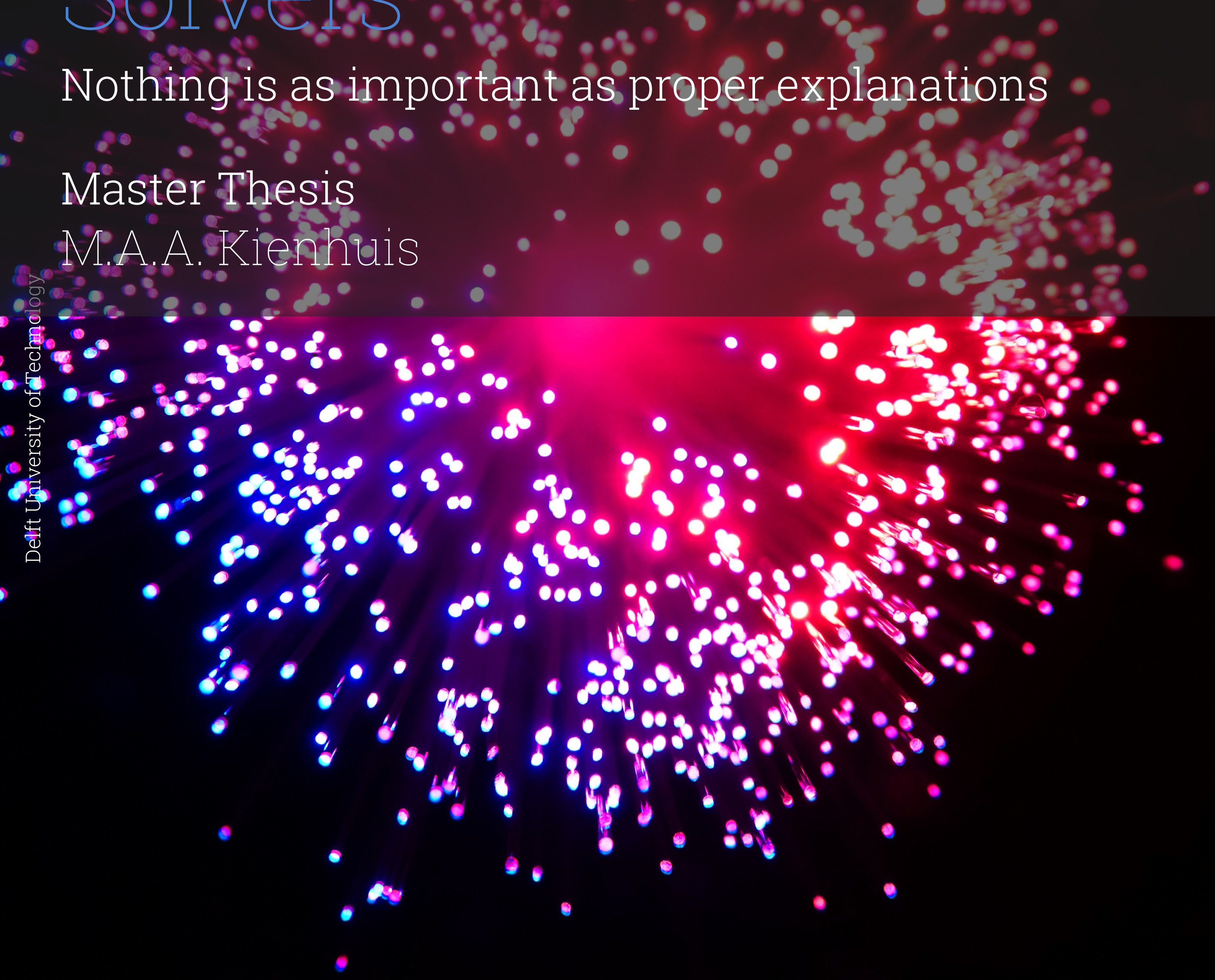
Solving Cumulative with Extended Resolution in LCG Solvers

Nothing is as important as proper explanations

Master Thesis

M.A.A. Kienhuis

Delft University of Technology



TU Delft

Solving Cumulative with Extended Resolution in LCG Solvers

Nothing is as important as proper explanations

by

M.A.A. Kienhuis

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday September 25, 2025 at 10:45 AM.

Student number:	5300126	
Project duration:	November 14, 2024 – September 25, 2025	
Thesis committee:	Dr. E. Demirović	Supervisor
	Dr. B.P. Ahrens	
	Ir. I.C.W.M. Marijnissen	Daily Supervisor

Cover: Fake Fireworks by Maurits Kienhuis

Preface

Over the last ten months I have had the pleasure of conducting my Master Thesis at the Algorithmics group at TU Delft. This will conclude my five year study in Computer Science, a programming journey that started seven years ago because I would not believe the correct solution to the Monty Hall problem. Having coded a simulation in Excel for this problem as my first programming experience it set me on a path I still enjoy walking today.

Throughout the years I have enjoyed working with concrete goals in mind. While the field of Computer Science tends towards black box artificial intelligence my preference lies with techniques which are clear and defined. Courses like Algorithms and Data-Structures all the way through to Artificial Decision Making are full of clever techniques that ask whether what they have achieved is optimal and often require a proof or validation. The focus is on innovation with room for lots of creativity often resulting in elegant solutions. That is why when the possibility arose for me to create my own extension of the LCG protocol I was highly interested.

I have attempted to write this thesis such that my peers may understand it in a single read while anybody not familiar with Computer Science may hopefully not require much more. This means there is a considerable amount of background for those that are interested in the exact foundations of the technique whilst also showcasing many results for those who wish to continue their own research in this direction.

I want to thank my supervisors, Emir Demirović & Imko Marijnissen for their guidance, creative input, and availability. They motivated me to assure the quality of this work. I wish to extend my gratitude for all my friends, first those at the EEMCS faculty for having to endure my ramblings and frustration about the limits of the Rust language and my hack-job solutions it would not accept. And second my group of friends at Proteus-Eretes who checked in on me regularly, supported me through stressed situations, and were always available. Finally I want to thank my parents, Erwin Kienhuis & Irene Nijsen, for their unconditional support and interest in my work.

Now lets explain, what were those ten months all about?

M.A.A. Kienhuis
Delft, September 2025

Abstract

Lazy Clause Generation (LCG) Solving is a state-of-the-art technique for solving finite domain problems. The combination of powerful Constraint Programming (CP) propagators and Conflict Driven Clause Learning (CDCL) from the SAT domain enables both powerful inferences and learning from conflicts. Most research has focused on strengthening propagator explanations but has limited itself to remain within the standard LCG protocol, which only uses basic assignment and bound literals.

This thesis will focus on structurally extending this CDCL resolution protocol with new literals. We investigate the addition of two different literals to this protocol: a partial sum inequality literal and a precedence literal specifically for the cumulative constraint.

We verify previous work on the inequality literal, expand the test suite, and take a dive into metrics specifically related to extended resolution. We propose the novel idea of the cumulative literal, assessing its performance. We evaluate these extensions against both a baseline solver without protocol extensions and against each other across a range of hyper-parameters and search strategies.

Our results show that extending the protocol is not a guarantee for performance, the linear inequality extension remains powerful in some scenarios but fails to improve on a series of benchmarks whilst the cumulative literal shows no improvement at all. Moreover, we find that structurally based extended resolution introduces overlap between the literals in a nogood which negatively affects nogood quality.

These findings showcase that the research is far from done. A powerful technique exists but future research should consider additional infrastructure to make sure these techniques flourish at their fullest potential.

Contents

Preface i

Abstract ii

1 Introduction 1

2 Problem Definition 3

3 Preliminaries 5
- 3.1 SAT 5
  - 3.1.1 CDCL 6
  - 3.1.2 Finite Domains 8
- 3.2 Constraint Programming 9
- 3.3 LCG 10
- 3.4 Cumulative and RCPSP 11
  - 3.4.1 Time-Resource Decomposition 12
  - 3.4.2 Task-Resource Decomposition 12
  - 3.4.3 Linear Inequalities 13
  - 3.4.4 Timetable 13

4 Related Work 16
- 4.1 Decomposition; Relying Purely On SAT 16
- 4.2 Structural Extended Resolution 16

5 Main Contributions 18
- 5.1 Extended Linear Inequalities 18
  - 5.1.1 Sequential Encoding 19
  - 5.1.2 Totalizer Encoding 20
  - 5.1.3 Theoretical Strengths 21
  - 5.1.4 Theoretical Weaknesses 21
- 5.2 The Cumulative Literal 23
  - 5.2.1 Additional Propagators 23
  - 5.2.2 Improvements 24
  - 5.2.3 Downsides 27

6 Experimental Results 28
- 6.1 Experimental Setup 28
  - 6.1.1 Method 28
  - 6.1.2 The Comparisons 28
  - 6.1.3 The Benchmarks 29
  - 6.1.4 The Metrics 30
- 6.2 Linear Extended Resolution 31
  - 6.2.1 Basic Results 31
  - 6.2.2 Multiplicity 36
  - 6.2.3 VSIDS 38
  - 6.2.4 Ordering 41
  - 6.2.5 Naive 44
  - 6.2.6 Overlap 47
- 6.3 Cumulative Literal Results 49
  - 6.3.1 Out Of The Box Comparison 49
  - 6.3.2 Indomain Split 51
  - 6.3.3 VSIDS 53

6.3.4 Minimization. 56
6.3.5 Free Search 57

7 Conclusions and Future Work 60
7.1 Future Work 61

References 62

1 Introduction

Being able to solve difficult problems in a quick and efficient manner is an important part of computer science. Although for some problems clever algorithms exist, there are also problems for which no algorithm is currently known or may never be found. This is where the field of solving was introduced. In the field of solving, problems are modelled as a world of constraints. A general purpose solver takes this world and attempts to find a configuration such that no constraint is violated. In its most pure form a solver can be seen as a brute-force device, trying every candidate solution until a valid solution is found. Modern solvers utilize many tricks to speed up this brute-force approach. With multiple solvers existing there is a trade-off between the solving approaches, search techniques, and inference steps. In 2009 the Lazy-Clause-Generation, LCG, solver [12] was introduced. This solver combined both a boolean satisfaction engine and a finite domain constraint programming engine into a single solver.

This new technique became a strong contender in the field of constraint programming. By utilizing the learning element of the satisfaction engine and the flexible yet powerful propagators of the constraint programming paradigm, it quickly became state-of-the-art [12] for multiple problems. However, the door was left open to future research into the protocol that connects the two solving engines. The two engines need to communicate in order to keep their views consistent. More importantly, they communicate to explain why they make certain decisions, utilizing each other to gain information. The current communication protocol only allows the use of the literals $[x = n]$ or $[x \leqslant n]$ as a means of explaining. There are scenarios where this grammar lacks the required complexity to properly model an event. In such a case the solver has to default to a weaker explanation resulting in gaps in the learning procedure slowing down the solver as it has to re-explore paths it would otherwise already have pruned.

Seeking to improve the connection between the engines, a number of new literals were introduced. Chu and Stuckey [7] introduced multiple new explanations based on the concept of structural based extended resolution affecting multiple constraints such as but not limited to *linear*, *lex*, & *disjunctive*. Schutt and Stuckey [23] utilizes the concept to introduce a new literal that also helps with transitive deductions to speed up the producer-consumer problem. By introducing new literals into the communication protocol both papers achieved a significant speed-up [7, 23].

However, research into these new literals is limited. The extensions are usually only tested by a single solver with a limited number of benchmarks relevant to each extension [7]. A solver may also be extended in multiple ways but without an ablation study taking place performance is attributed to all extensions being beneficial. In addition, not all global constraints were approached or documented. As such there exist no extended resolution techniques regarding the global constraint cumulative, present in many scheduling problems. While significant work has been done to properly explain this constraint [24] this hole remains. This also holds for other constraints and together with the lack of extensive testing shows an avenue for further research.

This paper will therefore do the following. We initially provide a deep-dive into extended resolution for the linear inequality constraint originally introduced by Chu and Stuckey [7] and introduce the totalizer encoding as an alternate construction. Additionally we investigate a novel approach where we utilize extended resolution to improve the cumulative constraint by extending the explanations created by the timetable propagator. The techniques will be implemented on the pre-existing solver Pumpkin. The original benchmarks will be repeated for verification and extended upon to show the effectiveness of structural based extended resolution.

We improve the current knowledge base twofold. By applying a wider range of benchmarks on the linear inequality extension we verify if the technique represents a more general uplift in performance or if there are test instances which do not see the expected improvements. By testing the totalizer encoding and approaching the cumulative constraint we are able to answer whether or not these approaches work and add something novel to the knowledge base.

Our results show that linear structural based extended resolution works well in general on instances that natively contain linear inequalities but fails to improve when it comes to the cumulative linear decompositions. The new totalizer encoding is a more robust encoding when beneficial variable orderings are not known. The novel approach for the timetable propagator also fails to perform remaining at an equal level of performance as a solver without extended resolution.

Apart from the performance results we take a look at the nogoods that are typically derived by such solvers and learn that in almost all scenarios there is the potential for duplication within the nogoods that is currently not parsed out by standard minimization techniques. This duplication in turn negatively effects the quality of the nogoods leading to a decrease in performance. We end the research by concluding that while extended resolution has a lot of potential any future work should investigate possible minimization rules to prevent structurally based extended resolution from performing worse than a non extended solver.

This thesis is laid out as follows. We introduce a small example to concretely explain the problem we are solving in chapter 2 and state the specific research questions. We then go over all the preliminaries required to fully understand what an LCG solver is in chapter 3. Having discussed the foundation we move on to the related works in chapter 4 discussing previous work on structural extended resolution and alternative approaches. Chapter 5 outlines the exact proposal of the extended resolution techniques provided by this thesis. We explain our method and analyse the results in chapter 6. The thesis will conclude in chapter 7 with a summary of the findings and a proposal for future research.

2 Problem Definition

We begin by exploring the concept of Lazy-Clause-Generation (LCG) Solving. An LCG solver is a finite domain solver combining the strengths of a regular Constraint Programming (CP) solver with a boolean domain Conflict Driven Clause Learning (CDCL) SAT solver. In essence we have a regular CP solver but there is one additional propagator added, the nogood/unit propagator. The nogood propagator keeps track of a series of nogood scenarios, e.g. $[\![x = 3]\!]$ results in a failure, that were learned at some point in the solving process, it then avoids such scenarios whenever possible.

In order to apply the CDCL procedure, a SAT only technique, we require a connection from the CP domain to the SAT domain. This is typically done by decomposing every finite domain variable into a set of boolean literals. A literal can either be a bounding $[\![x \geqslant n]\!]$ or an assigning $[\![x = n]\!]$ literal. This allows us to fully mimic the CP domain in our SAT engine. The next step is populating the trace, a record of all inferences and decisions, for our CDCL procedure with information. This is done by creating explanations upon every propagation by a CP propagator utilizing the SAT decomposition to explain why we made a propagation. e.g. $[\![x = 1]\!] \wedge [\![y \neq 2]\!] \rightarrow [\![z = 3]\!]$. If $[\![z = 3]\!]$ then becomes the reason we are in a conflict we may learn that $[\![x = 1]\!] \wedge [\![y \neq 2]\!]$ is a nogood scenario as it directly led us to a conflict.

However such explanations are often limited in their expressive power. When considering a linear inequality such as $x+y+z \leqslant 10$ we can only explain a propagation by means of the current assignment of the variables. e.g. $[\![x \geqslant 2]\!] \wedge [\![y \geqslant 3]\!] \rightarrow [\![z \leqslant 5]\!]$. While this is sufficient note that if $[\![z \leqslant 5]\!]$ ever becomes the reason of a conflict we can only learn that $[\![x \geqslant 2]\!] \wedge [\![y \geqslant 3]\!]$ is a nogood scenario. A more powerful explanation would be to say that $[\![x + y \geqslant 5]\!] \rightarrow [\![z \leqslant 5]\!]$. In this case the nogood now covers all possible expressions leading to $[\![z \leqslant 5]\!]$ meaning our explanation is much more general and we can avoid more of the search space as our original method would fail to foresee the conflict at $x = 3, y = 2$ as not both literals are set to true.

This is where structurally based extended resolution comes in. Structural based extended resolution, named extended resolution for the remainder of this thesis, is the practice of adding auxiliary variables to model more complex relations in the explanations. To continue our example, a candidate would be the linear inequality constraint where we model any partial sum as a set of extra literals, $[\![x+y+\cdots \geqslant n]\!]$, available to use in our explanations. This way it is possible to find much more general nogood scenarios in turn pruning larger parts of the search space via our CDCL procedure.

We investigate these approaches as current research does not test linear extended resolution on an extensive set of benchmarks. Current research mainly focuses on those benchmarks where the problem naturally consists of large linear inequalities such as knapsack or fails to consider linear extended resolution in a vacuum typically combining it with other extended resolution techniques. The totalizer encoding, relying on a tree like structure for consistency instead of the typical linear approach, is a novel method for which the trade-offs are currently unknown. We therefore think that these are good candidates to explore in our research.

One of the main focuses of this work will be the cumulative constraint where we look at extended resolution as a means of improving the quality of the found nogood scenarios. We do this in several approaches. The first two decompose the cumulative constraint via the time and task decomposition [24] into linear inequalities. We then utilize the linear extended resolution technique with $[\![\sum x_i \leqslant n]\!]$ to potentially solve these problems faster by explaining our inequalities in a more general manner. In our third approach we create a new literal specific to the timetable propagator to model a ”starts after compulsory section ends” literal, $[\![i \gg j, k, r]\!]$, hence named the cumulative literal. Indicating that a task i must start after the first task of j, k, r ends.

There are no known results at the time of writing for utilizing extended resolution specifically with the cumulative constraint. Extended resolution often requires a lot of implementation for the specific constraint under investigation. The more general variables introduced by extended resolution are therefore often only applicable to a few constraints. This means that most of the research has gone into specific constraints of which cumulative is not one we are aware of. We therefore wish to investigate what the effects of extended resolution for the cumulative constraint could be.

Given the aforementioned gaps in the knowledge base this research aims to answer the following questions:

1. Does linear extended resolution lead to fewer conflicts found when considering a broader set of benchmarks.
2. How does linear extended resolution impact the decompositions of the cumulative constraint.
3. How do the sequential and totalizer encoding perform in comparison with each other.
4. What characterizes the quality of a nogood with linear extended resolution.
5. Does extended resolution in the form of the cumulative literal lead to fewer conflicts found when solving the RCPSP problem.
6. What characterizes the quality of a nogood in relation to the cumulative literal.

We hypothesize that adding extended resolution improves the quality of the learned nogood scenarios in all cases resulting in lower conflicts. Given the precedence that extended resolution has shown consistent improvements [7] and the theory states that every explanation is more general this seems like a reasonable hypothesis. Between the sequential and totalizer encodings we hypothesize that the sequential encoding outperforms the totalizer encoding in scenarios where variable orderings are known while the absence of such orderings would see the totalizer remain a strong contender. Finally we hypothesize that the cumulative literal will show the biggest increase in performance as it will utilize extended resolution specifically created for the cumulative constraint.

3 Preliminaries

In this section we introduce how a lazy clause generation solver works. To properly create this understanding we first delve into how a satisfaction solver works drawing inspiration from the handbook of satisfiability [5]. Then secondly, we take a look at how a constraint programming solver works [12]. The final section then goes into how these two solvers are combined to form the lazy clause generator solver [12].

3.1. SAT

Satisfaction solving attempts to solve the boolean satisfiability, SAT, problem. In essence the goal is to find an assignment of true and false values for a boolean formula to be evaluated to true. SAT solvers generally decompose any formula into a Conjunctive Normal Form, CNF, formula first. Each disjunction within this CNF decomposition is viewed as a clause to satisfy. If all clauses are satisfied a solution to the formula is found.

At the core of a SAT problem are boolean literals $l_i \in B$ with l_i being set to one of $\{\top, \bot, unassigned\}$ via the value function $v(l_i)$. A literal l_i can also be negated as $\overline{l_i}$ or $\neg l_i$. Note that the value function accepts negated literals such that if a negation gets set to a truth value $v(\overline{l_i}) = \top$ this implicitly means that $v(l_i) = \bot$. In abuse of notation we may therefore only mention the value function working on non negated literals $v(l_i)$ as the negation is only relevant for assignment of this function. The $unassigned$ value reflects that the solver has yet to assign either $\top$ or $\bot$ to the literal and as such this value does not appear in any valid output of the solver.

A clause $c_i \in C$ is a disjunction, $\vee$, of literals l_i. There exist four ways a clause can be evaluated *falsified*, *satisfied*, *unit*, or *unresolved* [5]. In a falsified clause all literals are evaluated to $\bot$. A satisfied clause has at least one literal evaluated to $\top$. A clause is deemed to be unit if exactly one literal is $unassigned$ and the rest of the literals are evaluated to $\bot$. Any clause not categorized into these three categories falls into the fourth *unresolved* category.

The formula ϕ is then defined as the conjunction $\wedge$ of all the clauses $c_i \in C$. This formula represents the problem to be solved. Or more informally, the goal is to make sure every clause of this formula is satisfied. The resulting structure obtained through this definition is known as Conjunctive Normal Form, CNF, visualized in equation (3.1). As CNF only utilizes the logical operators $\{\wedge, \vee, \neg\}$ certain constraints will have to be transformed to fit the aforementioned structure. However, a CNF encoding is capable of modelling any boolean formula as it is functionally complete [26].

$$\underbrace{\overbrace{(l_1 \vee \overline{l_2} \vee \cdots \vee l_n)}^{\text{A clause } c_i} \wedge (l_1 \vee \overbrace{l_3}^{\text{A literal } l_i} \vee \cdots \vee l_n)}_{\text{The formula } \phi} \tag{3.1}$$

Vital in preventing wrongful decisions made by the solver are the unit clauses. The unit clause rule [10] states that if a clause is unit that the singular remaining literal l_i should be set to true $v(l_i) = \top$. This rule can have a cascading effect and is therefore repeated until no unit clauses remain in the formula. Efficient data-structures such as watch literals [5] exist to apply this rule recursively on only the affected clauses resulting in very efficient application of this rule.

Based on these rules Davis, Logemann, and Loveland [9] introduce the DPLL algorithm, a technique for automated theorem solving. This algorithm takes as input a formula ϕ and returns whether ϕ is satisfiable or not. The DPLL algorithm is a recursive algorithm. It starts of by checking ϕ for any unit clauses and applying the unit clause rule until no such clause remains. It also considers pure literals, literals who only appear in a single polarity in the problem, as setting these literals to true will satisfy any clauses they are present in these clauses are not relevant to the satisfiability of the formula and are therefore eliminated from the problem. After this the base cases are checked, two variations exist. The first base case occurs if there is a falsified clause within ϕ returning a $\bot$. The other base case occurs if all clauses in ϕ are satisfied returning $\top$. If no base cases are reached the splitting rule is utilized. A literal l_i is chosen and two recursive calls are made. One with $v(l_i) = \top$ and one with the complement $v(l_i) = \bot$. Alternatively the DPLL algorithm can be described by the following pseudocode.

Algorithm 1 The DPLL algorithm

```
function DPLL($\phi$)
    while $\exists c_i(c_i \in \phi \wedge unit(c_i))$ do
        $\phi \leftarrow unit\_clause\_rule(\phi, c_i)$
    end while
    while $\exists l_i(l_i \in \phi \wedge pure(l_i))$ do
        $\phi \leftarrow pure\_literal\_elimination(\phi, l_i)$
    end while
    if $\exists c_i(c_i \in \phi \wedge falsified(c_i))$ then
        return $\bot$
    else if $\forall c_i(c_i \in \phi \wedge satisfied(c_i))$ then
        return $\top$
    end if
    $l_i \leftarrow variable\_selection(\phi)$
    return DPLL($\phi \wedge l_i$) $\vee$ DPLL($\phi \wedge \overline{l_i}$)
end function
```

Where $\phi \wedge l_i$ represents an abuse of notation to show that the call continues with the formula ϕ and the literal l_i assigned the respective polarity. The variable selection procedure is a procedure which picks the next literal to branch on. This procedure is a study of heuristics which is an entire field of its own and often customized to suit the type of problem the formula encodes. Also note that no special back-jump has been implemented, while this will be discussed in subsection 3.1.1 the DPLL algorithm backtracks by relying on its recursive implementation.

This gives us all the tools to build a SAT engine. We have a series of boolean literals $l_i \in B$ that are used to construct clauses c_i. If all the clauses are satisfied a solution to the problem is found. The engine repeats the process of selecting a new literal and assigning it a value, applying the unit clause rule, and backtracking if necessary. This process may either terminate with a valid candidate solution if one is found or return $\bot$ after an exhaustive search.

3.1.1. CDCL

This generally covers the search and inference approach but it is missing a vital component of modern day SAT solving, learning. Also known as the Conflict-Driven-Clause-Learning, CDCL, approach it is the sole reason modern SAT solvers are capable of dealing with millions of literals [5].

A CDCL solver hosts a conflict analysis algorithm which is responsible for providing informative reasons as to why the conflict occurred. This enables a solver to apply non-chronological back-jumping allowing it to backtrack multiple decision levels at once without skipping over any potentially valid branches. This informative reason is called the learned clause or a no-good. This learned clause is added to the total set of clauses that describe the problem allowing the solver to apply the unit clause rule onto this learned clause as well.

The core strength of a CDCL solver comes from resolution. Resolution is a logic inference rule applicable to the propositional logic that a SAT solver is built on. Resolution is defined as per equation (3.2).

$$\curlywedge = \frac{(a_1 \vee a_2 ... \vee c), (b_1 \vee b_2 ... \vee \overline{c})}{\therefore a_1 \vee a_2 \vee ... \vee b_1 \vee b_2 \vee ...} \tag{3.2}$$

With the informal definition that two clauses can be combined into a single clause by copying over every literal and removing those which have both polarities present. Resolution itself can also be utilized as a proof system and it has been shown that resolution CDCL solvers are just as strong as resolution as long as they implement the restart feature [4]. A strong solver is desired as this means the solver can generally reach a valid solution within less decisions and / or conflicts.

In order to facilitate this process the tracking of the search process and the order of assignments of the literals must be kept track off. This ledger is known as the trace and it can be encoded by assigning a number of properties to each literal. These properties are the decision level, $\delta(l_i) \in 0, 1, ..., |C|$, and the antecedent $a(l_i) \in C$. The decision level encodes at what step in the search process the solver assigned a value to the literal l_i. This may also appear in the shorthand form of $l_i = \top@2$ denoting a value assigned for the literal l_i to the value $\top$ at decision level two. The current decision level δ is only incremented if a literal is assigned a value through the variable and value selection process, not if the assigned was forced by propagation like the unit clause rule. Therefore there could be multiple literals with the same decision level. The antecedent encodes which clause was unit resulting in the assignment of this literal. As not all literals will have an antecedent, e.g. the variable and value selection procedure produces no antecedent, this value could be undefined.

The goal of conflict analysis is to find an asserting conflict-driven clause. This means we are looking for a learned clause with exactly one literal from the conflicting decision level. While one could argue that all assigned literals and their assignments are responsible for the conflict this leads to large clauses that are far away from the actual conflict which is often local to the recent decisions [5]. A more effective way of finding these clauses is through Unique Implication Points, UIP. A UIP is a literal assignment event in the trace of the SAT engine that by itself is sufficient to imply the contradiction [5]. e.g. $v(c_i) = \top \rightarrow \bot$ where $v(c_i) = \top$ might cascade through multiple clauses before resulting in $\bot$. The cascading clauses form a more local reason that explains the conflict together with the falsified clause itself. Note that UIPs can both be found multiple times at a single decision level as well as over different decision levels.

The UIP closest to the conflict is called the 1UIP which is the first UIP found on the conflict decision level [5]. While it is possible to also learn extra no-goods based on other UIPs in the trace it has been shown that this tends to worsen a solvers performance. This loss of performance is attributed to the lower generality and higher distance in relation to the conflict [11, 28].

The 1UIP can be found through a backtracking algorithm that utilizes resolution. We start of by creating a queue to keep track of literals we still need to check. We then take the falsified clause c_c and add all literals that follow: $\delta(l_i) = \delta \wedge l_i \in c_c \wedge a(l_i) = \varnothing$ to our queue. Meaning any literals from the current decision level that are in the conflict clause but not the decision literal. The conflict clause c_c is then renamed as the learned clause c_l. A literal l_i is then popped from this queue. The literals of the antecedent $c_a = a(l_i)$ are then scanned for enqueuing with the same rule as the conflict clause $\delta(l_i) = \delta \wedge l_i \in c_a$ ignoring any literals we have already enqueued before. We then update our learned clause with resolution $c_l \leftarrow c_l \curlywedge c_a$. This process repeats until only a single element is left in the queue. Upon termination the resulting learned clause c_l gets added into the clause database $C \leftarrow C \cup \{c_l\}$.

Example 1: 1UIP

Consider the following formula $\phi : c_1 \wedge c_2 \wedge c_3$

$c_1 = \neg h \vee \neg e \vee f$

$c_2 = \neg e \vee g$

$c_3 = \neg f \vee \neg g$

And a variable value selection procedure which assigns a found literal to be true at the root. A SAT solver will construct a trace like. $h = \top@1$, $e = \top@2$, $f = \top@2$, $g = \top@2$ and then find a conflict in

$\neg f \vee \neg g$. We note that $c_c \leftarrow c_3$. We enqueue the relevant literals f, g and state that $c_l \leftarrow c_c$. We then apply the 1UIP procedure to find a new learned clause on c_l.

First we pop the literal g from the queue and note the antecedent 2. We apply resolution $c_l \leftarrow c_l \curlywedge c_2$, $c_l = \neg f \vee \neg e$. This removes the g literal from our conflict clause but we still have two literals from the current decision level. Next we apply another iteration of resolution $c_l \leftarrow c_l \curlywedge c_1$, $c_l = \neg h \vee \neg e$. We are now left with only one literal from the current decision level, e, and can add the current c_l to our formula ϕ. We then jump back to the second highest decision level in the learned clause which is $\delta(h) = 1$. We then resume the search procedure and end up with the following trace:

$h = \top@1$, $e = \bot@1$, $f = \top@2$, $g = \bot@2$ resulting in a valid solution of ϕ.

Once the conflict analysis is complete the engine back-jumps to the second highest decision level δ_j of the literals $l_i \in c_l$. Back-jumping is different from backtracking in that we may jump back multiple decision levels at once. In this scenario the solver takes all literals l_i past this decision level $\delta(l_i) > \delta_j$ and unassigns them $v(l_i) = undefined$. The decision level is then reset to the back-jump level $\delta \leftarrow \delta_j$. This makes it such that the learned clause is now unit at the current decision level. The solver process then continues with the unit clause rule step of the solving process.

Depending on the clauses learned and the configuration of the solver the number of learned clauses is capable of exploding. Therefore SAT solvers are able to prune their learned clause database. However, this limited size may prevent the solver from learning enough learned clauses to properly solve a problem. Alternative techniques such as phase saving [21] and restarts [18] can help lower the number of learned clauses required as these generally help with focusing on relevant and active conflicts allowing the pruning of clauses related to old and solved conflicts.

In order to guide the process a SAT engine might employ a heuristic for its decision making process. Variable State Independent Decaying Sum, VSIDS, [18] is a state of the art heuristic which guides a SAT engine in its decision making process to focus on recent conflicts. To apply VSIDS the engine keeps track of a score for each literal and its polarities l_i and $\overline{l_i}$. When a learned clause c_l is added to the database the score of every literal $l_i \in c_l$ is incremented. Note that we increment on a combination of literal and polarity. Then periodically these scores are divided by some constant. When the solver needs to make a decision it assigns the highest scored literal. This way VSIDS guides the solver to assign literals closely related to current conflicts. This heuristic is capable of speeding up search on difficult problems without hurting the performance of easier problems [18].

3.1.2. Finite Domains

SAT and all of its descendant techniques solely consider the prospect of boolean literals. While this is a powerful paradigm not all problems are based on boolean literals such as finite domain problems. In order to utilize SAT solvers on finite domain problems we require an encoding to represent the finite domain as a series of boolean literals.

The most common type of encoding is the one-hot encoding. e.g. if an integer variable $x = \{1..n\}$ is defined then n boolean literals representing the the variable assignment value $[x = 1]...[x = n]$ are created. Together with these n literals we require additional constraints to enforce consistency. An example of such a constraint is $\forall_i([x = i] \iff \forall_j(\neg[x = j] \vee i = j))$ with $i, j \in \{1..n\}$. This results in an extra number of clauses as overhead of which the bound is defined by the specific decomposition you apply and how the consistency constraints are structured.

An alternative method utilizes a log encoding. In the log encoding a literal l_i now represents a digit in a binary number, similar to how an integer is stored on most computers with the literal representing a single bit. This encoding requires only $log_2(|x|)$ literals to represent the bits for a single variable x. It also requires fewer atomic constraints since it only needs to create the lower and upper bound constraints. This encoding is generally more compact making it useful for variables with large domain yet the encoding typically has poor propagation performance [2].

This allows SAT to deal with finite domains but is still missing the constraints required to model a problem. Similar to how we decompose the finite domain so too is it possible to decompose any finite domain constraint into multiple boolean constraints. Certain constraints do have efficient decompositions that allow for fast SAT implementations [2]. But worst case scenarios do exist where every permutation of variable assignment might need to be decomposed into CNF before the SAT engine can attempt to solve it.

3.2. Constraint Programming

Instead of relying on SAT solvers and boolean decomposition it is also possible to utilize a solver that natively supports finite domains. This is where the concept of Constraint Programming, CP, is introduced. Instead of relying on boolean literals and decomposition this paradigm has native support for finite domain variables and their relevant constraints. Leveraging this native support we open up other avenues of approach. We now define what CP is and what those avenues of approach are.

At the core of a CP problem we have finite domain variables $x_i \in V$. These variables can be contiguous or have holes in their domain. In order to properly describe these variables we introduce the concept of a domain. A domain D_j represents a mapping from a fixed (finite) set of variables V to finite sets of integers [12]. That is to say $D_j(x_i)$ returns the domain of the variable x_i given the current configuration D_j. e.g. assume $D_1 = \{x_1 : \{1..10\}, x_2 : \{1, 2, 4\}\}$ then $D_1(x_2) = \{1, 2, 4\}$ returns all the values in the domain of x_2. The goal is then to shrink the domain such that for every variable a set of size one is returned. The resulting domain, assuming it does not violate any of the constraints, would be a valid solution.

Multiple domains D_j exist and can have different properties. A domain represents the current partial candidate solution of the solver, and as such once when possible values are removed or a variable is fixed to a certain value this represents a new domain. A false domain, resulting in an invalid candidate solution, is a domain D_j which maps a variable to an empty set i.e. $\exists x_i \in V(D_j(x_i) = \emptyset)$. Domains also have a transitive relation of strength between them. A domain D_1 can be stronger (or equal), $\sqsubseteq$, than another domain D_2 if it maps all variables in $x_i \in V$ to a (non-strict) subset of the mapping of D_2. Or more formally $D_1 \sqsubseteq D_2 \iff \forall x_i \in V(D_1(x_i) \subseteq D_2(x_i))$.

A constraint $c_i \in C$ will now be modelled with two properties, the affected variables $V(c_i)$ and the set of propagators $F(c_i)$. A propagator $f \in F$ is a function that maps a domain D_j to a stronger domain D_k. Propagators are not guaranteed to be idempotent functions, idempotent: $f(f(D_j)) = f(D_j)$, and may therefore be applied multiple times in succession. This application process can continue as long as the repeated application of the propagator results in a strictly stronger domain. Once the propagator fails to do so and $f(D_j) = D_j$ it has reached a fixpoint and the repeated application should be terminated. A propagator can therefore be described as a function that dictates how the domain of a variable should change according to a constraint and the domains of other variables.

As a blind repeated application of all propagators is potentially restrictively expensive a CP engine applies two techniques to apply propagators in an efficient manner. The first technique keeps an overview of which propagators are relevant, potentially not at a fixed point given the current domain, and secondly it can choose to implement an ordering of its propagators [22]. To track which propagators are at fixpoint the propagator itself can expose specific changes in the domain to watch for, such as an upper-bound change for a specific variable x_i, or a more general approach of exposing all of the variables this propagator considers relevant. The latter method runs the risk of potentially queuing up propagators which are at fixpoint but requires less complex propagators. The ordering by which propagators are checked is relevant to attaining a speed-up. In general fast propagators are applied before those which have a longer run time [22]. The granularity by which a solver chooses to rank its propagators is a hyper-parameter often defined at the implementational level.

A key component of CP solvers are the propagators. Where SAT in essence relies on the unit clause rule and a decomposition of the finite domain. A CP solver can implement more sophisticated propagators and utilize multiple propagators with different strengths. As propagators are functions that work in isolation of each other we can model a constraint multiple times from different perspectives [17]. This introduces different propagators to our model allowing for more possibilities in the inference process, quick but weak or strong but slow. Alternatively a constraint can itself also be modelled by different propagators without the need of introducing redundancy into a model. This can be possible with global constraints where multiple propagators can exist and can both be utilized in the same model or the solver can be programmed to only utilize a single kind of propagator.

Another strong element of a CP solver is the addition of programmable search. A SAT engine technically only ever sees boolean literals and clauses. As finite domain problems are decomposed it is difficult to explain to the SAT solver in which order we think it should search through the search space. SAT solvers do have access to the powerful VSIDS [18] heuristic which tracks the appearance of variables in recently learned clauses. However, in finite domain problems certain search orderings can be better than VSIDS. As a CP engine does have a proper view over the domain of each variable instead of a ”blind” boolean decomposition it can choose which values it wishes to select based on a pre-determined search ordering which is known as programmed search. Such an ordering may state to try out the lowest value in a variables domain first and/or to select a variable with the smallest domain [13].

3.3. LCG

Lazy Clause Generation, LCG, solving is a solving paradigm created by Ohrimenko, Stuckey, and Codish [19] and later re-engineered by Feydy and Stuckey [12] that combines the power of SAT and CP solving. It does so by utilizing a CP engine that has access to the flexible propagators with powerful inference techniques and combine it with a SAT engine that can apply a CDCL procedure to procure no goods. The LCG paradigm is competitive when it comes to finite domain solving and is state of the art for multiple problems [6, 12, 23, 24]. We now explain what goes into an LCG solver.

The initial version of LCG solving as by Ohrimenko, Stuckey, and Codish [19] approaches the paradigm by utilizing a SAT engine as the master solver and introduces CP propagators as lazy clause generators. As mentioned in subsection 3.1.2 the SAT paradigm requires a full decomposition of a constraint in order to solve the problem. The original version of LCG instead only decomposes the variables to the SAT domain. Once the SAT engine reaches a fixpoint with the unit clause rule [10] it goes through all possibly affected propagators. If a propagator is capable of propagating it presents an explanation clause to the SAT engine. This is a unit clause encoding the relevant propagation. This clause is then added to the database of the SAT engine. Such an explanation could be the specific clause of the SAT decomposition which would unit propagate at this point in time. A later re-evaluation showed that the approach has the potential of performing but could be slower in specific instances [20].

Current state of the art solvers turn this concept around and utilize a re-engineered version where a CP engine is the main engine that gets access to a SAT engine for unit propagation and CDCL procedures [12]. In this approach a CP engine solves the problem as it would normally but with three changes. First off, the propagators are expanded upon to provide explanations similar to before, when they propagate those explanations are provided to the SAT engine. Secondly, once propagation has occurred the SAT engine is scheduled for propagation. How and when this scheduling occurs is an implementation detail but the original model schedules it at the highest priority [12]. Thirdly, upon a conflict the SAT engine is triggered for conflict analysis saving a learned clause to its clause database and reporting the back-jump level. The SAT engine can therefore be seen as a kind of memoization propagator, keeping track of previous conflicts.

The protocol allowing communication between the CP and SAT engine consists of two literals. The assignment literal $[\![x = n]\!]$ and the bounds literal $[\![x \leqslant n]\!]$. The assignment literal denotes a variable, x, being assigned a value, n. The bounds literal denotes either a lower or upper bound which is encoded in the polarity of the literal. While the assignment literal would be adequate to model any explanation the bounds literal is often a more efficient representation with larger domains. These literals are the standard LCG language used to post explanations to the SAT engine. The domains of both engines is kept consistent more directly via channeling constraints. These are constraints that are enforced between the engines e.g. $x = n \iff [\![x = n]\!]$ and $[\![x \leqslant n]\!] \iff x \leqslant n$.

Utilizing this language of literals the CP and SAT engine communicate their findings with each other. Upon propagation taking place an explanation may be posted e.g. $\neg[\![x_1 = 3]\!] \wedge [\![x_2 \leqslant 2]\!] \rightarrow [\![x_3 = 4]\!]$ from the CP engine to the SAT engine. The appropriate change in the domain of the SAT engine will be made and the explanation stored as the antecedent in the SAT database. In the opposite direction the SAT engine will post literals and their polarity back to the CP engine e.g. $[\![x_4 \leqslant 8]\!] \leftarrow \top$ that it found using unit propagation. This will in turn enforce the channelling constraint which would either apply the information on the domain directly or schedule a propagator within the CP engine to apply it. The latter approach is relevant for chapter 4 as more complex literals can channel properties that need to be checked more than once.

To create effective explanations we require some engineering to make sure only relevant literals are used. Without engineering we utilize the generic explanation which offers the entire state of the domain as its reason for propagation. Whilst this is not incorrect it is too specific and captures variables irrelevant to the propagation. Typically only a subset of the domain is responsible for the propagation. We therefore engineer our propagators to only create explanations with literals directly related to the propagation. e.g. a *linear inequality* $\sum x_i \leqslant n$ constraint may only use lower bounds in its explanation when propagating as the upper bound has no information relevant to the propagation. We can test whether our engineered explanation is valid by resetting the entire domain and only assigning the literals in the explanation to true. Querying the propagator again should yield the same result.

As the SAT engine applies the CDCL procedure it has no concept of relations between the literals in the eventual learned clause. This means that we may have semantic duplicates which hinders performance. e.g. $[\![x \geqslant 2]\!]$ and $[\![x \geqslant 3]\!]$ may both be present in the learned clause, as they are separate literals in the encoding our SAT engine does not detect them as duplicates. To remove such semantic duplicates we utilize semantic minimization. Semantic minimization works by applying a set of predefined semantic rules where $[\![l]\!] \wedge \dots \rightarrow [\![r]\!]$ with r set for removal. One typical example of a semantic minimization rule is to remove literals that do not encode the strongest bound, e.g. in our example from before $[\![x \geqslant 3]\!] \rightarrow [\![x \geqslant 2]\!]$ and hence we can drop $[\![x \geqslant 2]\!]$. Any other semantic rules may also be introduced as long as they are sound given the meaning of the literal. In some cases we can also find such rules during the solving procedure, this is known as recursive minimization where we attempt to find possible $[\![l]\!] \wedge \dots \rightarrow [\![r]\!]$ rules given the current trace to see if a subset of our nogood $[\![l]\!] \wedge \dots$ is capable of fully implying $[\![r]\!]$ in which case we can also remove $[\![r]\!]$.

Since we have access to both the SAT and CP engine we can choose which type of search we want to utilize. For certain problems an efficient search pattern may not be known. In this scenario we can rely on the VSIDS heuristic from the SAT engine which can be performant when dealing with finite domain problems in the context of LCG [12]. If an efficient search pattern is known we can utilize the CP engine to execute a programmable search in accordance. It is also possible to interweave search strategies to potentially improve performance even more. As the LCG paradigm has access to both methods of search we expand our range of possibilities allowing for more potential improvements without having to change our solving paradigm between CP and SAT.

3.4. Cumulative and RCPSP

One problem that is prevalent within the solving paradigm is that of scheduling. Multiple variations of scheduling exist but for this thesis we will concern ourselves with the Resource Constrained Projects Scheduling Problem, RCPSP, as it is typically present as a sub-problem of other scheduling problems. An RCPSP problem is typically defined by a series of tasks. A task i has four properties, a domain of possible start times s_i, a duration d_i, a resource consumption r_i, and a series of precedent tasks $p_j \in P_i$, tasks that must be executed before this task can be executed, some work may also utilize c_i to denote the domain of completion times modelled as $c_i \leftarrow s_i + d_i$. The problem itself is then accompanied by a total available capacity c. Variations of this problem exist with MRCPSP allowing decisions to be made in s_i, d_i, and r_i but in a typical RCPSP problem only s_i is a decision variable which is what will be assumed for the remained of this thesis. To solve the problem tasks are assigned starting times such that the capacity is never exceeded and the precedence constraints are upheld. The goal is then to create a solution with the lowest makespan, the shortest time from when the first task is started to when the last task finishes.

To quickly denote such a problem the *cumulative* constraint was introduced. In its most well known form cumulative accepts three arrays and a single value. Each array is of size n denoting the number of tasks within the problem. Cumulative is then generally applied as follows:*cumulative*(S, D, R, c), with S being the array of starting variables s_i, D the array of fixed duration values d_i, R the array of fixed resource consumption values r_i, and the capacity c. Note that it does not handle the precedence constraints. The precedence constraints are modelled directly by linear inequalities $\forall p_j \in P_i(s_j + d_j \leqslant s_i)$. The cumulative constraint then enforces that no combination of tasks ever overflows the capacity for any time point t as denoted by Equation 3.3. Note that within the sum there is a boolean check to determine if a task is currently active and adds the resource requirement to the sum if this is the case.

$$\forall t \in [0 \ldots t_{max}] : \qquad (\sum_{i=0}^{n} (s[i] \leqslant t < s[i] + d[i]) \cdot r[i]) \leqslant c \tag{3.3}$$

3.4.1. Time-Resource Decomposition

There exist two decompositions of cumulative, the first decomposition we will look at is known as Time-Resource decomposition [25]. This decomposition flows naturally from Equation 3.3. It creates a linear inequality for every value of t up to the planning horizon t_{max}. This inequality then sums up the resource requirement of all of the active tasks and enforces that this does not exceed the capacity c. The Zinc decomposition is defined as below.

```
predicate cumulative(list of var int: s, list of var int: d,
                     list of var int: r,         var int: c) =
    let {set of int: tasks = index set(s),
         set of int: times = min([lb(s[i]) | i in tasks]) ..
                             max([ub(s[i]) + ub(d[i]) - 1 | i in tasks]) }
    in forall( t in times ) (
        c >= sum( i in tasks ) (
            bool2int( s[i] <= t /\ t < s[i] + d[i] ) * r[i]));
```

In order to help speed up the evaluation of this constraint we can explicitly introduce a new boolean literal B_{it} that denotes whether or not a task i is active at time t. This literal is defined previously by Schutt et al. [25] and defined as shown in Equation 3.4.

$$\begin{aligned} \forall t \in [0 \ldots t_{max}], \forall i \in [1 \ldots n] : \qquad & B_{it} \iff [\![s[i] \leqslant t]\!] \wedge \neg[\![s[i] \leqslant t - d[i]]\!] \\ \forall t \in [0 \ldots t_{max}] : \qquad & \sum_{i=0}^{n} B_{it} \cdot r[i] \leqslant c \end{aligned} \tag{3.4}$$

To expand this model Schutt et al. [25] states that support for holes in the domain of s_i can be integrated by $[\![s[i] = t]\!] \rightarrow \wedge_{t \leqslant t' < t + d[i]} B_{it'}$. But that this fails to perform on a practical level.

3.4.2. Task-Resource Decomposition

In the case of highly disjunctive problems or problems with large planning horizons one may prefer the task resource decomposition. The task resource decomposition functions under the assumption that not every time point needs to be checked but that only checking the edges of every task for an overflow is sufficient, it is therefore agnostic to the planning horizon t_{max}. The Zinc decomposition is defined as below.

```
predicate cumulative(list of var int: s, list of var int: d,
                     list of var int: r,        var int: c) =
    let { set of int: tasks = index set(s) }
    in forall( j in tasks ) (
        c >= r[j] + sum( i in tasks where i != j ) (
            bool2int( s[i] <= s[j] /\ s[j] < s[i] + d[i] ) * r[i]));
```

Similar to before there are some boolean variables that can be introduced to help speed up the evaluation. Schutt et al. [25] introduces the literals $B^1_{ij}, B^2_{ij}, B_{ij}$. B^1_{ij} denotes task j starting at or after task i starts and B^2_{ij} denotes that task j starts before task i ends. B_{ij} retains a similar definition from before but with a slight alteration being defined as task j starting during the execution of task i. The decomposition is then defined as per the constraints in Equation 3.5 [25]:

$$\begin{aligned} \forall j \in [1 \dots n], \forall_i \in [1 \dots n] - \{j\} : \quad & B_{ij} \iff B^1_{ij} \wedge B^2_{ij} \\ & B^1_{ij} \iff s[i] \leqslant s[j] \\ & B^2_{ij} \iff s[j] < s[i] + d[i] \\ \forall j \in [1 \dots n] : \quad & \sum_{i \in [1 \dots n] - \{j\}} r[i] \cdot B_{ij} \leqslant c - r[j] \end{aligned} \tag{3.5}$$

As the SAT solver does not know that B^1 and B^2 are in any type of relation we can add a series of redundant constraints to help the SAT solver understand this relation. These redundant constraints are defined as per Equation 3.6 [25]. Note that these constraints are created for all $i, j \in [1 \dots n]$ as long as $i < j$ to prevent duplicate constraints.

$$B^1_{ij} \vee B^2_{ij} \qquad B^1_{ji} \vee B^2_{ji} \qquad B^1_i j \vee B^1_{ji} \qquad B^1_{ij} \rightarrow B^2_{ji} \qquad B^1_{ji} \rightarrow B^2_{ij} \tag{3.6}$$

In general the time decomposition approach is more performant with cumulative problems and the task decomposition fares better with disjunctive problems [25].

3.4.3. Linear Inequalities

These decompositions rely on linear inequalities such as the $c \geqslant sum()$. These linear inequalities are modelled via the $int\_lin\_le(x, w, c)$ constraint. Here x resembles a series of input variables x_i. w is a series of equal size to x and represents the weight w_i of each variable. Finally c represents the capacity of the constraint. This then upholds constraints of the type $\sum_{i=0}^{n} w_i x_i \leqslant c$. Note that other constraints such as $wx \geqslant c$ and $wx = c$ can be represented by properly negating the constraint for the other inequality or by enclosing a value by $wx \geqslant c \wedge wx \leqslant c$ to obtain equality. From a solver perspective we can abstract the wx behind an interface and therefore generally consider x to already be a weighted variable. Therefore we will consider the constraint as $int\_lin\_le(x, c)$ for the remainder of this work.

One method of modelling the linear inequality, and the one this work will compare against, is a generically explained upper bound validity check. We assume $lhs = \sum_{i=0}^{n} x_i$, and the functions $lb()$ and $ub()$ to return the upper and lower bound of our variables. For every variable we then propagate on the following condition for all $x_i \in x$ that $ub(x_i) = min(ub(x_i), c - lb(lhs) + lb(x_i))$. The efficiency of this propagation can be improved by incrementally keeping track of the $lb(lhs)$ value to avoid re-computation as long as the bookkeeping costs are less than those of maintaining the incrementality.

Due to the inclusion of all variables $x_i \in x$, bar the variable we propagate on, in the decision making process this also requires all variables to be present in the explanation of the propagation. Creating an explanation of $O(n)$ size where $expl(x_i) = \prod_{j=0}^{n} [\![x_j \geqslant lb(x_j)]\!] : i \neq j \rightarrow [\![x_i \leqslant new\_ub]\!]$. These explanations can be optimized by dropping the redundant clauses where $lb(x_j)$ is equal to the lower bound defined at root. No extra propagations can occur on the lower bound of x_i as we cannot infer anything on the bounds of the lhs value apart from the $lhs = \sum_{i=0}^{n} x_i$ definition.

3.4.4. Timetable

Apart from utilizing a decomposition it is also possible to use the global cumulative constraint directly. A global constraint typically retains a more complete view of the problem than its decomposition counterpart. They can therefore incorporate more sophisticated techniques that allow for problem specific reasoning. For cumulative there are two main approaches to this, timetable filtering or energetic reasoning. Energetic reasoning is generally very performant on highly cumulative instances. However if this is not the case timetable filtering is the better propagator [24]. The remainder of this section will go into timetable filtering.

Timetabling reasons with compulsory parts. A compulsory part is the make up of all time points t at which a task i must be running, similar to the B_{it} literal from subsection 3.4.1. This compulsory section is defined at the bounds for a task i at $[max(s_i), min(s_i) + d_i]$ given that $max(s_i) \leqslant min(s_i) + d_i$ otherwise a compulsory section does not exist. Utilizing the compulsory parts the timetable constraint will build a timetable to determine compulsory resource consumption as per example 2. With this schedule of mandatory consumption the propagator will now view if these compulsory parts may prohibit any other task j from running within that time-frame. If this is the case the propagator will adjust the starting time $s_j + d_j \leqslant s_i \vee s_j \geqslant s_i + d_i$ depending on where the compulsory section lies. The timetable propagator is not idempotent and has the same propagational strength as the time decomposition [24], instead it offers a different means of explaining propagations. Techniques such as building the timetable incrementally can help speed up evaluations.

Example 2: Compulsory components

We start of by creating a task A of $d_A = 5$ and $s_A = [0, 1, 2]$. In order to determine the compulsory section we test the domain of every permutation of A. A_0, A_1, A_2 represent those permutations at their respective start times. We then view the timetable and see at what time points those permutations all overlap. In this case that occurs from $t = 2$ until $t = 5$. We thus have a compulsory section A_c from $[2 \ldots 5]$. Alternatively this can be calculated by taking the latest possible starting time of A and the earliest possible completion time of A and taking the interval. This is illustrated in figure 3.1.

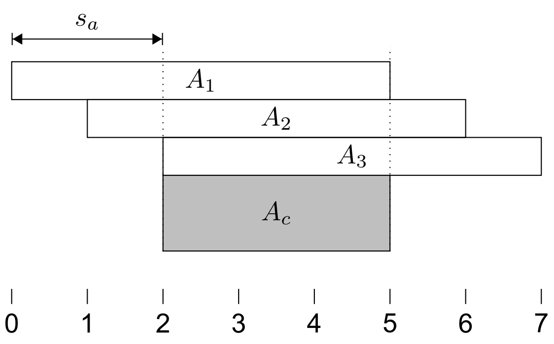


Figure 3.1: Task A has a potential start time of $s_a = [0, 1, 2]$ and a duration of 5. The compulsory section of A is marked in gray.

To properly harness the strength of LCG solving with this propagator an explanation should be posted when any propagation occurs. For this we follow the work of Schutt et al. [24] and utilize their definitions. We define a *profile* to be a tuple $D_i = (A_i, B_i, C_i)$. A denotes a time interval $[s \ldots e - 1]$ over which our compulsory section persists. B is the set of all tasks i that have a compulsory part within A, $ub(s[i]) \leqslant s \wedge lb(s[i]) + d[i] \geqslant e$. Finally C is defined as the sum of all the resources r_i of all tasks $i \in B$. A profile must have a maximal time interval A with respect to B and C. This means there is no alternative construction of A such that B and C remain identical, it covers the entire compulsory section. To achieve this we state there does not exist an profile $([s' \ldots e' - 1], B, C)$ where $s' = e$ or $e' = s$.

There are three ways in which a timetable propagator may explain its propagations: naive, big-step, or pointwise [24]. These explanations do not influence the propagational strength of the propagator, all variations construct the same timetable, but focuses solely on providing general explanations for the LCG procedure. The naive explanation is only slightly better than the general explanation. It looks at a profile D_i, considers the bounds of all tasks involved in B_i, adds the bounds of task j and then posts the explanation. Big-step integrates knowledge of the profile into the explanation. Whereas with naive we only utilize the bounds of individual tasks big-step utilizes the bound of the profile. A big-step explanation explains that every task has a compulsory section in A_i. This makes it more general than naive as naive may also explain compulsory sections outside of A_i which is irrelevant information. Their explanations are defined as per equations (3.7a) and (3.7b) [24].

$$\left([\![lb(s[j]) \leqslant s[j]]\!] \wedge \bigwedge_{1 \leqslant i \leqslant p, l \in B_i} [\![lb(s[l]) \leqslant s[l]]\!] \wedge [\![s[l] \leqslant ub(s[l])]\!] \right) \rightarrow [\![LB[j] \leqslant s[j]]\!] \tag{3.7a}$$

$$\left([\![s_1 + 1 - d[j] \leqslant s[j]]\!] \wedge \bigwedge_{1 \leqslant i \leqslant p, l \in B_i} [\![e_i - d[l] \leqslant s[l]]\!] \wedge [\![s[l] \leqslant s_i]\!] \right) \rightarrow [\![LB[j] \leqslant s[j]]\!] \tag{3.7b}$$

$$\left([\![t_l + 1 - d[j] \leqslant s[j]]\!] \wedge \bigwedge_{k \in B_i} [\![t_l + 1 - d[k] \leqslant s[k]]\!] \wedge [\![s[k] \leqslant t_l]\!] \right) \rightarrow [\![t_l + 1 \leqslant s[j]]\!] \tag{3.7c}$$

The most performant explanation is the point-wise explanation [24] defined as equation (3.7c). The pointwise explanation creates an explanation at relevant time points similar to time decomposition. The difference lies in how those time points are chosen instead of iterating over every time point of the profile we can take a more efficient approach. let $[t_1, \ldots, t_m]$ be a set of time points such that $t_0 = lb(s[j])$ and $t_m + 1 = LB[j]$. Then $\forall_j 1 \leqslant j \leqslant m : t_{j-1} + d[j] \geqslant t_j$, stating that the time points increment with step size smaller than or equal to $d[j]$. Another condition is that there exists a mapping of time points to profiles $P(t_l)$ such that $\forall_l 1 \leqslant l \leqslant m : s_{P(t_l)} \leqslant t_l < e_{P(t_l)}$, making sure there is a conflict profile to utilize. Then a pointwise explanation is constructed for every time point $t_l | 1 \leqslant l \leqslant m$. The goal is to now increment t_l by the largest amount of time whilst not violating those conditions. This results in a chain of explanations, visualized in Figure 3.2, where t_1 flows from the current configuration of the problem and a time step is reliant on the explanation of the time-step before it to assure the $[\![t_l + 1 - d[j] \leqslant s[j]]\!]$ literal holds. In a way we can therefore state that $t_j \rightarrow t_{j+1}$. There is one small possible exception to the step size. If the possibility arises where a profile falls in between $[t_j + 1 \ldots t_{j+1} - 1]$ due to a large enough d_j resulting in the profile being skipped another time point at the end of the profile $t_i = e_i - 1$ may be introduced as this helps with performance [24].

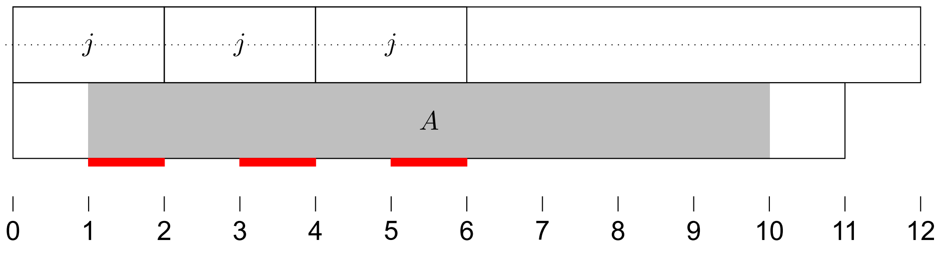


Figure 3.2: Task A has a compulsory grey section and task j now exceeds the capacity of the dotted line. Red markers visualize the time points t_j that must be explained. Upon propagation task j is shifted right to $t_j + 1$. The first three time points during this process are visualized.

4 Related Work

Let us now discuss two approaches that have been researched with the goal of speeding up the solving of finite domain problems within the paradigm of LCG. We show that work can by-pass structural extended resolution to achieve performance and we show what work has been done on structural extended resolution, which is typically associated with speeding up a single global constraint per literal.

4.1. Decomposition; Relying Purely On SAT

Generating the explanations may not be useful at all. An LCG solver does not directly fully decompose its CP constraints to the SAT engine. Yet, it can still achieve a full decomposition through an unfortunate search process that triggers the propagator to explain almost every possible permutation of variable assignments. One way of mitigating this is by partially decomposing the relevant constraints into their SAT encoding directly [1] This prevents the usage of the more expensive CP propagator that needs to generate these explanations every time at the cost of a larger SAT model. This work was later expanded upon to show that partial decompositions of a constraint are not required and that it is more advantageous to fully decompose the active constraints allowing the lazy decomposition approach to be utilized for constraints for which partial decompositions are unknown [3].

The strength of the CDCL procedure may also be increased. As mentioned in subsection 3.1.1 current CDCL solvers are just as strong as resolution. Resolution can be generalized to allow for more compact proofs. One of these generalizations is called extended resolution [27]. Utilizing this stronger form of resolution a SAT solver was adapted to apply extended resolution [4, 14] with varying levels of success. Audemard, Katsirelos, and Simon [4] shows that implementing a version of ER with restrictions to local search does lead to an overall improvement. While the unrestricted version of Huang [14] seems to perform worse on various benchmarks relative to its non extended version. The most important takeaway from both these works is that extended resolution can have a significant effect on the runtime of hard instances. With multiple open instances being closed [14] and beating non extended solvers with orders of magnitude [4].

4.2. Structural Extended Resolution

An alternate method is to extend the communication protocol between the CP and SAT engine with new literals. This concept is known as structural extended resolution.

New literals should not be introduced without reason. First and foremost is the overhead associated with the new literal. While the new literal may reduce a proofs size significantly if the channelling constraint is too expensive all speed-ups from the shorter proof will be lost to the incurred cost of computation to apply this shorter proof. Another framework of relevancy is that of generality [7]. A more general explanation helps in the creation of more general and therefore stronger nogoods. A stronger nogood is desirable as it prunes a larger part of the search tree. Given a language of resolution used by the LCG solver it may already be possible to construct the best possible explanation. Chu and Stuckey [7] define a framework for assessing the generality of an explanation and define two properties. An explanation is maximally general if given a fixed language no strictly more general explanation may be constructed. An explanation is universally maximally general if the explanation is maximally general with respect to the universal language consisting of all possible logical expressions [7].

Reasoning that linear constraints are by far the most common constraint within models Chu and Stuckey [7] introduce a new partial sum literal. Using a linear inequality like $x_1 + 2x_2 + 3x_3 + 4x_4 \leqslant 30$ as their example. Given that $x_1 \geqslant 1, x_2 \geqslant 2, x_3 \geqslant 3$ we are able to infer that $x_4 \leqslant 4$. For this inference there exist multiple maximally general explanations utilizing the standard LCG language e.g. $[\![x_2 \geqslant 1]\!] \wedge [\![x_3 \geqslant 3]\!] \rightarrow [\![x_4 \leqslant 4]\!]$ relying on the bound literals. However, none of those explanations are universally maximally general e.g. the proposed explanation fails to capture the information relating to x_1 and only considers x_2 and x_3 relevant. To this extend they introduce the partial sum literal $[\![x_1 + 2x_2 + 3x_3 \geqslant 11]\!]$ which is universally maximally general.

Creating these literals can be expensive as the amount of permutations grows exponentially with respect to the number of variables in the sum. In order to properly manage the number of literals Chu and Stuckey [7] restrict the creation of these literals two fold. First all variables in the literal must follow a certain ordering that does not have holes in it e.g. any valid literal should be definable as $[\![\sum_i^k a_i x_i \geqslant k]\!]$ and a literal such as $[\![a_1 x_1 + a_3 x_3 \geqslant 3]\!]$ would be invalid as $i = 2$ is missing. The orderings of Chu and Stuckey [7] are constructed manually by minimizing the width of the search order, although it is unclear how, but argue such a process can be approximated. Secondly, only intervals of a predefined multiplicity will be generated. e.g. utilizing a multiplicity of 3 would result in the literals $[\![\sum_{i=1}^3 a_i x_i]\!]$, $[\![\sum_{i=1}^6 a_i x_i]\!]$ being eligible for creation. In their experimental analysis Chu and Stuckey [7] show that utilizing their structured ordering as well as a multiplicity of 5 are performant properties of this new literal.

$[\![x_i > y_i]\!]$ and $[\![i \ll j]\!]$ are ordering literals introduced to help with the *lex_less* and *disjunctive* constraints respectively [7]. The $[\![x_i > y_i]\!]$ literal and its non-strict form were implemented to extend the *lex_less* constraint allowing for a direct channel between two variables. The disjunctive literal $[\![i \ll j]\!]$ states that task i should finish before task j starts. A decomposed view of this literal would look as follows $[\![s_i + d_i \leqslant s_j]\!]$ which is similar to the *lex_less* literal but now with an offset to the channel. Utilizing these extensions and the aforementioned linear constraint an increase in performance for the BIBD (linear, lex_less) and jobshop (disjunctive) problems was found [7]. Recent work also sees a re-appearance of the disjunctive literal now known as the reified difference constraint [6]. Leveraging this literal together with a novel representation of signed variables they manage to outperform multiple state of the art solvers when it comes to disjunctive scheduling [6]. Due to the intricacies of their implementation the ablation study deems it impossible to remove the effect of the implicit literals.

Another type of literal tracks a partial state of a system. In order to speed-up the *table* constraint Chu and Stuckey [7] utilize a literal to track which tuples have been picked. This literal, $[\![x_1 = 1 \wedge \cdots \wedge x_i = n]\!]$, is then presented in the negated form stating that this literal has not been picked and that through a combination of these negations certain values can be removed from a variable its domain. This reasoning is then additionally applied to the *regular* and BDD / MDD structures to track intermediate states of automata and taken paths in decision diagrams. Utilizing this literal type a speed-up for the nonogram problem was achieved [7].

The work of Schutt and Stuckey [23] also introduces a variation of the reified difference literal, now with a difference instead of an addition, $[\![x - y \leqslant d]\!]$ in the problem of producer consumer scheduling. This is a type of problem in which resources can be replenished and consumed from some reservoir over time. They utilize this literal to simplify the classification process required for the propagation as introduced by Laborie [16]. Claiming that these literals are also more general for the classification channels making it possible a classification occurs sooner in the propagator. The literals re-appear in the explanation for the order propagator, a type of propagator capable of dealing with reservoir constraints. It is then shown that the order propagator is already quite performant and once teamed up with the difference propagator the combination outperforms all others by an order of magnitude [23].

[7] states that language extensions do exist for other global constraints but that any extensions they could think of would most likely be impractical due to the overhead from the channelling constraints. Yet no mention of which extensions and what relevant channelling constraints would be classified as such is made clear.

5 Main Contributions

In this chapter we propose two structurally extended resolution techniques. The first technique has two variations and considers the linear inequality constraint. We propose the addition of creating extra predicates to model partial sums according to either the linear sequential encoding or the tree like totalizer encoding. The details of which are found in section 5.1. The other technique creates a new literal specific to the timetable propagator modelling task precedence as $[\![i \gg j, kr]\!]$ stating task i must occur after the compulsory section created by tasks j, k, r found by the timetable propagator. This technique is discussed in section 5.2.

5.1. Extended Linear Inequalities

For the linear inequalities we propose the addition of new auxiliary variables with a similar approach to that off Chu and Stuckey [7]. However, instead of creating a large amount of literals in the SAT encoding explicitly this work maps the partial sums directly to finite domain variables instead. These variables are then channelled accordingly to enforce consistency. This section will describe the reason for this approach, introduce the sequential and totalizer encoding, and determine possible trade-offs.

There are multiple ways of keeping this encoding consistent. In this thesis we put the responsibility of consistency within the int_lin_le propagator, as it was the most accessible approach. This keeps the responsibility in one place, avoids having to overhaul the CDCL procedure of Pumpkin, and is able to directly rely on the lazy predicate engine. One can also chose to model the structure with simple (in)-equality constraints in alternative locations. These locations may consist of but are not limited to taking place within the modelling language resulting in solver agnostic decompositions, by decomposing the constraint directly in the solver, or utilizing directly channelled literals [7] instead of mapping sums to intermediate variables.

We propose two encodings for this structural extended resolution. The sequential and totalizer encodings. The sequential approach is identical to previous work [7]. Where the partial sums are defined as $p_i = \sum_{n=0}^{i} x_n$. This definition can be extended with the concept of multiplicities, increasing the step size of the variables covered by p_i by m as follows $p_i = \sum_{n=0}^{im} x_n$. This definition is then further extended in a recursive manner such that a partial sum depends on the previous partial sum instead of all underlying variables $p_i = p_{i-1} + \sum_{n=(i-1)m}^{im} x_n$. This results in small areas within the encoding in which consistency is enforced. Note that we therefore no longer force consistency over all the variables directly as enforcing consistency over the encoding in turn guarantees the variables are consistent with the constraint.

The totalizer encoding instead follows a tree structure with the variables as leaf nodes. This work considers a left filled tree where the inner nodes of the tree represent partials sums. The parameter b is the branching factor and describes how many child nodes a node may have. We consider p_0 to be the root node of such a tree. If we then create a $b = 2$ tree with sufficient variables to have at least three internal nodes we can define $p_0 = p_1 + p_2$. Extending this definition towards the leaves we end up with $p_1 = x_1 + x_2$. This then also results in small areas within the encoding in which the consistency is enforced.

There are three types of orderings we consider for these extensions. Input order, random, and scalar. The input order takes the variables as they are introduced in the model. The random approach shuffles the array before integrating the variables into the model and the scalar approach determines the weight of each variable in the linear inequality and sorts it on that. Chu and Stuckey [7] introduces problem and instance specific orderings of the variables. And while this has the potential to improve a solving run it requires domain specific knowledge. This is generally not available in typical general purpose solvers and is therefore not explored here.

The new variables introduce more general learning opportunities. There are two main reasons for this. First of all if a partial sum variable is part of a learned clause it covers a more general condition than if the boundaries of its children were used instead. Therefore this learned clause is capable of pruning a larger part of the search tree which can lead to a better search path. Secondly this approach allows us to learn about a forced lower bound. The default behaviour does not allow reasoning over the lower bounds of the variables x_i which makes sense as we only care that their upper bound does not violate the constraints capacity c. However, it is now possible for this concept to be introduced as we can learn about the forced lower bound of a partial sum through resolution. Therefore it is possible to reason about the lower bounds of the variables x_i leading to a potential improvement in propagation.

Out of the box this technique creates a lot of duplicate variables that can easily be removed. Pumpkin models a linear equality as two linear inequalities. e.g. $\sum x_i = k \rightarrow \sum x_i \leqslant k \wedge \sum x_i \geqslant k$. Without any additional bookkeeping this means an encoding may be constructed twice, once for the lower bound and once for the upper bound. Therefore we also add a simple check to our solver. We keep track off every partial sum we encounter and if we encounter the same partial sum we reuse the variable responsible for that partial sum.

The sections below will introduce their respective data structures and the propagations within them. For the sake of brevity the explanations are not shown. However, in order to create the explanation relevant to a propagation one can take every occurrence of a bound and replace it with the respective literal as follows. $lb(x) = [\![x \geqslant lb(x)]\!]$ and $ub(x) = [\![x \leqslant ub(x)]\!]$. Then replace every mathematical operator with a $\wedge$ to obtain the reason for the propagation. Note that the use of min and max operators is purely there to avoid accidental overrides of less strong bounds. They are part of the algorithmic procedure but their left hand components should never trigger an actual propagation and should be ignored when creating the explanation.

5.1.1. Sequential Encoding

The sequential encoding follows a linear structure of partial sums. These partial sums p_i are introduced every m variables. We define $p_i = p_{i-1} + \sum_{j=mi+1}^{m(i+1)} x_j$ with $p_0 = \sum_{j=1}^{m} x_j$. And then introduce the bound $p_n \leqslant c$ on the last partial sum to uphold the constraint. In order to more easily define our propagations we define the variables x_j that are directly relevant to a partial sum p_i to be in the locality set $x_j \in L(p_i) | j \in [im + 1 \dots (i + 1)m]$. Similarly we define the inverse $P(x_j) = p_i$ to give us the partial sum that is directly relevant for a variable x_j and a $prev(x_j) = p_{i-1}$ to access the partial sum directly before the directly relevant partial sum. This data structure is visualized in figure 5.1.

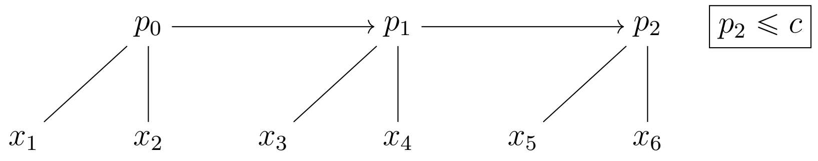


Figure 5.1: A sequential encoding with $m = 2$ for the constraint $\sum_{i=1}^{6} x_i \leqslant c$. Arrows help direct the definition that $p_1 = p_0 + x_3 + x_4$. We also define the locality set as $L(p_1) = \{x_3, x_4\}$

There are 6 propagations taking place in the sequential encoding to enforce consistency. These are defined in equation (5.1).

$$lb(p_i) = lb(p_{i-1}) + \sum_{x_j \in L(p_i)} lb(x_j) \tag{5.1a}$$

$$ub(p_i) = ub(p_{i-1}) + \sum_{x_j \in L(p_i)} ub(x_j) \tag{5.1b}$$

$$lb(p_i) = lb(p_{i+1}) - \sum_{x_j \in L(p_{i+1})} ub(x_j) \tag{5.1c}$$

$$ub(p_i) = ub(p_{i+1}) - \sum_{x_j \in L(p_{i+1})} lb(x_j) \tag{5.1d}$$

$$lb(x_j) = lb(P(x_j)) - ub(prev(x_j)) - \sum_{x_k \in L(P(x_j))-x_j} ub(x_k) \tag{5.1e}$$

$$ub(x_j) = ub(P(x_j)) - lb(prev(x_j)) - \sum_{x_k \in L(P(x_j))-x_j} lb(x_k) \tag{5.1f}$$

The propagations in equation (5.1a) and equation (5.1b) represent the bottoms up information flow, passing lower and upper bounds from the lower partial sums to the higher partial sums. Propagations in equation (5.1c) and equation (5.1d) represent the top down information flow with lower bounds being updated in the case of mandatory consumption and upper bounds being updated with respect to the remaining capacity in the system. Equation (5.1e) and equation (5.1f) resemble the consistency within a locality set. Updating the variables x_j with respect to the upper and lower bounds of the directly adjacent partial sums and other variables x within the locality set. At the bounds of the encodings only those propagations which are possible occur. e.g. we cannot apply equation (5.1c) at the right side of the encoding as there would not be a p_{i+1}. While not shown here we also add a dummy node to the left of the encoding to avoid out of bounds operations.

The explanations for the sequential encoding scale with size $O(m)$. As each propagation generally contains $m+1$ predicates in the explanation. This $m+1$ resembles either two predicates on the partial sums a_i and $m-1$ predicates on the variables x_i in the local scope with the -1 representing the skipping of the variable under propagation. Alternative the explanation consists of m predicates resembling the variables x_i in the local scope and one predicate resembling the bound on a partial sum a_i.

5.1.2. Totalizer Encoding

The totalizer encoding follows a tree of vertices $v_i \in V$. If v_i is not a leaf vertex then its children are denoted as $v_j \in C(v_i)$. If this vertex is not the root then its parent node is denoted as $P(v_i)$. The variables originally passed on into the constraint $x_i \in X$ represent the leaf vertices of the tree. We then construct a left filled b-ary tree until we arrive at a layer with only one vertex, the root. Any internal node v_i is then defined as $v_i = \sum C(v_i)$ representing a sum over all its child vertices. The root v_0 is then constrained such that $ub(v_0) \leqslant c$ respecting the constraint. This is visualized in figure 5.2.

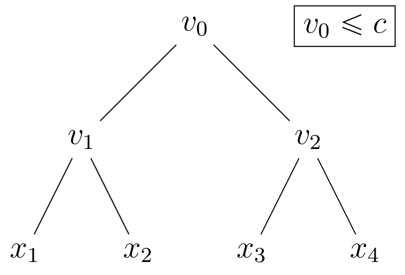


Figure 5.2: A totalizer encoding with $b = 2$ for the constraint $x_1 + x_2 + x_3 + x_4 \leqslant c$. Note that a parent is defined by its children e.g. $v_1 = x_1 + x_2$.

In order to keep this data structure consistent there are four propagations taking place within the propagator. These are shown in equation (5.2).

$$lb(v_i) = \sum_{v_j \in C(v_i)} lb(v_j) \tag{5.2a}$$

$$ub(v_i) = \sum_{v_j \in C(v_i)} ub(v_j) \tag{5.2b}$$

$$lb(v_i) = lb(P(v_i)) - \sum_{v_j \in C(P(v_i))-v_i} ub(v_j) \tag{5.2c}$$

$$ub(v_i) = ub(P(v_i)) - \sum_{v_j \in C(P(v_i))-v_i} lb(v_j) \tag{5.2d}$$

The propagations in equation (5.2a) and equation (5.2b) resemble the bottom up information flow. They look at the variables and put the realistic bounds on the tree, information flows up to the root where it may be possible to detect an inconsistency between the two bounds. Propagations in equation (5.2c) and equation (5.2d) resemble the top down information flow. Equation (5.2c) propagates the concept of mandatory consumption where a parent node has a lower bound set and the upper bounds of the neighbours of v_i are too low to possibly make up for this consumption resulting in extra consumption being required of v_i. Equation (5.2d) deals with capacity constraints updating the upper bound of v_i because its neighbours are already consuming a certain amount of the capacity. Note that leaf nodes do not propagate on rules a and b. And that the root node does not propagate on the rules c and d.

The explanations for the totalizer encoding scale with $O(b)$ as each propagation within the totalizer encoding contains exactly b predicates. Either resembling a bound on the parent node and $b-1$ predicates representing the bounds of the neighbours or of one predicate for each of the b child nodes when propagating on a parent node.

5.1.3. Theoretical Strengths

The cost of these techniques is configurable. Through the hyper parameters of m and b there is full control on how much you want the technique to be applied. Very large values means you lose the resolution strength but have much less channelling to perform. This means that as a user it is easy to adjust the technique according to the problem you are solving and create multiple runs in an ensemble structure. For most of the extended resolution techniques it is typically a boolean scenario, either with extended resolution or without but now more control can be retained.

The structural extension approach has the possibility of leading to smaller learned clauses. One of the main benefits is how the size of the explanations is bounded by our hyper parameters m and b as the size of the explanation is abstracted behind the more general variables introduced by the encoding. While is possible that in the worst case scenario these smaller explanations still unfold in such a way that they contain all the variables x_i in the explanation. It is also now possible for much smaller conflict explanations and learned clauses to exist that do not contain all those variables.

There is the potential for exponential pay-off. While a typical linear inequality conflict would be explained by $O(n)$ variables we are capable of consistently explaining the entire inequality with $O(m, b)$ variables, since m and b are fixed this means that we typically explain a linear inequality conflict with an $O(1)$ explanation size. This means that no matter the size of the inequality we can always explain the conflict in a set number of variables.

5.1.4. Theoretical Weaknesses

First off variable orderings are hard to come by. Typically a solver does not have any knowledge about what type of problem it is solving. Hence it can also not make many inferences as to what a good variable ordering could be. While a heuristic sorts such as a scalar sort may be beneficial it is not guaranteed to be fruitful [7]. This means that the technique may very well be reliant on the order by which the problem was modelled. This is an external factor that is hard to influence in some cases and may therefore make it difficult to apply the technique. A poor ordering results in the the entire encoding needing to be explained as it could occur that x_1 implies a change to x_10 meaning all variables in between would need to be decomposed if we resolve on that decision level. This results in an $O(n)$

explanation worst case for the sequential encoding and an $O(log(n))$ worst case explanation for the totalizer encoding.

Secondly is the potential risk of overlap. In section 3.3 we spoke briefly about semantic minimization, the practice of removing duplicate predicates from the nogood. However, it may be possible for two similar partial sums to enter the nogood e.g. $[\![x_1 + x_2 \leqslant 4]\!] \wedge [\![x_1 + x_3 \leqslant 4]\!]$ for which we have no minimization rule. Note that in the scenario where x_2, x_3 are fixed we now have two exact duplicates in the nogood. This means that the technique may suffer from duplication in certain scenarios to which the current implementation has no answer. One solution is to look at methods to linearly combine the two inequalities into a single predicate avoiding the potential duplication.

Finally the last downside to linear extended resolution is the large amount of bookkeeping required to keep the structural extension consistent. A single variable update now requires an additional $O(n)$ or $O(nlog(n))$ updates to the internal data structure for the sequential or totalizer encoding respectively, the distance to walk the entire data structure. It is possible for this propagation chain to terminate earlier due to an internal node having a stronger bound via nogood learning. And while this may perhaps introduce a lower amortized complexity it would have to be on an instance by instance basis.

5.2. The Cumulative Literal

This section will discuss the contribution of the cumulative literal. For the sake of brevity we only consider the lower bounds propagation, also known as a right shift. The upper bounds propagations can be achieved by scaling all variables usually involved in upper-bound propagation by -1 and applying the same technique as with lower bound propagation.

A cumulative literal $[\![i \gg j, k, r]\!]$ will represent the decision of having a task i start after a mandatory component created by the tasks j, k, l. When set to true this literal indicates that a task cannot overlap with the mandatory part of the tasks in the profile and must occur after it. When set to false the task must occur in part either before or during the mandatory part of the profile. The term profile will therefore be abused slightly. By definition [24] it requires that all tasks mentioned in the profile have mandatory components that all overlap between the bounds of the profile. However, due to the learning procedure we may learn that a task i must start after a profile of tasks j and k while those tasks may not have been fixed enough to have mandatory components. Alternatively we may state ”a task i must occur during the profile” where again we consider a profile to be potentially unfixed, in which case overlapping with it may be fully possible (j and k may be disjoint). A literal therefore more precisely represents starting the task i after or at the earliest completion time between j, k, r and the term profile will be adopted to suit this definition when applicable.

These literals will be set by channelling constraints and the timetable propagator. The literals on their own do not achieve much as they merely represent relations within the current state of the candidate solution. Being set via their channelling constraints only in the trivial cases when propagations are no longer possible. However, techniques exist which can detect these relations. One of those techniques is timetabling which is capable of detecting mandatory parts and then specifying which tasks need to shift to be properly accommodated. The timetabling algorithm therefore gets a new role. It is now responsible for detecting these relations and setting the appropriate literals to true.

The amount of cumulative literals created needs to be hampered. If we want to represent all possible combinations we get an upper bound of $O(n2^n)$. This is not feasible. However, we already denoted that our only source of relevant literals will be coming from the timetable propagator. Therefore the timetable propagator is given the additional task of creating these cumulative literals as it finds them. If more sources of cumulative literals are to be added they too must create the literals themselves before propagating on them. This does not introduce the risk of setting this literal at a too late decision level. The timetable propagator will always detect the literal if it is relevant to set. Re-iterating that in the trivial channel no propagations can be made due to the literal.

Timetabling explanations are highly relevant for the cumulative literal. As we are still within the LCG paradigm we now need to explain why we set the cumulative literal to true. For this we rely on the already existing explanations of naive, big step and pointwise as defined by Schutt et al. [24]. However unlike those explanations we go on a per profile basis. Explain the profile utilizing one of those techniques and then set the according literal. Note that in the case of pointwise utilizing only the first time-point suffices as it represents a valid point in the overlap from where we can deduce the literal.

5.2.1. Additional Propagators

Setting the literals is one thing but we now wish to also channel them properly. For this we introduce two new propagators that will function in a reified context. With a reified context we mean that if the literal is true then the propagators are active, if the propagators detect a conflict before they are activated the literal is set to false. Those two propagators are the greater or equal to minimum and less than minimum propagators, GEQMin and LTMin. GEQMin will be reified with the literal as is, representing the relation to hold, while LTMin will receive a negated variant of the literal, representing the relation to not be present.

GEQMin represents the typical timetable propagation. It looks at the profile, determines the minimum of all the possible earliest completion times, defined as ect_{min}, and shifts the start variable to be greater or equal to this minimum value, hence the name GEQMin. GEQMin has a slightly optimized explanation that states the lower bounds of all variables only need to be greater than the minimum instead of the lower bound per variable. The propagation can be viewed in equation (5.3a) and the explanation in 5.3b. We can trivially set the literal to false if shifting it to the right of the mandatory part is no longer possible. This happens when $lb(s_i) > ect_{min}$ and is explained with equation (5.3c)

$$ect_{min} = min(\forall_j \in P | lb(c_j)) \rightarrow lb(s_i) \tag{5.3a}$$

$$\bigwedge_{j \in p} [\![c_j \geqslant ect_{min}]\!] \rightarrow [\![s_i \geqslant ect_{min}]\!] \tag{5.3b}$$

$$[\![s_i \leqslant ub(s_i)]\!] \wedge \bigwedge_{j \in p} [\![c_j \geqslant ub(s_i) + 1]\!] \tag{5.3c}$$

To then trivially set the variable to true and channel the literal when it is set to negative we require the inverse of this propagator. LTMin therefore represents the concept of making a shift impossible. It looks at all the profiles and determines the minimum of the latest completion times, defined as lct_{min}, and shifts the start variable to be less than this value, for ease of use we call the task with the lct_{min} the restrictor denoted by r. The propagation can be viewed in equation (5.4a) and its explanation in equation (5.4b). As we are working with a negated literal this means we can set the original literal to true in the trivial case when the task has to occur shifted to the right of any mandatory component of the profile. This occurs when $lb(s_i) >= lct_{min}$ and is explained with equation (5.4c).

$$lct_{min} = min(\forall_j \in P | ub(c_j)) \rightarrow ub(s_i) \tag{5.4a}$$

$$[\![c_r \leqslant lct_{min}]\!] \rightarrow [\![s_i \leqslant lct_{min} - 1]\!] \tag{5.4b}$$

$$[\![c_r \leqslant ub(c_r)]\!] \wedge [\![s_i \geqslant lb(s_i)]\!] \wedge \tag{5.4c}$$

The LTMin propagator is required as not being able to do a right shift does not imply we will perform a left shift. A small theoretical mistake that was made at some point was to create a literal per task and profile but not per the polarity of the shift. The idea being that $l \rightarrow [\![s_i \gg \{c_j, c_k, c_r\}]\!]$ and $\neg l \rightarrow [\![c_i \ll \{s_j, s_k, s_r\}]\!]$ where the polarity of the literal immediately performs the shift. This removes the use for LTMin as now we have two GEQMin propagators which upon detecting inconsistency immediately set their complement to true. However, this approach fails on a theoretical level as when a right shift is no longer possible we do not have enough knowledge to state a task must now left shift. It could be that tasks in the profile are not yet fixed meaning we can overlap with tasks in the profile instead of shifting over them. If such a case exists the LTMin propagator will attempt to shrink the domain of s_i as quick as possible such that timetable can perform the left shift.

There is no direct link between two literals considering the same combination of profile, shifted task, and complement polarity. While setting one literal to true would implicate the other being set to false this was instead done via the consistency channel of the relevant propagators. The main downside to this is that explanations may be less general as once decomposed by 1UIP they are partly the explanation of the GEQMin and swapping the literal out would then add extra predicates from the underlying explanation type. The literals from GEQMin can either be removed by the semantic minimizer or worst case could introduce extra predicates if 1UIP wishes to decompose the constraint further. However, this is left as a future refinement. The important part is that both implementations arrive at the same fixpoint. This means that while one may lead to better learned no goods neither technique will hinder the learned no goods from propagating.

5.2.2. Improvements

When learned, a cumulative literal is capable of representing a much more powerful concept than an LCG solver can currently represent. In order to learn that a task must take place after a profile it can only be modelled by incremental steps. e.g. $[\![s_i \geqslant t]\!] \rightarrow [\![s_j \geqslant t + d_i]\!]$ which represents a shift over a single time-point. Requiring all time points to be explained before the relation is fully learned. The cumulative literal is much more general as it simply states that a task must occur to the right of the profile but has no specific requirement as to at what time-point it must occur.

Reasons for individual propagations are more general. Before this technique any shifting explanation had to re-explain that there was a mandatory part somewhere, including predicates describing the lower and upper bound of all involved tasks. The cumulative literal itself is the representation of a mandatory part but it abstracts away all of the upper bounds. A propagation over a start variable is now represented by the literal and just the lower-bounds of all tasks in the profile. As the literal can be set through learning it therefore covers a wider variation of explanations making it more general. It also slices the average explanation size for propagations in halve.

The literals allow for stronger conflict detection. The literals represent a relation between a variable and a profile. Initially, without the cumulative literal, we may learn that we do not want to shift to the right of a partially fixed profile, either we wish to overlap with parts of it or shift to the left of it. However, the only way to represent this is with incremental steps. Simply said, if the upper bound of the profile moves left, then the variable follows suit. However this makes no deductions over the lower bound of that variable. Timetabling then shifts the task to the right via the lower bound. Note that we are in a consistent state, however there is no feasible solution. With the cumulative literal this first has to be abstracted into the relation. Following the same steps as before we want to either overlap or shift to the left of the profile. We must first set the relation to false after which the variable is propagated on. Now when timetabling comes along it too must declare its intentions via the relation by setting it to true. However, as we have already set the literal to false this raises a conflict. This is further demonstrated in example 3.

Example 3: Stronger conflict detection

We create two tasks which have compulsory sections from 1 to 5 and 0 to 4 as visualized in figure 5.3. The tasks are not fully fixed yet and the compulsory sections may therefore grow to 7 and 6 respectively. The line s_i represents the current domain of s_i. In scenario one we learn that s_i cannot occur to the right of the profile, but as we do not have extended resolution the best conclusion we can therefore make is $[\![s_i \leqslant 6]\!]$. Then applying a timetable lower bound we know it must occur to the right of the profile but as we again do not have extended resolution we can only conclude $[\![s_i \geqslant 4]\!]$ represented in the domain of s'_i. In the case of extended resolution this is different. We again learn the we cannot occur to the right of the profile and have to set $[\![\neg l]\!]$, l is the cumulative literal responsible for this shift. This shrinks the domain again such that $[\![s_i \leqslant 6]\!]$. But now when timetable wants to enforce occurring to the right of the profile we must first set $[\![l]\!]$ However this means that we set both $[\![l]\!] \wedge [\![\neg l]\!]$ which is a conflict.

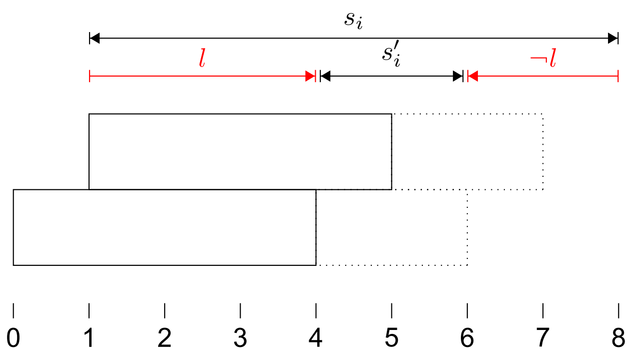


Figure 5.3: A possible timetable with a task i to be shifted while no feasible solution will exist. Hard outlines define mandatory components with dotted outlines defining the upper bounds of the tasks. s'_i represents the domain of s_i after propagation in a non extended resolution solver.

More expensive propagators can be bypassed. In our definition of propagators we stated that they are not idempotent, idempotent: $f(f(D_j)) = f(D_j)$, this means that when a propagator manages to propagate it will most likely be re-enqueued as variables relevant to itself were changed and it can make a second round of deductions. However, if such a propagator is expensive this is something we wish to avoid. So too is the case with the timetable propagator. While timetabling is generally regarded as a cheap propagator in comparison to alternatives such as edge-finding it is expensive when it comes to upholding the basic relations represented by the cumulative literal. A very optimized time table propagator may be able to detect a chain of shifts having to occur yet without engineering efforts may require multiple calls to propagate to enforce all of the shifts. As a shift is a cheap operation to perform once initially inferred we can instead utilize cheaper propagators to take care of it instead of the timetable propagator. The GEQMin propagator is that propagator, it is a very lightweight propagator capable of maintaining the relation of the cumulative literal preventing extra calls to the timetable propagator. A possible scenario is discussed in example 4

Example 4: Cheaper calls

We take the scenario of three disjoint tasks j, k, i each with duration $d = 4$. $s_j = [0 \dots 2]$ and $s_k = [4 \dots 6]$. s_i will then be bounded to be within $s_i = [8 \dots 12]$. We then update $s_j = [1 \dots 2]$. This in turn affects the domains of s_k and s_i. A solver may therefore require two calls to the timetable propagator. Once to update that $[\![s_j \geqslant 1]\!] \rightarrow [\![s_k \geqslant 5]\!]$ and then once more to propagate that $[\![s_k \geqslant 5]\!] \rightarrow [\![s_i \geqslant 9]\!]$. Instead these calls would be made separate to the GEQMin propagator meaning we avoid building two timetables for propagations we can resolve with a much cheaper propagator. The scenario is visualized in figure 5.4

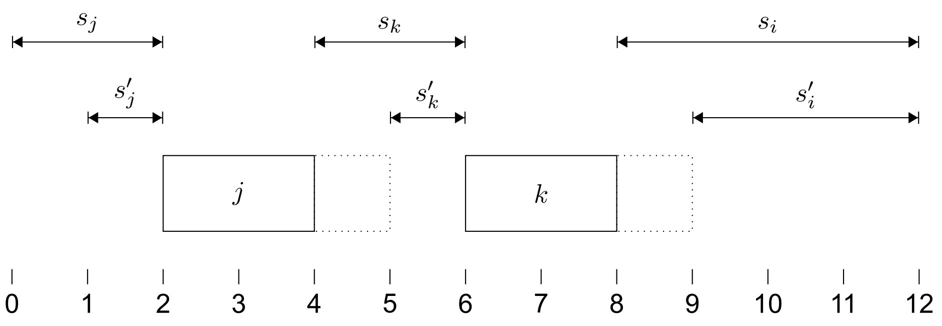
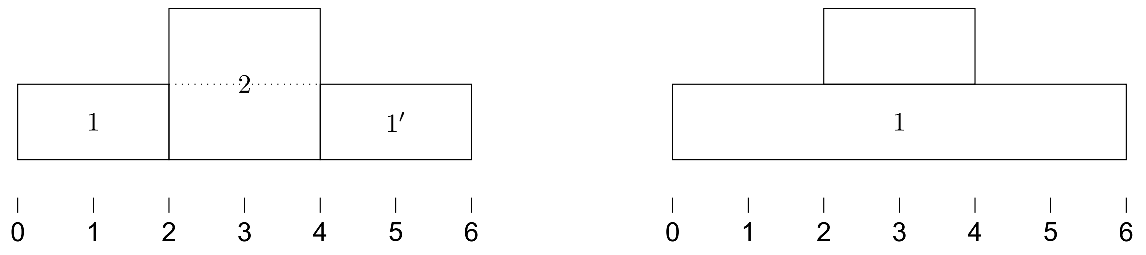


Figure 5.4: s_n defines the start variables for the given tasks of j, k, i. These tasks are not capable of overlapping. Their mandatory components are depicted by the outlined boxes. The dotted boxes represent the additional mandatory component for their s'_n variants.

To an extend skipping redundant profiles becomes an automatic feature instead of a possible risk but other risks are introduced. A redundant profile is one which contains more tasks than required. Typically in those cases the tasks with the lowest capacity are not actually part of the reason for propagation, as the capacity when not shifting was going to overload anyway, but they do appear in the profile generated by the timetable depending on the implementation. However, utilizing these cumulative literals means that we can make bigger steps than our profiles are large. This allows us to ”skip” over profiles of which we have already seen a subset of when propagating in sequence. However sequences that start off with the super set remain to be an issue as in this case the technique has no automatic features to detect the redundancy. This may also lead to overlap between literals in scenarios where redundant tasks enter the profile. This may hinder propagation as there is duplication and timetable may therefore not propagate on the correct literal, but a duplicate, meaning certain learned clauses will not fire until the literal becomes trivially true.



(a) A timetable where between 2 and 4 the mandatory consumption increases. We assume profile 1’ to be a shifted version of profile 1 and that profile 2 its tasks are a strict superset of profile 1. Any task that was already shifting due to profile 1. Will now pick up redundancies when it shifts over profile 2.

(b) A timetable where between 2 and 4 the mandatory consumption increases. We assume we are shifting a task over profile 1. This time the profile bound agnostic approach of the cumulative literal directly propagates to the end of the profile, avoiding the superset previously defined in profile 2.

Figure 5.5: A scenario depicting redundant profiles in the timetable. On the left we see the propagation as usually performed by a timetable implementation. On the right we see the approach of the cumulative literal.

5.2.3. Downsides

Utilizing the cumulative literals in the conflict explanations of timetable was not achieved. While there may be unexplored avenues this implementation of the cumulative literals will not utilize the cumulative literals when an overflow has happened in the timetable itself. Any conflicts raised by the GEQMin and LTMin propagators will utilize these literals as it is part of the reification process to do so. Timetable may also raise a conflict on the cumulative literals but only when setting the literal and seeing it was already set to the other polarity. However any timetable overflow error is still described by the underlying explanation of naive, big-step or, pointwise.

If a conflict with the cumulative literal is raised on the same level that it was inferred on it has a low chance of entering the learned clause with the 1UIP algorithm. The main reason for this is that the propagation which lead to the conflict was preceded directly by the propagation that set the cumulative literal. As the 1UIP algorithm describes adding all literals from the current decision level to be swapped out until we reach the 1UIP we immediately swap out the new cumulative literal. Note that we may therefore only find other cumulative literals that were already on the trace via other variables with lower bounds set. However even then the cumulative literals may be hard to add to the learned clause as described by the following section.

The cumulative literals have a limited entry scenario when trying to enter a learned clause in the 1UIP algorithm. 1UIP is an algorithm that will swap out predicates from the nogood with its explanations until only one predicate from the current decision level is left. This means that for our new literal to enter the learned clause we would prefer to set it at an earlier decision level and then propagate another time on the conflicting decision level with the shifted variable in the conflict. This guarantees the literals entry into the produced no-good. As the cumulative literal was set at a previous level it will not be swapped out. However, depending on how the problem is solved that second propagation may be limited in appearance. One example for this is the RCPSP benchmark from the MiniZinc benchmark suite which fixes tasks greedily. As the tasks are fixed to a starting point when first viewed by timetable there is no need for such a second propagation. Thereby it is only possible for this second propagation to occur if the cumulative literal was a learned property or eventually tasks become bounded enough yet not fixed to allow for those unfixed profiles to appear which would result in second propagations. This may hinder the cumulative literal to perform as an unlucky variable/value selection strategy may therefore prevent the technique from being applied more thoroughly or at all.

6 Experimental Results

We now delve into the performed experiments. We initially discuss how these experiments were performed, what are the metrics we are interested in and why, and on what benchmarks are we putting our focus. This should allow for a proper reproduction of the work provided. After this we aim to introduce the context for the first four research questions, concerning our interests in linear inequalities, in section 6.2. Finally we focus on the last two research questions, concerning the cumulative literal, in section 6.3. The research questions will be repeated and answered in chapter 7.

6.1. Experimental Setup

In order to describe how to reproduce the work put forth we explain a number of elements here. We initially describe the method used, explaining version numbers and intricacies of the environment and comparison method. We then explain what comparisons are made, focusing on different solver configurations under test. We follow this up with a view of the relevant benchmarks and finally a description of the metrics that we will be investigating.

6.1.1. Method

The proposed techniques were implemented on top of the already existing open-source solver Pumpkin. This solver was built with Rust 1.85.0. All instances are decomposed FlatZinc models decomposed by use of MiniZinc 2.9.2. and Pumpkins own FlatZinc decomposition rules. All experiments were run on the DelftBlue supercomputer specifically the partition utilizing Xeon E5-6248R 24C 3.0GHz processors with 3.8GB of memory for each process. Each experiment is timed out after 10 minutes of solver time as reported by the solver itself. In the case of printing the statistics taking up a significant portion of runtime the job is terminated after 12 minutes.

Comparisons are based on a best common denominator technique. Within each table the respective configurations are shown up top. For every test instance we then look at the best achieved result per configuration in the table. We take the worst of those achievements and mark that objective value as the best common denominator. Comparisons will then take place as if every solver only managed to solve to this best common denominator. If an instance does not have a best common denominator it is discarded for all solvers. This is helpful as some of the benchmarks provided are quite small and comparing solely on optimally solved problems would result in an even lower sample size.

6.1.2. The Comparisons

All solvers will be compared on three main search strategies: Annotated search, Free search, VSIDS. Annotated search resembles the search strategy present in the MiniZinc model, this is typically some search strategy known to work well. Free search is a solver specific approach to a search strategy, in this version of Pumpkin it first solves an instance utilizing annotated search and once the first candidate solution is found it switches over to VSIDS. VSIDS is a dynamic search strategy utilizing a heuristic based on conflict appearance as described in chapter 3.

For linear extended resolution we compare the baseline Pumpkin without linear extended resolution against a variation that utilizes the sequential encoding, *seq*, and a variation that utilizes the totalizer encoding, *tot*. All solvers will run in their most general configuration multiplicity $m = 1$ (seq) and branching factor $b = 2$ (tot) unless otherwise mentioned. The variable orderings when setting up the encodings may also be changed denoted as input order (default), scalar (s), and random orderings (r). We also introduce a variation of the solver which does not remove duplicates created by overlap in the encodings, the variation which does deduplicate may be tagged *dedup*. All variations of the solver only utilize extended resolution on linear inequalities of size $\geqslant 4$ due to overhead costs.

For the cumulative literal we introduce three variations of the solver based on the underlying explanation type used by the timetable propagator: stepwise naive, stepwise bigstep and pointwise. We then compare all three explanations with their complements which have access to extended resolution. We compare them on the aforementioned search strategies as well as a slight variation of the default annotated search strategy.

6.1.3. The Benchmarks

Let us first introduce the benchmarks of interest for the linear inequality variations. From the MiniZinc benchmark suite we use the RCPSP- bl, j120, pack, and pack_d instances. Those will be split up into *RCPSP-time* & *RCPSP-task* denoting the decomposition used. Furthermore we also include a *general* set of benchmarks: the BIBD, multidimensional-knapsack, market split (s: satisfiable, u: unsatisfiable), radiation, road construction and vehicle routing problems also drawn from the MiniZinc benchmarks. We add two extra problems to this suite. The first is a trivially unsatisfiable problem consisting of 20 instances each with two linear inequalities that together are unsatisfiable e.g. $\sum x_i \leqslant k \wedge \sum x_i \geqslant k + 1$. Their goal is to display the purely theoretical benefit of linear extended resolution. We also include 50 knapsack instances from Jooken, Leyman, and De Causmaecker [15]. The original benchmark was trimmed by selecting 50 instances from the $n = 400$ pool selected by those which encountered the most conflicts within a set time-out. The variables were then ordered relative to their profit/cost ratio to be comparable to previous work [7]. All problems were picked based on the presence of int_lin_le constraints.

For the cumulative literal we are only interested in benchmarks that utilize cumulative. To remain as close to just cumulative as possible this means we specifically care about RCPSP benchmarks. For this we pick six RCPSP benchmarks, most of which are publicly available on the MiniZinc benchmark repository. We test on the bl, pack, pack_d, psplib_j120, and psplib_j90 dataset as these are easily accessible and have been used in this field consistently for multiple decades. We also include the new CV dataset[8]. As the CV dataset is more modern and was made to be challenging for exact methods, such as LCG solving, it may be a more important benchmark in these comparisons.

Benchmark	Count
bibd	16
bl	40
CV	623
j90	480
j120	600
knapsack	50
m-knapsack	7
market_split_s	20
market_split_u	20
pack	55
pack_d	55
radiation	23
road-cons	11
trivial_unsat	11
vrp	74

Table 6.1: Number of instances per benchmark

6.1.4. The Metrics

Here we share some insight into the metrics that we will be measuring in our benchmarks. We initially showcase the default metrics already available in Pumpkin. We then showcase metrics specific to extended resolution and finally add two sections to discuss extended resolution metrics relevant to linear inequalities and cumulative specifically

We start off by reviewing the metrics already available within Pumpkin. Of relevance are the number of conflicts, decisions, propagations, LBD and learned clause length $|l|$. The main metric of this thesis will be conflicts as a lower number of conflicts ultimately means a more efficient search. We utilize LBD and $|l|$ to measure the quality of the nogoods, lower scores are better as they indicate a more re-usable nogood which typically prunes a larger search space. As a solver spends a great deal enforcing the extended resolution channels we lay this bare by comparing on the number of propagations as a means of comparing the solver efficiency, again lower is better as it means we spent less effort keeping our encodings consistent. Finally we also measure the number of decisions made. While we will not make any direct comparison on the number of decisions it will be part of our analysis on VSIDS to determine how likely VSIDS is to make decisions on extended resolution predicates. Do note that time is not a main component of our analysis as practical efficiency was not a goal of the thesis.

Secondly we introduce metrics specific to extended resolution. We measure the ratio by which registered nogoods contain extended resolution and the nogood propagator propagates on nogoods with extended resolution. These are called %NoGood-extended (%NG-ext) and %NoGood-Propagation (%NG-prop) respectively. Together these two metrics form the basis for any extended resolution learning effectiveness research. If %NG-ext is low it means that very few extended resolution predicates actually end up in the nogood database, this is possible as our problems may not purely consist of extended resolution affected constraints or the 1UIP does not contain extended resolution. While a low %Prop-ext can indicate an irrelevance of the nogoods that contain extended resolution. The higher both metrics are the more likely extended resolution can have an impact.

For linear extended resolution we created a set of specific metrics related to the encodings. We keep track of two histograms, one counting the size of a partial sum covered by an extended predicate when it is encountered on learning and one for during nogood propagation. This helps us contextualize from where in the encodings the predicates we encounter typically hail as well as comparing their relevance. e.g. We may learn a lot of size two nogoods but they never propagate meaning they have low relevance. We also track a number of duplication statistics as it may be the case that two extended resolution predicates enter the nogood that cover similar elements. For this we decompose every nogood until no more extended resolution is present. We then track a series of values. The ratio of nogood propagations that contained duplication (%Prop-dup) and how many of the nogoods contained such duplication (%NG-dup). And two statistics to measure this overlap known as %OverlapPred determining per learned nogood how much overlap was present and RawOverlap counting every decomposed nogood and counting how many of the predicates were duplicated as per equation (6.1).

$$\%OverlapPred : \frac{1}{n}\sum \frac{\#duplicate}{\#total} \tag{6.1a}$$

$$RawOverlap : \frac{\sum \#duplicate}{\sum \#total} \tag{6.1b}$$

Finally for the cumulative literal we create new metrics to track our secondary propagators, timetable propagator, and potential overlap. We count how many propagations not directly enforced by the timetable propagator the GEQ and LT propagators perform, these may be referred to as secondary propagations. Typically these will be compared relative to the total number of propagations performed (%GEQ, %LT). We also track the total number of unique literals that were created. A unique literal is any set of tasks in the profile, a shifting task, and a polarity describing whether it was a right of a left shift. Here we also introduce %NG-dup and %Prop-dup similar to before to indicate overlap which is defined as there being a task which is both in the cumulative literal, and for which both bounds are present in the nogood.

6.2. Linear Extended Resolution

We now cover the experimental results of linear extended resolution. We initially compare the baseline Pumpkin implementation with the version that contains extended resolution. We follow this up with an analyses of the generality factors of multiplicity and branching factor. We then discuss the effect of VSIDS followed by the ordering of variables. We then compare the proposed version of the technique with one that does not apply variable re-use in the case of similar encodings. Finally we end with a deep dive into the overlap that occurs between partial sum predicates and other variables present within the nogoods.

6.2.1. Basic Results

This section will answer the why and if as to whether linear extended resolution works without any alterations to existing models and/or the solver on a broader set of benchmarks. To answer this we compare three solver variations, one without extended resolution, one utilizing the sequential encoding, and one utilizing the totalizer encoding. We utilize the most general configurations $m = 1, b = 2$, rely directly on the input ordering of the variables, and utilize the annotated search strategies.

Linear extended resolution is a very strong tool for LCG solvers. Extended resolution allows for more general nogoods allowing the search space to be pruned more effectively. A more effectively pruned search space should result in fewer conflicts. As this approach is capable of removing exponential amounts of work it has the ability of severely lowering the number of conflicts required to solve a problem. By utilizing linear extended resolution we see in table 6.2 that the general benchmarks attain much better performance while the RCPSP benchmarks either fluctuate or deteriorate. Focusing solely on the improving runs we see that the trivial unsat model now solves instantly and that other models obtain different levels of improvement ranging from moderate to orders of magnitude fewer conflicts. Even more complex models such as road construction and vehicle routing resolve with only half or an order of magnitude fewer conflicts. Therefore linear extended resolution is an extension that a general LCG solver should consider.

The RCPSP benchmarks show unprecedented behaviour by increasing the number of conflicts when utilizing extended resolution. Typically when implementing extended resolution the number of conflicts should always decrease as any single explanation considered in a vacuum is more or equally as general as the original. We have seen as such in previous work that any alteration made still resulted in fewer conflicts [7]. However, it seems that when combining these explanations into a single nogood we end up with a nogood that is less general than before. We note that this only happens on the RCPSP model but that it does occur for both decompositions of cumulative. The justification as to why this occurs will be presented in section 6.2.6 providing an argument that the more complex variables create duplication in a nogood.

category	config / suite	def	seq	tot
rcpsp time	bl	**3817**	8005	6463
	j120	**1571**	2248	2062
	pack	**30598**	39120	33980
	pack_d	**896**	993	969
rcpsp task	bl	**9406**	10590	9513
	j120	**11753**	14627	12780
	pack	**79604**	93912	82971
	pack_d	125K	125K	**124K**
general	bibd	847	400	**317**
	knapsack	274K	**10110**	21256
	m-knapsack	232K	**3543**	12175
	market_split_s	72933	**25352**	25450
	market_split_u	13091	**6390**	7585
	radiation	10829	**5065**	5097
	road-cons	17522	**1947**	3000
	trivial_unsat	297K	**0.0**	**0.0**
	vrp	44067	23629	**16050**

Table 6.2: A comparison between the average number of conflicts found.

Extended resolution remains to always be a trade off between effectiveness and efficiency. That is to say utilizing extended resolution can give great benefits in the form of fewer conflicts yet exacts a cost in terms of propagations. Table 6.3 shows us the other side of this trade off by viewing the number of propagations required. Most importantly the only benchmarks which directly win this trade off appear to be BIBD, both knapsacks, and the radiation benchmark. In those scenarios we achieve enough effectiveness that the propagational costs actually lower. Not all the cost is directly lowered if there are fewer propagations, there still exist extra channels that need to be checked each time but may not propagate. This table is therefore a reminder to the importance of an efficient implementation of both the solvers propagation procedure as well as the extended resolution techniques. Especially as such techniques can be incredibly expensive.

Each encoding has their own respective channelling cost to consider. The underlying architectures can drastically influence how much of the data structure requires to be updated each time. This is best demonstrated by comparing both tables 6.2 and 6.3 on the vehicle routing problem. Where the totalizer encoding out performs the sequential encoding by 1.5x in terms of conflicts but 50x in terms of propagations. Or by viewing the knapsack problem where the totalizer encoding loses out in conflicts but wins in terms of propagations, each time significantly.

category	config suite	def	seq	tot
rcpsp time	bl	**746K**	3971K	2842K
	j120	**1954K**	11M	7109K
	pack	**10M**	39M	29M
	pack_d	**4447K**	18M	14M
rcpsp task	bl	**1784K**	3015K	2668K
	j120	**7958K**	17M	13M
	pack	**11M**	22M	18M
	pack_d	**25M**	35M	32M
general	bibd	499K	**354K**	424K
	knapsack	**5952K**	11M	6350K
	m-knapsack	2647K	**1967K**	2974K
	market_split_s	**371K**	2682K	2576K
	market_split_u	**57K**	419K	573K
	radiation	**1437K**	1914K	1866K
	road-cons	140M	**55M**	67M
	trivial_unsat	1370K	32.0	**31.0**
	vrp	1292K	311M	**6361K**

Table 6.3: The average number of propagations required to solve the problem

To answer why this techniques work better we first look at the LBD scores. The patterns based on conflict improvements remain present within the LBD and average nogood size metrics indicating an improvement in nogood quality. The goal of extended resolution is to encode a more general state that was previously not describable with the available predicates. Typically this results in many predicates being required to describe a state now described by a single predicate. We therefore expect that solvers utilizing extended resolution will learn smaller learned clauses with smaller LBD scores. Table 6.4 shows as such. While the RCPSP benchmarks see little change there are phenomenal improvements made in the general benchmarks. This means that the general benchmarks are utilizing fewer predicates in their nogoods and achieving much better LBD scores, even in scenarios where the actual size varies little. However, for RCPSP this change is not seen, in fact the scores tend to get worse when utilizing linear extended resolution. This may hint at linear extended resolution only replacing one or two predicates in an RCPSP nogood or could be the result of going to another part of the search tree where larger specific nogoods are required to get out of again.

The sequential encoding is capable of providing the most general learned clauses when compared to the totalizer encoding. Both encodings have their own strengths and weaknesses. A strength of the sequential encoding is that it can cover any size partial linear inequality, when set to its most general configuration, in a single predicate. The totalizer encoding is limited to explaining with only predicates covering partial sums of size $i \in \mathbb{N}, b^i$. This means that the sequential encoding has the ability to create much more general explanations. We see this reflected in table 6.4 as the LBD score is typically lower for the sequential encoding on the general benchmarks as compared to the totalizer encoding. However, that the sequential encoding can make the most general explanation does not mean that it always will. Certain problems do not offer the most favourable circumstances, with this we typically refer to variable ordering in the encoding and variable selection ordering as will be discussed later in section 6.2.4. This then is the most likely reason that there are also scenarios where the totalizer encoding performs better in terms of LBD.

		LBD			|l|		
	config	def	seq	tot	def	seq	tot
category	suite						
rcpsp time	bl	**3.264**	3.422	3.363	**15.04**	19.92	19.24
	j120	**6.629**	7.767	7.249	**29.00**	40.89	39.57
	pack	**2.163**	2.575	2.339	**11.48**	15.40	14.14
	pack_d	**1.042**	1.235	1.155	**26.23**	33.46	30.58
rcpsp task	bl	3.129	3.182	**3.127**	**9.164**	9.890	9.282
	j120	**6.373**	7.269	6.709	**16.66**	23.11	20.01
	pack	**2.532**	2.606	2.536	**7.498**	8.391	7.902
	pack_d	3.075	3.035	**3.003**	6.300	6.362	**6.178**
general	bibd	13.99	**4.336**	4.494	31.24	16.50	**14.82**
	knapsack	71.15	**1.991**	3.087	72.15	43.87	**20.61**
	m-knapsack	11.01	**1.561**	2.719	17.06	**4.995**	7.587
	market_split_s	7.607	**1.473**	2.405	8.700	**6.038**	7.498
	market_split_u	6.089	**1.557**	2.466	7.135	**5.929**	7.243
	radiation	3.254	2.754	**2.721**	11.47	**8.981**	9.532
	road-cons	74.49	**3.108**	3.408	**91.14**	1251	1080
	trivial_unsat	9.366	**0.0**	**0.0**	10.21	**0.0**	**0.0**
	vrp	1221	1142	**187**	1227	1148	**202**

Table 6.4: A comparison of nogood scoring statistics. LBD represents a better heuristic quality of the nogood whereas $|l|$ encodes the actual size of the nogood representing the actual space taken up on average.

This improvement in nogood quality stems from extended resolution. As we can see the significant presence of extended resolution in both nogood and nogood propagations as per table 6.5. The main reason for this is that if a problem either mainly consists of and/or is mainly constrained by linear inequalities there will be many extended predicates used in explaining conflicts and propagations. Be that as it may, a theoretical model only consisting of linear inequalities can in practice be a much more complex model as is the case with the RCPSP model, containing redundant constraints and a large array of bool2int constraints. This is also the reason that certain problems have a larger affinity for linear extended resolution than others. In this case the general class of benchmarks was chosen based on linear inequality presence. Therefore when viewing table 6.5 we see this play out. RCPSP models being more complex do not manage to obtain a large extended resolution presence in the nogoods while for general benchmarks they are very present. One exception to this is the trivial unsat suite. As this technique is capable of solving this very theoretical problem so efficiently it does not encounter any conflicts to learn nogoods from.

Extended resolution presence within a nogood does not guarantee it is a good nogood. While we discuss possible reasons for the behaviour shown in table 6.5 later in section 6.2.6 we for now note that there are benchmarks on which there are a lot of extended resolution predicates in the nogoods but those nogoods tend to never propagate, or propagate fewer than their non extended counterparts. This is mainly prevalent among the RCPSP benchmarks and the BIBD benchmark. A potential explanation is that such nogoods backtrack further and hence propagate fewer or that the technique does not work for them.

		%NG-ext		%Prop-ext	
	config	seq	tot	seq	tot
category	suite				
rcpsp time	bl	84.3	90.4	8.50	9.02
	j120	73.3	76.4	1.45	1.43
	pack	44.4	43.9	2.73	2.74
	pack_d	16.1	16.5	0.259	0.190
rcpsp task	bl	13.9	11.0	1.14	1.07
	j120	25.7	24.9	0.670	0.643
	pack	7.70	5.09	1.95	1.71
	pack_d	1.71	0.830	0.840	0.848
general	bibd	80.4	83.2	0.309	0.342
	knapsack	100	100	100	100
	m-knapsack	100	99.9	100	99.9
	market_split_s	100	99.9	100	99.9
	market_split_u	100	99.9	100	99.9
	radiation	99.9	97.5	99.6	97.1
	road-cons	99.6	99.8	99.0	97.9
	trivial_unsat	0.0	0.0	0.0	0.0
	vrp	95.1	99.9	93.0	98.6

Table 6.5: A comparison of linear extended resolution presence in the nogoods depicted by %NG-ext denoting how many of the nogoods contain extended resolution and %Prop-ext denoting how many nogood propagations were conducted on nogoods with nogood propagations

We can also see exactly from where in the encodings this benefit stems from. Predicates which cover a larger partial sum are significantly more important than the predicates covering a smaller partial sum. The most general predicates cover the largest partial sums. This means their relevance is much higher than predicates which cover a smaller portion. For this we take a look at section 6.2.1 describing predicate relevance for the market split unsatisfiable benchmarks. Here we see that our predicates which cover larger partial sums are much more relevant for no good propagation. In fact the covers of size 16 and 17 appear five times less in the no goods than those of size 11 and 12 yet relatively propagating much more. e.g. a size 16 cover is about 25 times more likely than a size 11 cover to be used for propagation. The improved performance of extended resolution therefore mainly seems to come from these predicates which cover larger parts of their respective linear inequalities.

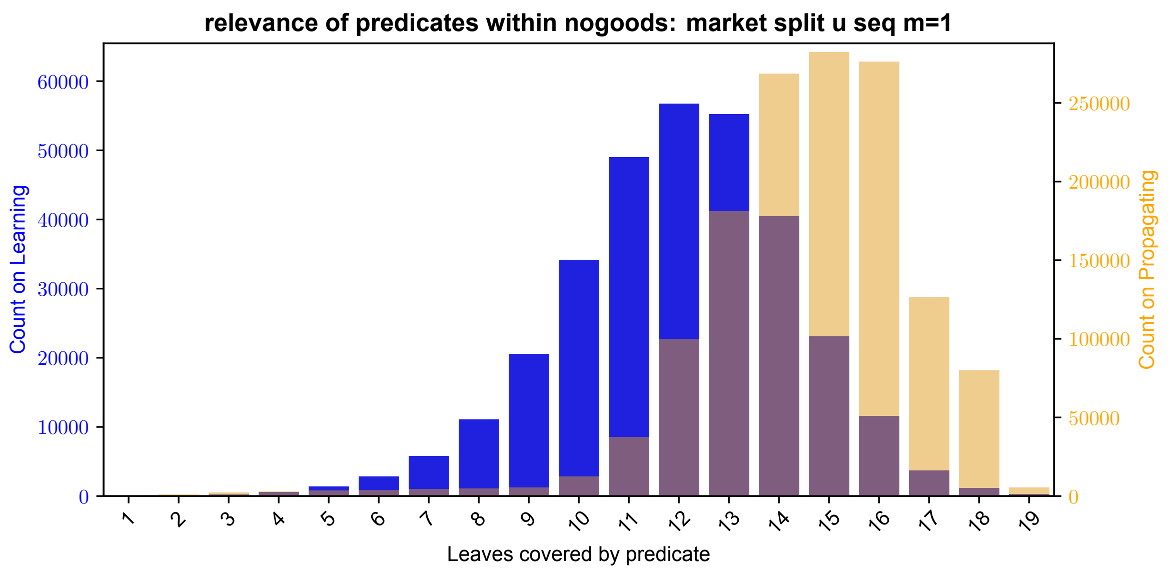


Figure 6.1: A histogram denoting predicate relevance by the size of the covered partial sum. The left axis denotes how often such a predicate is encountered in a nogood. The right axis denotes how often such a predicate appears in a propagating nogood. Results were obtained on the market split u benchmark.

6.2.2. Multiplicity

This section aims to answer how the encodings are influenced by their respective generality settings of multiplicity and branching factor. These are hyper-parameters that dictate how often we introduce a partial sum relative to the number of variables present in the inequality. We repeat the same scenario as before but now allow to vary the generality setting between a range of $[1 \dots 5]$.

The generality versus channelling cost trade off is present but varies on a model basis. Utilizing a worse generality can help improve solver efficiency. By decreasing the generality we have a sparser encoding with fewer partials variables that we need to create. From there it naturally flows that we also require fewer propagations to keep the encodings consistent resulting in a lower cost. It also means that there is an optimal generality setting per benchmark where a specific generality setting manages to solve the problem the quickest. This models a trade-off between effectiveness and efficiency. We show three variations of this trade-off in figure 6.2. Where pack-d task has no specific preference, radiation shows a clear local minima near 3 and 4 and market split seems to potentially have a minima near 4 or may continue to decrease.

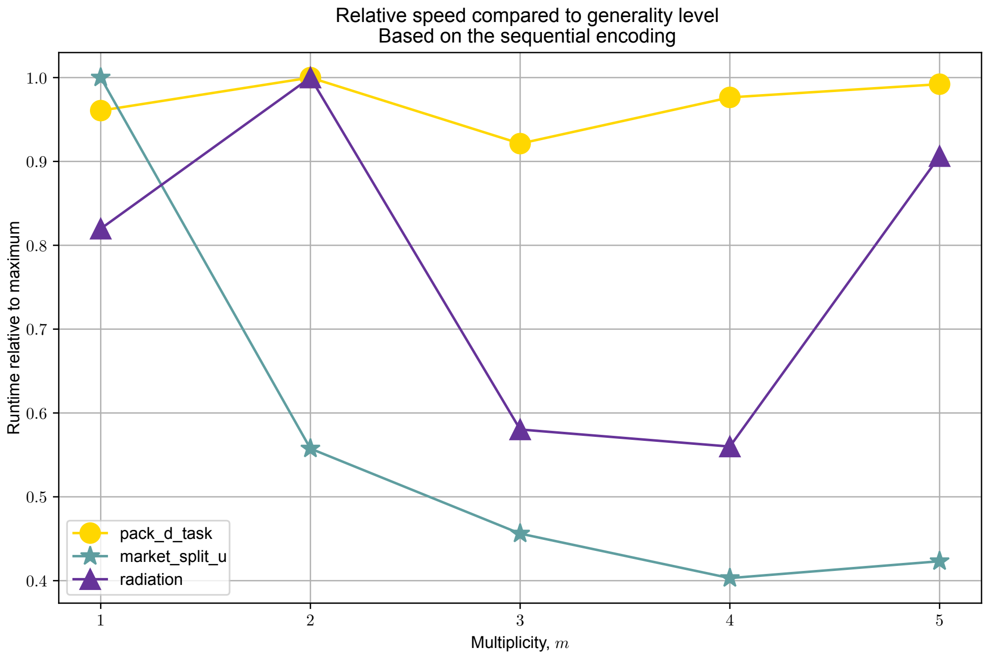


Figure 6.2: A line plot denoting the generality versus runtime trade-off. We focus on three specific benchmarks with different behaviours

The reason that this trade off exists is because as we decrease generality we increase the number of conflicts we encounter. That data is displayed in table 6.6 We see that as we decrease generality we decrease the effect of extended resolution. While worsening the generality does not immediately undo all of the effects of extended resolution we see the effects weaken as they return more and more to the default behaviour without extended resolution.

Sometimes worsening the generality can actually lower the number of conflicts found. By changing the multiplicity and branching factor the encoding creates different clusters which may therefore result in different behaviour. Such a change should not be too significant unless the variables in the inequality are specifically clustered in which case breaking up such a cluster can be detrimental. No further research into this was performed but it serves as a viable alternative explanation as to why a more general setting does not always achieve fewer conflicts as opposed to arguing that it would be solely because of different propagation orderings.

category	config suite	seq_m1	seq_m2	seq_m3	seq_m4	seq_m5	tot_b2	tot_b3	tot_b4	tot_b5
rcpsp time	bl	8005	8519	8798	7766	**7474**	6463	**5722**	6187	6674
	pack	39K	39K	38K	38K	**37K**	**33K**	35K	34K	34K
	pack_d	888	**871**	896	877	877	864	801	869	**797**
rcpsp task	bl	10K	10K	11K	10K	**9750**	**9513**	9573	9895	9637
	pack	84K	85K	82K	**79K**	82K	74K	**73K**	79K	76K
	pack_d	106K	105K	106K	105K	105K	105K	105K	105K	106K
general	bibd	400	**385**	390	422	433	**317**	349	391	470
	knapsack	**11K**	14K	17K	21K	23K	**30K**	49K	47K	78K
	m-knapsack	**3543**	4253	6035	6609	8075	12K	**11K**	21K	19K
	market_split_s	**25K**	26K	27K	27K	28K	**25K**	34K	32K	34K
	market_split_u	**6390**	6462	6858	7237	7916	**7585**	8182	8760	8974
	radiation	**7408**	7700	7951	8330	9028	**7638**	8397	8785	9154
	road-cons	**2989**	3155	3327	3689	3842	**4844**	5867	5833	10K
	trivial_unsat	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	vrp	**23K**	24K	25K	27K	29K	**16K**	17K	19K	22K

Table 6.6: A comparison of average conflicts encountered. We compare different levels of generality in either encoding denoted by the mi or the bi for the used multiplicity or branching factor respectively.

Utilizing a lower generality can help improve solver efficiency. By lowering the generality we have fewer partials variables that we need to create. From there it naturally flows that we also require fewer propagations to keep the encodings consistent. It also means that there is an optimum generality setting per benchmark where a specific generality setting manages to achieve the fewest number of propagations to solve a problem, thus modelling a part of the trade-off between solving effectiveness and channelling costs. While not overly present we can see for road construction that if we are using the totalizer encoding we may wish to keep it at the most general configuration.

category	config suite	seq_m1	seq_m2	seq_m3	seq_m4	seq_m5	tot_b2	tot_b3	tot_b4	tot_b5
rcpsp time	bl	3971K	2861K	2591K	2132K	**1942K**	2842K	2008K	1894K	**1840K**
	pack	39M	27M	22M	20M	**18M**	29M	23M	21M	**18M**
	pack_d	17M	11M	9174K	8133K	**7401K**	13M	9828K	9177K	**7861K**
rcpsp task	bl	3015K	2604K	2548K	2360K	**2094K**	2668K	2308K	2247K	**2122K**
	pack	19M	16M	14M	**13M**	**13M**	15M	13M	14M	**12M**
	pack_d	28M	24M	23M	**22M**	**22M**	25M	23M	**22M**	**22M**
general	bibd	354K	303K	298K	**297K**	310K	424K	344K	**335K**	359K
	knapsack	13M	9932K	7270K	5820K	**4841K**	8650K	7079K	**6015K**	6429K
	m-knapsack	1967K	1268K	1204K	967K	**897K**	2974K	**1617K**	2501K	1621K
	market_split_s	2682K	1567K	1118K	869K	**702K**	2576K	2437K	**1297K**	1898K
	market_split_u	419K	228K	173K	129K	**121K**	573K	286K	323K	**184K**
	radiation	4030K	2981K	2586K	2407K	**2325K**	3704K	2921K	2589K	**2545K**
	road-cons	85M	**84M**	85M	87M	87M	**110M**	115M	112M	132M
	trivial_unsat	32.0	24.0	22.0	**20.0**	**20.0**	31.0	25.0	**21.0**	**21.0**
	vrp	311M	105M	59M	40M	**30M**	6361K	4178K	3499K	**3273K**

Table 6.7: A comparison of propagations required on average. We compare different levels of generality in either encoding denoted by the mi or the bi for the used multiplicity or branching factor respectively.

6.2.3. VSIDS

This section aims to answer what the effects are of extended resolution on a solver configured with the dynamic search strategy VSIDS. This helps to create an understanding as to how informative these literals may be when they are decided upon. This section considers VSIDS to be active in all variations of the solver.

Combining VSIDS with extended resolution has the possibility of significantly improving performance. Combining the already shown to be stronger explanations with the now also more general decisions gives a solver a tremendous increase in generality. However, this does not have to be good. VSIDS is a strong technique because it tends to focus on recent conflicts, searching within a specific area of the search space in an attempt to solve the current conflicts. Extended resolution runs the risk of introducing unnecessary chatter in the form of extra variables for VSIDS to consider. This double edged sword is clearly visible in table 6.8 when viewing some of the totalizer encodings. VRP, initially finding around a factor three improvement without VSIDS now sees a 20x improvement. Yet, as it remains VSIDS bibd sees a factor of seven in terms of deterioration where it initially was around three times better.

	config	def	seq	tot
category	suite			
rcpsp time	bl	3771	3913	**3625**
	j120	**163**	677	237
	pack	**12K**	14K	**12K**
	pack_d	**51.97**	168	110
rcpsp task	bl	**5285**	7748	6474
	j120	**3735**	8694	4507
	pack	41K	40K	**38K**
	pack_d	49K	**38K**	45K
general	bibd	**302**	366	2166
	knapsack	**58.5**	10K	212
	m-knapsack	340K	106K	**81K**
	market_split_s	408K	**51K**	65K
	market_split_u	**7769**	9155	8477
	radiation	28K	**16K**	18K
	road-cons	582	**545**	657
	trivial_unsat	141K	**0.0**	**0.0**
	vrp	63K	28K	**3543**

Table 6.8: A comparison of conflicts encountered between all three possible techniques when utilizing the VSIDS method. The abnormality in knapsack is due to low n

VSIDS is not capable of exploiting linear extended resolution as much with regards to learning better nogoods. As VSIDS is a heuristic search strategy the ordering by which it selects variables to decide upon may be random in the eyes of our encodings. This potentially unfavourable ordering our encodings follow opposite of the ordering VSIDS arrives at means that the encodings have to use less general explanations resulting in lower quality learned clauses. This is best observed for the sequential encoding which before managed to get similar LBD scores for RCPSP and LBD scores around 1 for the general benchmarks and now no longer does. Showing a strong deterioration in the quality of the nogoods. We can back this up by taking a look at the predicate relevance histogram for unsatisfiable market split again in section 6.2.3. What we see here is that when utilizing VSIDS we encounter many more sizes of partial sums. Where before we mainly found predicates only covering partial sums $\geqslant 9$ it is now almost uniformly distributed with the variation potentially tapering off near the predicates covering large partial sums.

VSIDS can still exploit linear extended resolution through its variable selection process. By appearing in many nogoods the extended resolution predicates are relevant enough for VSIDS such that it will make decisions on them. While such a decision is typically not capable of fixing any individual variable from the original problem it does serve as a very general decision capable of deciding some property over multiple variables at once. While VSIDS will not favour any specific node according to what we think is the the most general one, e.g. making decisions that either explicitly or implicitly cover a large partial sum, it may still be able to exploit this extra level of generality. According to table 6.9 VSIDS tends to make a large portion of its decisions on extended resolution predicates. However, we do not have the data to tell what those decisions exactly are.

category	config suite	def	seq	LBD tot	def	%Decisions-ext seq	tot
rcpsp time	bl	**5.05**	5.80	5.44	0.0	28.2	**29.0**
	j120	**4.68**	8.21	4.86	0.0	**40.1**	36.7
	pack	**9.29**	12.8	10.9	0.0	29.5	**31.2**
	pack_d	**3.58**	5.01	3.94	0.0	39.9	**40.1**
rcpsp task	bl	**4.92**	5.88	5.36	0.0	14.8	**15.4**
	j120	**5.64**	10.9	6.81	0.0	**32.3**	24.4
	pack	**9.10**	11.0	10.8	0.0	**11.8**	11.3
	pack_d	6.59	6.63	**6.54**	0.0	**10.0**	8.39
general	bibd	5.17	**2.69**	3.40	0.0	18.6	**24.9**
	knapsack	217	10.0	**5.17**	0.0	72.3	**76.0**
	m-knapsack	16.7	11.4	**10.2**	0.0	52.0	**76.1**
	market_split_s	8.48	6.34	**5.82**	0.0	51.7	**53.6**
	market_split_u	6.19	5.04	**4.91**	0.0	**52.6**	51.2
	radiation	8.54	**5.79**	5.86	0.0	**51.6**	47.4
	road-cons	9.30	**8.76**	9.49	0.0	**6.46**	5.18
	trivial_unsat	8.87	0.0	0.0	0.0	0.0	0.0
	vrp	59.6	26.0	**22.5**	0.0	38.6	**71.7**

Table 6.9: A comparison based on the LBD scoring metric to assess nogood quality when utilizing VSIDS. The second column describes how many of the decisions made by VSIDS were performed on an extended resolution predicate.

Extended resolution predicates that do not cover a large partial sum can still be very relevant if they cover a large section implicitly. In section 6.2.1 we claimed that only extended resolution predicates covering a large partial sum were really relevant within the nogood scope. However, this argument requires a small addendum. An extended resolution predicate, covering only a small part of the total linear inequality inversely covers a very large part of the linear inequality. That is to say, while the predicate directly only covers a small part of the partial sum it also encodes information over all the variables it does not explicitly cover. e.g. we have a small partial sum covered by p and enter $[\![p \leqslant 9]\!]$ into a nogood. The maximum constraint of the entire inequality is set to be $\sum \leqslant 10$. This means that while p only covers a small partial sum explicitly it implicitly also covers the rest of the partial sum with a nogood of $[\![p' \geqslant 1]\!]$ where p' explicitly covers the partial sum not covered by p. This behaviour is properly demonstrated in section 6.2.3 by the convex curve that can be seen through the yellow bars. While we do not show all the histograms here VSIDS typically relies more on this implicit behaviour for its nogoods. Also few of its histograms show the ability to create predicates covering large partial sums and many have the orange curve on the left side of the histogram indicating it relies heavily on smaller covers of partial sums for its nogood propagations.

While VSIDS may not be as good at utilizing extended resolution to create more general nogoods it does still exploit linear extended resolution. While it does not have to come as a surprise it is good to validate whether or not extended resolution manages to enter the learned clauses when we utilize VSIDS. To this extend the answer is yes according to table 6.10. Most of the benchmarks show that more than 50% of all learned clauses contain some form of extended resolution with none lower than 15%. Even more notable are the RCPSP benchmarks, showing high extended resolution presence but almost no utilization of those nogoods.

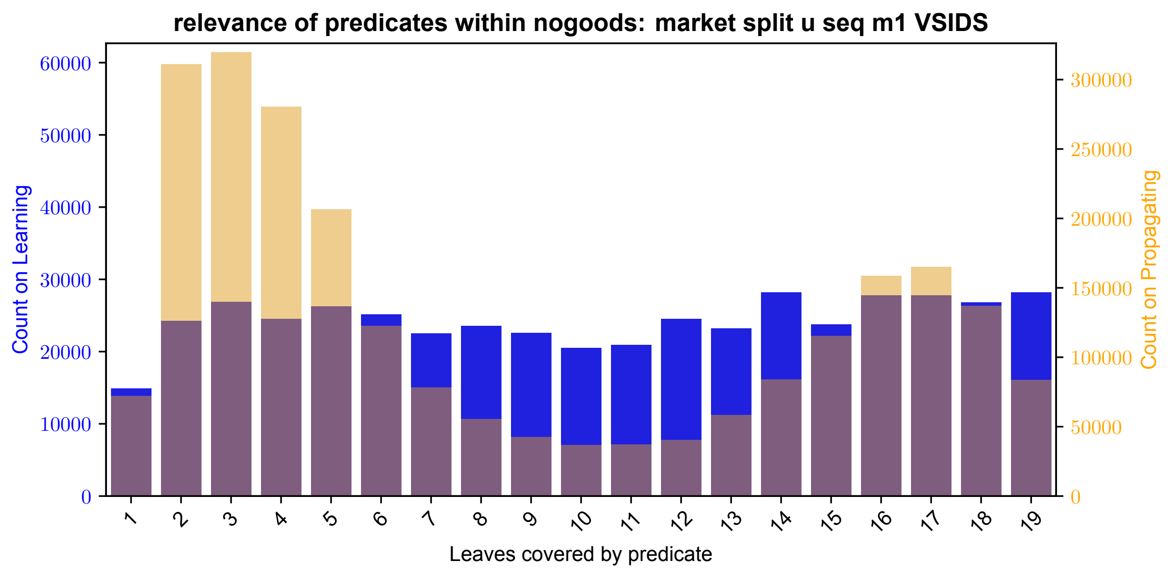


Figure 6.3: A histogram denoting predicate relevance by the size of the covered partial sum. The left axis denotes how often such a predicate is encountered in a nogood. The right axis denotes how often such a predicate appears in a propagating nogood. Results were obtained on the market split u benchmark utilizing the VSIDS search strategy.

		%NG-ext		%Prop-ext	
	config	seq	tot	seq	tot
category	suite				
rcpsp time	bl	90.6	91.7	6.04	5.85
	j120	88.3	83.5	0.050	0.020
	pack	74.2	72.4	2.69	1.87
	pack_d	89.4	86.9	0.017	0.014
rcpsp task	bl	35.9	32.5	1.59	1.43
	j120	55.3	43.4	0.221	0.117
	pack	32.8	29.3	2.10	1.67
	pack_d	15.5	17.2	1.01	1.26
general	bibd	79.9	77.8	0.793	1.02
	knapsack	100	100	100	100
	m-knapsack	99.1	99.9	99.9	99.9
	market_split_s	96.1	99.8	99.9	99.9
	market_split_u	96.2	98.9	99.7	99.8
	radiation	99.5	98.4	99.7	87.7
	road-cons	47.6	39.8	15.1	11.4
	trivial_unsat	0.0	0.0	0.0	0.0
	vrp	99.7	98.3	99.1	98.0

Table 6.10: A comparison of linear extended resolution presence in the nogoods when utilizing the VSIDS strategy. %NG-ext denotes how many of the nogoods contain extended resolution and %Prop-ext denotes how many nogood propagations were conducted on nogoods with nogood propagations

6.2.4. Ordering

We now investigate to see what happens if the variable ordering given by the input file was lost. We aim to answer how significant a good variable ordering is and which encoding is most robust to poor orderings. We specifically answer this question in relation to the input ordering as is given in the model file, a scalar ordering sorting our inequality beforehand based on any scalar values, and a random ordering where the inequality is scrambled beforehand.

Poor orderings still outperform not using linear extended resolution at all. The generic explanations used for linear inequalities are still worse explanations compared to those of a poorly optimized extended resolution one. Even if the entire linear inequality needs to be resolved for the 1UIP there will eventually be some predicate that will cover a partial sum, however small it may be it will still be more general than the generic explanation. We see this in table 6.11, focusing mainly on the general benchmarks. Here we see that no matter the ordering we pick we always outperform the baseline approach. This means that while a good ordering can help with a significant boost, implementing the technique is the most important factor, as even random orderings can achieve better performance than the baseline. For RCPSP this is more difficult to explain as it did not respond well to the technique initially. The most interesting part is that there is almost no difference between the input ordering and the random ordering. We do see an improvement between input ordering and scalar ordering between the two encodings. Suggesting that the input ordering for RCPSP was already unfavourable and perhaps a better ordering must be found before it can properly exploit linear extended resolution.

Aligning the ordering with the search strategy is important. An intuitive way in computer science to approach many problems typically starts with sorting your problem. That upfront one time cost typically pays back later as it allows for other faster approaches on your now sorted entries. Ordering our variables by scalar is not the weirdest concept, in RCPSP we may expect that creating clusters of tasks which all have high resource requirements will quickly lead to conflicts. As these tasks will typically clash with each other in a fight over the resources they are highly relevant for conflicts and therefore putting them close together can be beneficial. Such reasoning can also be extended to knapsack problems, high weight items will compete with each other more clearly, first determining the best high weight elements before adding low weight elements. However, the search strategies in these scenarios have not been altered. This means that even while the ordering is better it is not aligned with the search strategy. Therefore we run into the same worst case explanations sizes as we did random orderings, all be it to a slightly lesser extend due to the clustering of potentially relevant to each other variables. We see this too in table 6.11, scalar sorting still loses out to the input ordered sort but tends to achieve equal or better performance than the randomly sorted approaches.

	ordering			input		scalar		random
	config	def	seq	tot	seq	tot	seq	tot
category	suite							
rcpsp time	bl	3817	8005	**6463**	7229	**5148**	8530	**7158**
	j120	1303	1835	**1690**	1727	**1617**	1812	**1740**
	pack	21K	27K	**24K**	25K	**24K**	28K	28K
	pack_d	791	888	**864**	**841**	860	847	**792**
rcpsp task	bl	9406	10K	**9513**	10K	10K	11K	**10K**
	j120	10K	12K	**10K**	11K	**10K**	11K	11K
	pack	73K	87K	**76K**	80K	**78K**	**75K**	86K
	pack_d	108K	107K	107K	109K	**107K**	109K	**107K**
general	bibd	847	400	**317**	400	**317**	614	**577**
	knapsack	57K	**5832**	7718	42K	**39K**	51K	**47K**
	m-knapsack	232K	**3543**	12K	122K	**57K**	190K	**129K**
	market_split_s	72K	**25K**	25K	**53K**	56K	51K	**44K**
	market_split_u	13K	**6390**	7585	**10K**	11K	10K	10K
	radiation	10K	**5065**	5097	**5074**	5115	**8320**	8458
	road-cons	17K	**1947**	3000	**6578**	6665	12K	**10K**
	trivial_unsat	297K	0.0	0.0	0.0	0.0	0.0	0.0
	vrp	8493	5827	**5130**	6168	**5639**	6591	**6586**

Table 6.11: A comparison of the average number of conflicts encountered when comparing different types of orderings.

Part of the reason why random orderings work so poorly for the sequential encoding yet manage to thrive better on the totalizer encoding can be observed in figures 6.4 to 6.7 showcasing the market split unsatisfiable benchmark once more. Here we compare how the presence of different sized predicates changes between the input ordering and the random ordering. We first see that with the random ordering the sequential model becomes chaotic. It tends to learn predicates mainly from some arbitrary clusters and even the potentially high value predicates at 1 and 19 seem to not be over utilized. This indicates that the sequential encoding is learning predicates of lesser value. The advantageous structure is lost. However, if we view the totalizer approach it manages to retain most of its structure. The structures are not identical, they should not be as we do have a decrease in performance. Which can be attributed to the loss of relevant size 2 covering predicates. But because the totalizer encoding is made up of independent leaves instead of a long dependent chain it has branches in the tree it can still explain in a general fashion. This allows us to retain a lot of relevant predicates with size 4. We see this with other benchmarks as well, where the learning of the sequential encoding no longer follows a curve and the totalizer encoding, while different remains to obtain relevant clauses at fixed sizes.

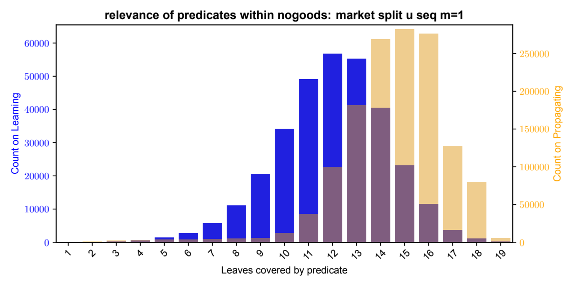


Figure 6.4: sequential encoding predicate relevance histogram for unsatisfiable market split, input ordering

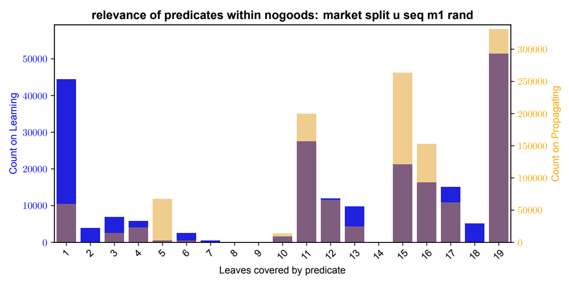


Figure 6.5: sequential encoding predicate relevance histogram for unsatisfiable market split, random ordering

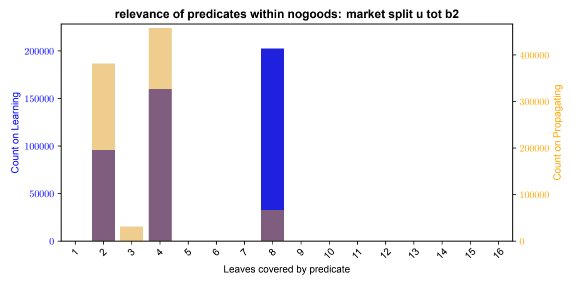


Figure 6.6: totalizer encoding predicate relevance histogram for unsatisfiable market split, input ordering

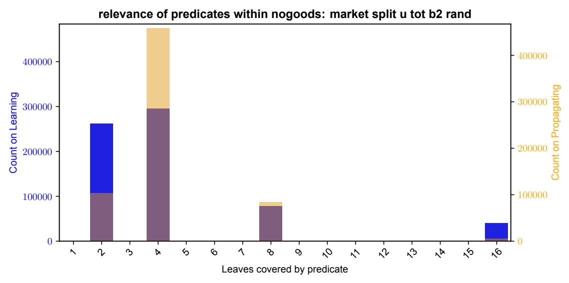


Figure 6.7: totalizer encoding predicate relevance histogram for unsatisfiable market split, random ordering

This loss of the most general predicates due to an improper ordering can drastically deteriorate the nogood quality, affecting the sequential encoding the most. The more our search strategy and variable ordering are aligned the more general the explanations can be and the more general the eventual 1UIP is. If this ordering is therefore random or ordered against the search strategy finding a good general 1UIP can be substantially hindered as explaining a conflict near the end of the linear inequality may require the 1UIP to explain the entire linear inequality if the conflict arose due to a change on a variable near the beginning of the linear inequality. While both encodings have the same worst case explanation size of $O(n)$ the scenarios in which this occur are different. Both encodings approach $O(n)$ if $O(n)$ variables from the inequality in the nogood were set at the current decision level and thus need to be explained. Yet, the sequential encoding also creates an $O(n)$ explanation if the distance between any two variables approaches $O(n)$. This means that in a scenario where only two variables are present in the nogood from the current decision level the sequential encoding can worst case still create an $O(n)$ explanation while the totalizer encoding would create an $O(log(n))$ explanation. We therefore expect that with random orderings the totalizer encoding is more robust to such scenarios and manages to retain some level of performance where the sequential encoding may lose it entirely.

We see this reflected in table 6.12. When ordered randomly it is the totalizer encoding which manages to consistently find higher quality nogoods.

	ordering			input		scalar		random
	config	def	seq	tot	seq	tot	seq	tot
category	suite							
rcpsp time	bl	3.264	3.422	**3.363**	3.453	**3.344**	3.513	**3.446**
	j120	6.558	7.602	**7.119**	9.447	**8.029**	9.942	**8.313**
	pack	2.070	2.402	**2.204**	2.420	**2.295**	2.708	**2.528**
	pack_d	1.049	1.242	**1.162**	**1.217**	1.128	1.137	**1.107**
rcpsp task	bl	3.129	3.182	**3.127**	3.188	**3.161**	3.293	**3.212**
	j120	6.339	7.174	**6.634**	8.305	**7.321**	8.945	**7.635**
	pack	2.550	2.611	**2.542**	2.661	**2.616**	2.734	**2.718**
	pack_d	3.084	3.044	**3.009**	3.085	**3.044**	3.049	**3.031**
general	bibd	13.99	**4.336**	4.494	**4.336**	4.494	9.919	**6.883**
	knapsack	67.78	**2.043**	2.881	14.90	**9.438**	59.66	**28.76**
	m-knapsack	11.01	**1.561**	2.719	9.531	**7.870**	10.09	**9.072**
	market_split_s	7.607	**1.473**	2.405	6.429	**6.055**	6.794	**5.619**
	market_split_u	6.089	**1.557**	2.466	5.362	**5.267**	5.399	**4.842**
	radiation	3.254	2.754	**2.721**	2.762	2.762	3.200	**3.108**
	road-cons	74.49	**3.108**	3.408	41.58	**13.85**	51.17	**14.77**
	trivial_unsat	9.366	0.0	0.0	0.0	0.0	0.0	0.0
	vrp	1222	1144	**186**	1414	**223**	640	**253**

Table 6.12: A comparison of nogood quality via the LBD metric when comparing different types of orderings.

6.2.5. Naive

One may be compelled to forego the implementation of de-duplication. While this thesis mainly considers the technique with de-duplication implemented the initial implementation did not have this. To test this configuration we rollbacked the solver to before when this technique was added and compare their performances. As the bookkeeping utilized for the de-duplication is also the backbone that drives the extended resolution specific metrics those will not be available for comparison. We aim to answer if such a de-duplication technique is necessary.

Duplication incurs a significant cost for equality constraints. The equality constraint is modelled as two separate inequality constraints, $\geqslant \wedge \leqslant$ in Pumpkin. This means that for any equality constraint we create twice as many variables than actually required. Having twice as many variables means we have twice the propagational cost to enforce consistency. we view table 6.13 to support our claim. Initially we see no difference for the RCPSP benchmarks. These also do not contain any linear equalities. But the more present the linear equalities are the more the number of propagations decrease. BIBD and market split contain mainly linear equalities and those are among the benchmarks who stand the most to gain. But we also see increases for vrp and knapsack. While these problems do not explicitly define an equality constraint it is there implicitly in the objective functioning binding the objective value. It is not guaranteed that a decrease in number of propagations will be achieved. Radiation is a benchmark which is also in the category with a series of linear equality constraints but there we do not see this benefit, most likely due to the equalities being too small meaning the cost for keeping the encodings consistent is negligible.

category	config suite	seq	seq-dedup	tot	tot-dedup
rcpsp time	bl	3971K	3971K	2907K	**2842K**
	j120	10M	10M	6692K	**6685K**
	pack	**29M**	31M	27M	**25M**
	pack_d	17M	17M	14M	**13M**
rcpsp task	bl	3098K	**3015K**	2693K	**2668K**
	j120	16M	16M	12M	12M
	pack	19M	19M	15M	15M
	pack_d	34M	34M	31M	31M
general	bibd	1533K	**354K**	882K	**424K**
	knapsack	25M	**14M**	14M	**9769K**
	m-knapsack	14M	**12M**	41M	**34M**
	market_split_s	5814K	**2682K**	5181K	**2576K**
	market_split_u	915K	**419K**	964K	**573K**
	radiation	10M	10M	9858K	**9846K**
	road-cons	92M	92M	116M	**114M**
	trivial_unsat	10K	**32.0**	23K	**31.0**
	vrp	432M	**219M**	10M	**5599K**

Table 6.13: A comparison of the number of propagations between two variations of the solver. One solver uses raw linear extended resolution while the other de-duplicates all possible extended literals.

Removing fully duplicated predicates only affects nogood quality in certain scenarios. The risk of perfectly duplicated predicates to enter the nogood can be very low. Table 6.14 shows us almost no variation in the conflict counts for most problems. Only in benchmarks where the equality constraint is very prevalent, such as bibd and market split, is there a noticeable decrease. This suggest that if a problem mainly consists of linear equalities there is a potential to improve solver effectiveness but it in general has very little effect. Especially the usual randomness involved in propagation ordering makes it difficult to mark this as a noteworthy achievement. Note that by removing duplication the trivial unsatisfiable benchmark requires some degree of solving before we can conclude that it is unsatisfiable.

category	config suite	seq	seq-dedup	tot	tot-dedup
rcpsp time	bl	8434	**8005**	**6375**	6463
	j120	**2014**	2018	**1858**	1866
	pack	31K	31K	28K	28K
	pack_d	**886**	888	864	864
rcpsp task	bl	10K	10K	**9470**	9513
	j120	13K	13K	11K	11K
	pack	**82K**	83K	73K	**72K**
	pack_d	120K	**119K**	119K	**118K**
general	bibd	401	**400**	319	**317**
	knapsack	12K	12K	35K	35K
	m-knapsack	14K	14K	127K	**121K**
	market_split_s	25K	25K	28K	**25K**
	market_split_u	6739	**6390**	8061	**7585**
	radiation	15K	15K	16K	16K
	road-cons	**3251**	3268	5237	**5139**
	trivial_unsat	70.90	**0.0**	280	**0.0**
	vrp	23K	**22K**	**14K**	15K

Table 6.14: A comparison of the number of conflicts between two variations of the solver. One solver uses raw linear extended resolution while the other de-duplicates all possible extended literals.

Removing the duplicate predicates affects VSIDS. Initially when the solver would come across the duplicate predicates it would increment only one of them in the VSIDS procedure. With the removal of the now duplicates it means that those specific variables are more likely to be bumped higher. Again we note that VSIDS is a highly erratic method so the results in table 6.15 can either be in correlation with the removal of the duplicates or due to VSIDS' random initialization being different. Benchmarks such as market split and vrp show improvements suggesting that improving VSIDS' activity metric this way can be beneficial. While this also suggests that these benchmarks therefore benefit from making more decisions on extended resolution predicates it cannot be concluded definitively.

category	config: VSIDS suite	seq	seq-dedup	tot	tot-dedup
rcpsp time	bl	4000	**3913**	**3407**	3625
	j120	697	**660**	**229**	232
	pack	**13K**	14K	**11K**	12K
	pack_d	252	**172**	170	**112**
rcpsp task	bl	**6360**	7748	**6248**	6474
	j120	8491	**8164**	4646	**4170**
	pack	46K	**40K**	43K	**38K**
	pack_d	41K	**37K**	**37K**	42K
general	bibd	640	**366**	**332**	2166
	m-knapsack	**105K**	106K	**65K**	81K
	market_split_s	**21K**	24K	41K	**30K**
	market_split_u	12K	**9155**	11K	**8477**
	radiation	46K	**21K**	31K	**27K**
	road-cons	1522	**545**	852	**657**
	trivial_unsat	2408	**0.0**	389	**0.0**
	vrp	57K	**29K**	7451	**3646**

Table 6.15: A comparison of the number of conflicts between two variations of the solver. One solver uses raw linear extended resolution while the other de-duplicates all possible extended literals. Both solvers utilized the VSIDS technique.

6.2.6. Overlap

This section will analyse the effect of the risk of overlap. Which is when a learned nogood contains two predicates that each cover a respective partial sum yet those partial sums have variables in common. We seek to answer to what degree overlap is present and its effect on a solvers performance.

To reiterate on the statistics measured. In order to asses if this overlap has taken place a nogood is first split up in its lower and upper bound predicates. We mark all the variables present for each partition. We then decompose any extended literals down to their leaves, marking any variables that would otherwise enter the nogood and counting any duplicate flags. From this we obtain whether or not there was any overlap present and the severity of this presence. In some cases this overlap is necessary to properly explain the 1UIP yet this would still be classified as overlap.

Overlap is a very present phenomenon when using linear extended resolution. A direct implementation of linear extended resolution has no way of detecting when two partial sums overlap nor any means of rectifying such an issue. As overlap is therefore an inherent risk it can be present in large numbers. Table 6.16 presents us with this evidence in the first set of columns showcasing that almost every benchmark has a lot of nogoods which contain some form of overlap. The threat comes from the fact that nogoods which contain overlap therefore also contain duplication to some degree. Duplication is bad as it can hinder the nogood from propagating. We see this intuition reflected back in table 6.16. As the %Prop-dup column is typically lower than the %NG-dup column. This means that a nogood with duplication is less likely to propagate than a nogood without. The metrics for overlap, the last set of columns, indicate the severity of the overlap if a nogood were to be fully decomposed as to remove extended resolution. Here we see that in some benchmarks more than half of all predicates could be semantically removed were the nogood decomposed. We also see that both encodings fall victim to this overlap metric, while in some cases the presence of it may be different both encodings can attain very high scores in all duplication metrics. We can therefore conclude that it is duplication which poses a realistic not seen before risk when implementing linear extended resolution.

		%NG-dup		%Prop-dup		%OverlapPred		RawOverlap	
	config	seq	tot	seq	tot	seq	tot	seq	tot
category	suite								
rcpsp time	bl	26.7	27.5	1.95	2.18	3.42	2.22	4.77	2.84
	j120	31.8	32.6	0.559	0.585	5.94	3.39	8.91	4.87
	pack	6.74	4.81	0.523	0.489	1.10	0.535	1.79	0.846
	pack_d	2.76	2.14	0.019	0.009	0.206	0.101	0.525	0.303
rcpsp task	bl	1.02	0.404	0.053	0.029	0.106	0.035	0.229	0.062
	j120	5.36	4.32	0.100	0.079	0.713	0.193	1.71	0.387
	pack	1.97	0.697	0.353	0.106	0.392	0.049	0.665	0.069
	pack_d	0.542	0.454	0.082	0.071	0.225	0.182	1.34	1.07
general	bibd	5.36	10.5	0.017	0.067	0.936	0.777	1.05	0.587
	knapsack	2.55	62.4	0.777	55.6	0.855	16.6	1.23	16.1
	m-knapsack	85.4	79.9	37.6	26.2	45.0	14.0	53.5	16.3
	market_split_s	95.6	87.9	40.9	19.8	55.1	41.1	58.3	43.1
	market_split_u	95.6	87.9	37.8	22.5	54.6	38.5	57.8	39.7
	radiation	78.9	68.2	40.4	32.8	16.1	10.4	16.4	10.0
	road-cons	65.2	52.3	49.6	26.0	11.9	2.57	14.8	2.37
	trivial_unsat	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	vrp	10.1	10.6	56.4	28.6	0.523	0.173	0.474	0.183

Table 6.16: A comparison of overlap metrics between the two different encodings. The first two columns identify how often overlap is present in a learned nogood and in a propagating nogood respectively. The latter two columns identify how severe the overlap is per nogood and over all nogoods respectively.

Overlap does not mean linear extended resolution does not work. The strength of being able to solve exponential work problems and the improved generality provided by extended resolution can be greater than the risk introduced. We have seen for multiple benchmarks that we can achieve orders of magnitude in performance uplifts. Section 6.2.6 shows a scatter plot indicating as such. Here we see that a high presence of overlap within the nogoods does not guarantee worse performance. And that while there does appear some pattern of the larger dots being above the smaller dots they are are still very far below the center line. In this scenario the advantage of utilizing extended resolution was more beneficial than the risk of duplication.

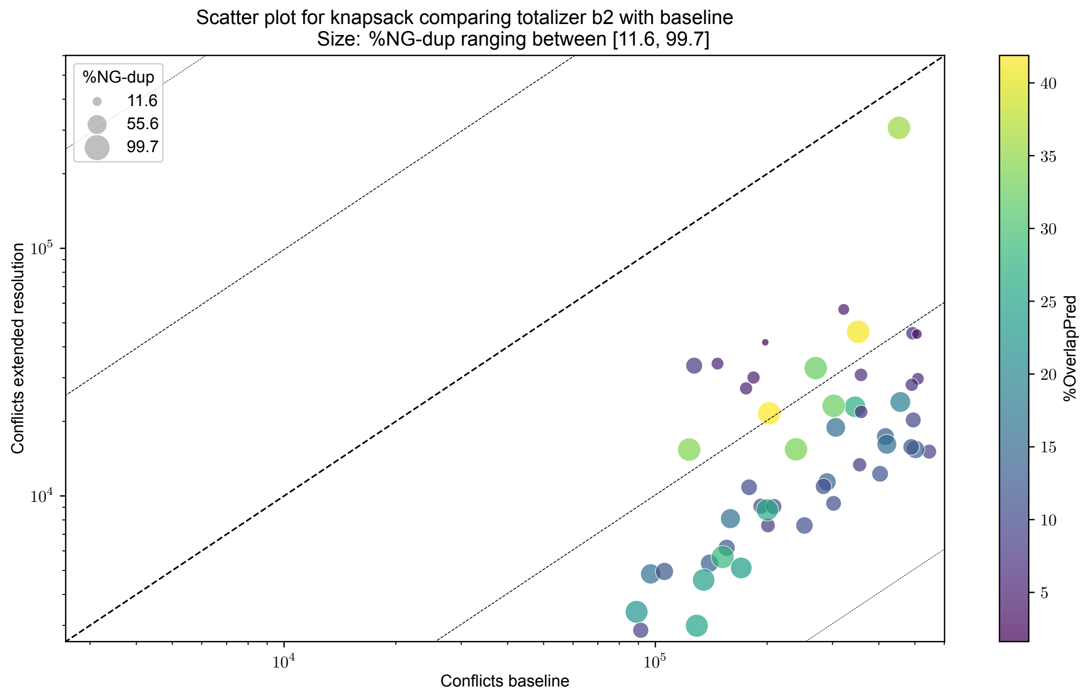


Figure 6.8: Conflicts scatter plot between seq m1 and baseline for knapsack 400. Size depicts the %NG-dup metric and colour indicates the amount of overlap within those nogoods.

Problems that stand to benefit less from extended resolution can be significantly affected by this overlap risk. If the problem to solve does not obtain much benefit from extended resolution then extended resolution can still create the risk of partially disabling the nogood via duplication. Figures 6.9 to 6.12 all indicate that overlap is a possible reason why the RCPSP benchmarks under perform. Note that larger and more brightly Coloured dots tend to stray further from the center line. Indicating that it is only those instances where overlap is high that perform significantly worse.

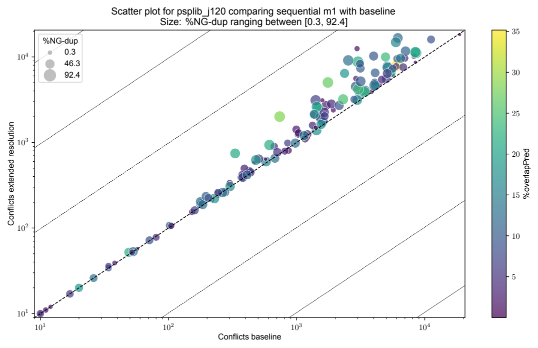


Figure 6.9: Conflicts scatter plot between seq m1 and baseline for j120 time. Size depicts the %NG-dup metric and colour indicates the amount of overlap within those nogoods.

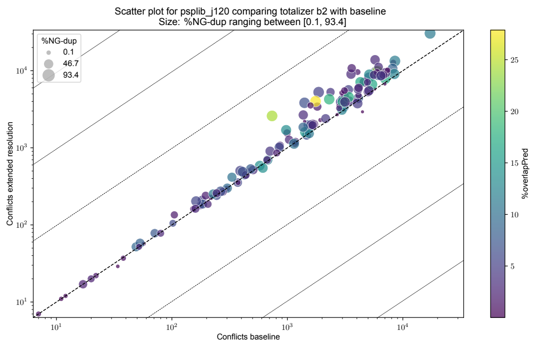


Figure 6.10: Conflicts scatter plot between tot b2 and baseline for j120 time. Size depicts the %NG-dup metric and colour indicates the amount of overlap within those nogoods.

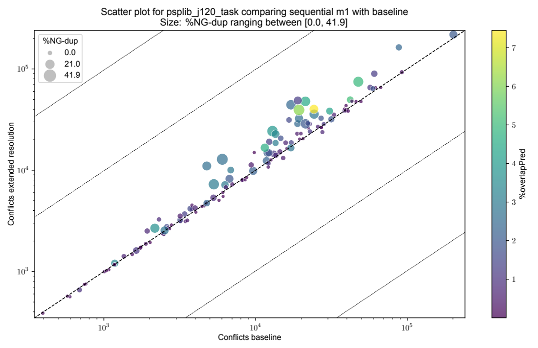


Figure 6.11: Conflicts scatter plot between seq m1 and baseline for j120 task. Size depicts the %NG-dup metric and colour indicates the amount of overlap within those nogoods.

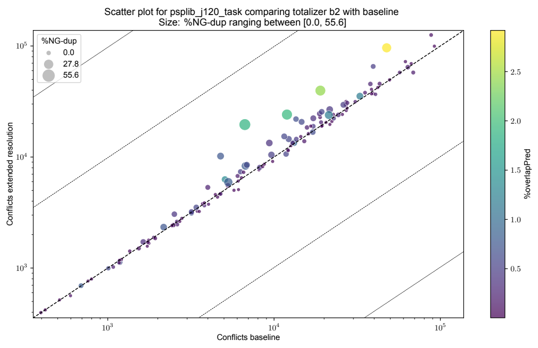


Figure 6.12: Conflicts scatter plot between tot b2 and baseline for j120 task. Size depicts the %NG-dup metric and colour indicates the amount of overlap within those nogoods.

6.3. Cumulative Literal Results

We now turn to the performance of the cumulative literal. The following comparisons will start of by comparing the presence of extended resolution within the nogoods and nogood propagations. We then compare the number of conflicts to see any bottom line performances. Finally we may compare some other metrics as a justification for our observations based on our conflict information. The subsections will first compare extended resolution directly out of the box on the MiniZinc RCPSP model. We then utilize a different search annotation to discuss its effects. VSIDS will then be utilized as a means of thoroughly activating the new technique. We conclude the section with sideline observations made during this research and a short analysis on whether improper minimization is a potential risk.

6.3.1. Out Of The Box Comparison

We first compare the cumulative literal directly out of the box. This means we utilize the default search annotation provided by the MiniZinc model and compare any impact extended resolution may have for all three explanation types.

The addition of the cumulative literal initially has very little effect on the bottom line number of conflicts found. This is rather interesting as we would expect some degree of variance to be introduced, either positive or negative. While the low presence in nogoods would explain such a phenomenon we would expect to see some pattern. However according to figure 6.13 such a pattern does not exist, and while not shown here this is the case for all other combinations of explanation and benchmark. Instead every instance is solved with about the same number of conflicts. Agnostic relative to the presence of extended resolution in the nogoods. Table 6.17 concurs with this pattern and shows that there are very few differences in the aggregate.

explanation	naive		bigstep		pointwise	
variant	basic	ext	basic	ext	basic	ext
suite						
CV	121K	**120K**	95K	**94K**	92K	**91K**
bl	7392	**7245**	3693	**3635**	3326	**3313**
j120	31K	31K	15K	15K	15K	15K
j90	10K	10K	4011	**3960**	3601	**3571**
pack	73K	**72K**	54K	54K	52K	52K
pack_d	106K	**105K**	39K	39K	38K	38K

Table 6.17: A comparison of the number of conflicts encountered on average between the baseline implementation and the extended implementation for all three main line explanation types.

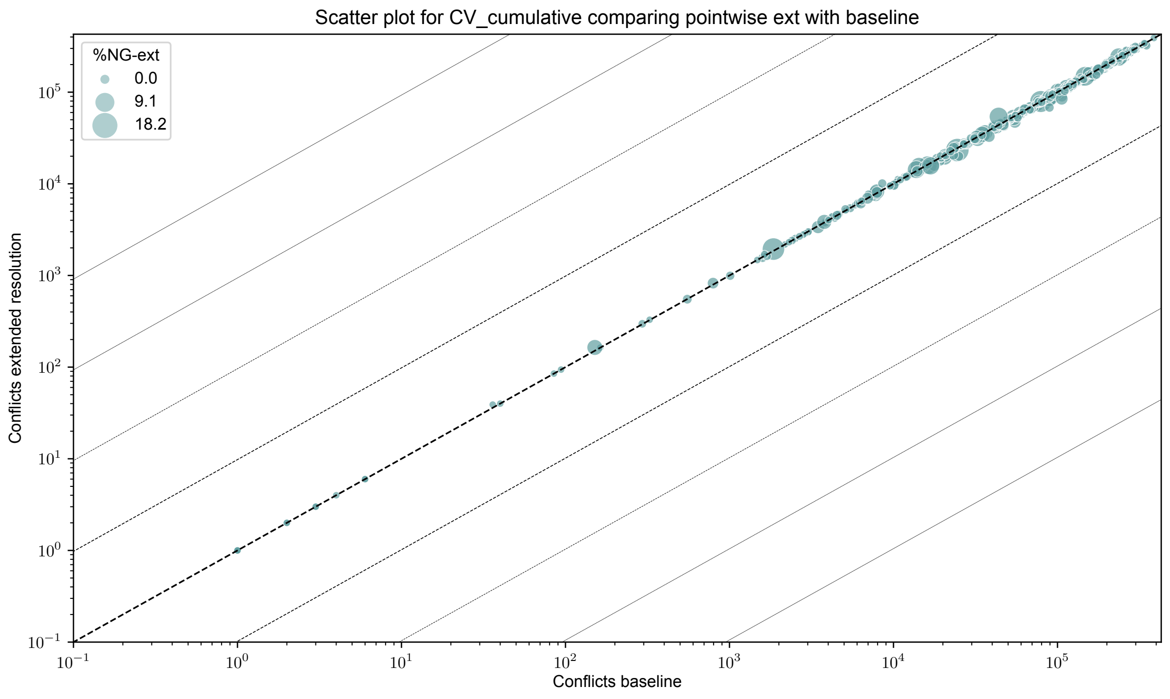


Figure 6.13: A comparison of the number of conflicts encountered per instance for the CV dataset utilizing the pointwise explanation.

The cumulative literal is not capable of obtaining a high presence within the nogoods. As we have so few secondary propagations there are also way fewer opportunities for the cumulative literal to enter a nogood. While J120 is an outlier we typically should not expect more than 5% of our nogoods to contain extended resolution as per table 6.18. The result of this is also that few of our propagations actually occur on nogoods with extended resolution at all. While we will not make any direct conclusions we see that in this out of the box scenario we are less likely to propagate on nogoods with the cumulative literal. Suggesting that either the cumulative literal does not help improve the strength of the learned nogood, the class of nogoods where the cumulative literal makes an appearance is already a weak class of nogoods that do not see much propagation, or we only propagate on such nogoods once but their asserting level is very early in the search tree.

config	naive		bigstep		pointwise	
	%NG-ext	%Prop-ext	%NG-ext	%Prop-ext	%NG-ext	%Prop-ext
suite						
CV	0.957	1.24	1.11	1.05	1.12	1.45
bl	5.03	8.41	5.44	6.54	7.49	10.0
j120	18.9	13.9	15.0	9.71	20.5	15.0
j90	7.67	5.70	6.11	3.44	9.39	5.91
pack	2.26	1.67	2.19	1.58	1.83	1.68
pack_d	5.32	4.15	3.89	2.40	7.83	4.87

Table 6.18: A comparison of extended resolution presence within the nogoods. %NG-ext denotes how often a nogood contains extended resolution whereas %Prop-ext denotes how often a nogood with extended resolution propagates.

There are very few secondary propagations made by our GEQ and LT propagators. This makes sense as out of the box the MiniZinc model uses an assigning value selection strategy, indomain min. This means that when timetable finds a profile it will mainly consist of tasks that are already fixed. By having these fixed profiles there is no need to propagate utilizing the GEQ and LT propagators. The only time when these propagators do start to see action is when, due to nogood learning, we get profiles of which some tasks are yet to be fixed. We see this lack of secondary propagations reflected back in table 6.19 as almost none of the propagations performed were made by these new propagators. Note especially the lack of LT propagations. Those can only occur if we learn, through nogood propagation, that a cumulative literal must be set to false, yet this rarely happens, suggesting a low utilization of the cumulative literal in general.

config	naive		bigstep		pointwise	
	%GEQ	%LT	%GEQ	%LT	%GEQ	%LT
suite						
CV	0.021	0.001	0.027	0.001	0.041	0.001
bl	0.162	0.004	0.190	0.003	0.195	0.004
j120	0.103	0.001	0.119	0.001	0.121	0.001
j90	0.054	0.000	0.063	0.001	0.065	0.001
pack	0.078	0.000	0.089	0.000	0.052	0.000
pack_d	0.171	0.001	0.155	0.000	0.193	0.000

Table 6.19: The number of propagations performed by the GEQ and LT propagators relative to the total amount of propagations.

6.3.2. Indomain Split

Next up we see if utilizing a bounding search strategy works better than an assigning one. For this we change the search annotation from indomain min to indomain split. Where indomain split bisects the domain and retains the lower half as possible values. This way we attempt to introduce more secondary GEQ propagations to help improve cumulative literal presence on the trail.

Switching the value selection strategy to a bounding approach has no significant effect on the number of conflicts found. This is demonstrated in table 6.20. Comparing the difference in conflicts found between the baseline and a version of Pumpkin which uses the cumulative literal we see almost no difference. More often than not a small increase is noted. While not shown here all possible scatter-plots again show a similar pattern to figure 6.13.

explanation	naive		bigstep		pointwise	
variant	basic	ext	basic	ext	basic	ext
suite						
CV	**86K**	88K	**66K**	68K	**64K**	65K
bl	5167	**4972**	2825	**2743**	2722	**2686**
j120	23K	23K	12K	12K	11K	11K
j90	9928	**9762**	**3867**	3870	**3606**	3659
pack	47K	47K	39K	39K	39K	39K
pack_d	88K	**86K**	32K	32K	32K	32K

Table 6.20: A comparison in the average number of conflicts found. A comparison between the baseline implementation and the one using extended resolution, both are using the indomain split annotation.

Utilizing a bound value selection strategy does have the potential of introducing more cumulative literals into the nogood yet nogood generality remains unaffected. We mainly see increase in extended resolution presence in nogoods consistently as shown in table 6.21. However, even with the increase of such presence no such differences were present in the conflict relation. This could either indicate that this alternative search strategy may still be blocking us, nogood presence might still be too low after all, or that the cumulative literal does not provide the sought after improvement irrelevant to the solver configuration.

explanation	naive		bigstep		pointwise	
variant	assign	bound	assign	bound	assign	bound
suite						
CV	0.928	**6.82**	1.06	**7.46**	1.06	**7.23**
bl	5.03	**6.70**	5.44	**6.70**	7.49	**8.27**
j120	18.9	**20.8**	15.0	**16.3**	20.5	**20.6**
j90	7.66	**8.33**	6.11	**6.35**	**9.39**	8.80
pack	2.27	**5.17**	2.19	**5.15**	1.82	**4.31**
pack_d	5.32	**8.06**	3.89	**9.21**	7.83	**8.56**

Table 6.21: A comparison of extended resolution presence within the nogoods defined as %NG-ext and denoting how often a nogood contains any form of extended resolution. Comparing between the indomain_min and indomain_split annotations denoted as assign and bound respectively.

The increase of extended resolution presence in the nogoods can be explained through an increase of secondary propagations. By shrinking the domains stepwise instead of directly assigning a variable we create timetable profiles of a small width which can still broaden. This means that upon the next decision its the responsibility of the GEQ propagator to make sure tasks are shifted properly when the domain split occurs. We see this reflected in table 6.22. Here we see that while some explanations only have small fluctuations there are those where a significant increase of secondary propagations occurs, e.g. pack_d. Therefore the more varied the search strategy the more secondary propagations will take place which in turn helps these literals to become present in the nogoods. Do take of note that an increase in secondary propagations does not mean a proportional increase of nogood presence must take place but that rather a minimum of secondary propagations is required before nogood presence occurs.

explanation		naive		bigstep		pointwise
variant	assign	bound	assign	bound	assign	bound
suite						
CV	0.022	**0.103**	0.027	**0.109**	0.040	**0.125**
bl	0.167	**0.309**	0.193	**0.308**	0.199	**0.296**
j120	0.104	**0.152**	0.121	**0.161**	0.123	**0.163**
j90	0.055	**0.081**	0.064	**0.085**	0.066	**0.089**
pack	0.079	**0.164**	0.090	**0.156**	0.053	**0.138**
pack_d	0.172	**0.384**	0.156	**0.459**	0.193	**0.420**

Table 6.22: A comparison of the percentage of propagations performed by the GEQ and LT propagators combined relative to the total number of propagations. Comparing between the indomain_min and indomain_split annotations denoted as assign and bound respectively.

6.3.3. VSIDS

We now test if allowing our search method to make decisions on the cumulative literal is potentially beneficial and if a more erratic search strategy can improve cumulative literal presence. We aim to answer conclusively if the cumulative literal, as it is current implemented, is beneficial for nogood quality. And if VSIDS making decisions on the new literals is beneficial.

VSIDS seems unable to properly exploit the cumulative literal to achieve significant improvements. While there are outlier benchmarks suggesting a possibility of improvement most benchmarks seem to increase the number of conflicts encountered as per table 6.23. While we have seen with previous search strategies that adding the cumulative literal did not change nogood quality in any significant way we also suffered from low presence of the technique in the solving process. We therefore go into two possible scenarios. We may hypothesize that the failure to improve stems from poor decision making by VSIDS. Or alternatively that nogood quality is not increased even when the presence of the technique is high. It is however clear that in most cases a consistent improvement of performance is not achieved.

explanation		naive		bigstep		pointwise
variant	basic	ext	basic	ext	basic	ext
suite						
CV	31K	**29K**	29K	**28K**	28K	**27K**
bl	3871	**3607**	**2600**	3135	**2561**	4017
j120	**2969**	5328	**2715**	4550	**2676**	4375
j90	**1934**	3312	**2070**	2788	**2234**	2831
pack	**19K**	20K	**17K**	19K	**15K**	19K
pack_d	18K	**14K**	**10K**	13K	**11K**	14K

Table 6.23: The average number of conflicts encountered. Comparing between the baseline and a solver utilizing extended resolution when using VSIDS

We hypothesized that VSIDS made poor decisions. However, VSIDS is not overly likely to decide upon the cumulative literals in the first place. Note that any explanation utilizing the cumulative literal also post the lower bounds of all tasks in the profile. Therefore the cumulative literal has a lot of competition for VSIDS even before a potential 1UIP is formed as it is always posted along side another specific set of variables. Combining this with the fact that more variables have been added to VSIDS increasing the amount of chatter in its activity store it becomes clear that VSIDS may not make any monumentally different decisions. The result is that VSIDS is unlikely to make decisions on the cumulative literal and that decisions made on the cumulative literal are perhaps not as impactful as initially idealized.Table 6.24 corroborates the frequency of decisions on extended resolution showing that around 10% - 15% of decisions are made on extended resolution.

config suite	naive-ext	bigstep-ext	pointwise-ext
CV	12.6	11.4	11.5
bl	23.5	20.9	21.0
j120	20.0	17.2	17.3
j90	10.6	9.44	9.41
pack	15.5	13.0	12.1
pack_d	9.88	9.94	9.42

Table 6.24: The number of decisions made on extended resolution predicates relative to the total amount of decisions when utilizing VSIDS.

To help cement our view that decisions made on extended resolution do not contribute to an effective search procedure as it is currently implemented we view figure 6.14. Here we can clearly see that the spread of instance conflict scores, made significant in colour, has no relation to the amount of decisions made on extended resolution. One observation we do take away from this is that instances which make decisions on extended resolution stray further from the center line. This makes sense as those would be the instances most dissimilar to the original solving traces and hence would form the pack of outliers.

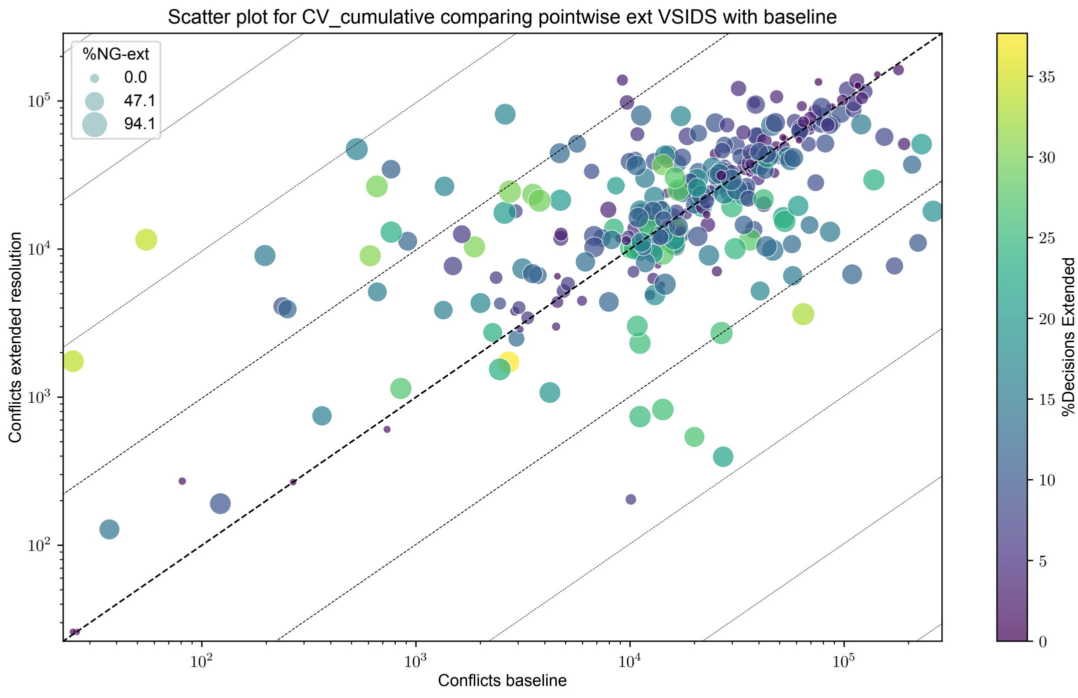


Figure 6.14: Conflicts scatter plot between extended resolution and the baseline for the pointwise explanation on the CV dataset, utilizing VSIDS search.

The search strategy can help with exploiting extended resolution. A challenge for the well known search order of in domain min is managing to introduce the cumulative literals into nogoods. Where with linear extended resolution we would prefer a very structured search approach the opposite is true for the cumulative literal. The more erratic the search the more likely we are to create incremental profiles which introduces more secondary propagations. We see this reflected in table 6.25 showcasing the high presence of cumulative literals in both nogoods and propagations.

The cumulative literal is not effective within the learned nogoods. With the cumulative literals now present in a significant number of the nogoods we can properly determine their effectiveness by seeing if clauses with the cumulative literal tend to propagate more. Table 6.25 shows the difference between the amount of nogoods containing a cumulative literal and how often the nogood propagator propagates on a nogood with the cumulative literal. Here we see that more likely than not a nogood that contains the cumulative literal on average is not that likely to be used for propagation. To elaborate, if we saw that 10% of all nogoods contain extended resolution yet are responsible for 20% of propagations we would conclude that those nogoods are more performant. It is still possible that nogoods with the cumulative literal are typically already part of a poor class of nogoods yet the consistency by which they lose out in table 6.25 makes this a weak claim. It is in fact more likely that the cumulative literal provides very little benefit to a nogoods strength. This can also be viewed in figure 6.14 where the size of the dots indicates how many of the nogoods contain extended resolution and there is no discernible pattern. An argument could even be made that due to the lower number of propagations that contain extended resolution whether or not the cumulative literal has a negative effect on nogood quality.

config	naive-ext		bigstep-ext		pointwise-ext	
	%NG-ext	%Prop-ext	%NG-ext	%Prop-ext	%NG-ext	%Prop-ext
suite						
CV	50.1	29.3	49.3	28.8	50.7	29.2
bl	70.0	75.6	67.7	69.0	68.4	70.4
j120	81.9	61.8	80.3	59.8	80.1	60.2
j90	63.8	53.5	63.4	52.3	61.4	50.6
pack	52.6	39.5	46.9	35.1	50.8	39.4
pack_d	57.3	43.0	59.9	43.6	59.0	44.7

Table 6.25: A comparison of cumulative literal presence in the nogoods depicted by %NG-ext denoting how many of the nogoods contain extended resolution and %Prop-ext denoting how many nogood propagations were conducted on nogoods with nogood propagations. Run using the VSIDS method.

The amount of conflicts found is inconsistent with the LBD scoring. We typically utilize LBD as a means of comparing no-good quality. Therefore we expect that in cases of similar performance LBD too remains similar. Typically, although not always, if we see a lower LBD score we expect fewer conflicts. This is not necessarily the case as can be seen by comparing j120 and j90 between tables 6.23 and 6.26 which boast a lower LBD score with extended resolution but fail to achieve the better performance. Our main argument as to why this happens is that when discovering the extra conflicts it learns much smaller clauses with lower LBD scores to leave those conflicts again, thus dragging down the LBD average. We presume the nogoods learned from these conflicts are mainly relevant deeper in the search tree, irrelevant for solving the problem, or occur near the end of the solving stage. Otherwise we would expect them to help prune the search space more effectively and in turn fewer conflicts to arise.

explanation	naive		bigstep		pointwise	
variant	basic	ext	basic	ext	basic	ext
suite						
CV	8.43	**8.27**	8.96	**8.63**	9.00	**8.74**
bl	4.17	**4.09**	4.16	**4.03**	4.47	**4.22**
j120	9.95	**5.94**	11.9	**6.48**	12.2	**6.58**
j90	7.26	**4.51**	8.61	**5.03**	8.81	**5.15**
pack	8.09	**7.88**	8.74	**7.84**	8.80	**8.06**
pack_d	5.95	**5.10**	6.04	**5.48**	6.06	**5.45**

Table 6.26: A comparison of nogood quality via the LBD scoring metric. Comparing between the baseline and a solver utilizing extended resolution when using VSIDS.

The cumulative literal does not introduce any direct means to help shrink nogood size. The initial idea was that regular timetable propagations include a lot of lower and upper bounds of all tasks involved. By utilizing the cumulative literal those explanations can now be made by utilizing only one of the type of bounds and the cumulative literal, as the other type of bound is abstracted behind the cumulative literal. With these more efficient propagation explanations made by the GEQ and Lt propagators a smaller nogood size was expected. However, ignoring the rows with the LBD abnormalities from table 6.26 j90 and j120, we see no such change occur in table 6.27. It therefore appears as if the cumulative literal only replaces a single predicate within a nogood. This may also indicate that the nogoods the cumulative literal ends up in are already too specific for it to achieve any significant added benefit.

explanation	naive		bigstep		pointwise	
variant	basic	ext	basic	ext	basic	ext
suite						
CV	15.1	**14.8**	16.1	**15.6**	16.4	**15.9**
bl	7.03	**6.93**	7.13	**6.98**	7.68	**7.34**
j120	14.4	**8.98**	17.3	9.87	17.8	9.97
j90	10.2	**6.65**	12.3	**7.53**	12.7	**7.67**
pack	13.5	**13.2**	14.6	**13.5**	14.5	**13.8**
pack_d	11.0	**9.60**	11.1	**10.9**	11.3	**10.6**

Table 6.27: A comparison of average nogood size. Comparing between the baseline and a solver utilizing extended resolution when using VSIDS.

6.3.4. Minimization.

We have seen that not having any minimization strategies for linear extended resolution actually introduced a weakness to the technique. In this section we will discuss if the same issue is present for the cumulative literal.

There is potentially some level of partial duplication present. One of the metrics we were able to obtain was how often it would occur that a nogood would contain both a cumulative literal and at least one set of bounds, both lower and upper, describing a start variable from a task in the profile of that cumulative literal. As a typical trace is not formed in a neat lab environment where the model consists of only a single cumulative constraint and no other constraints at play it is very much possible for such a scenario to occur where the variables initially abstracted away behind a cumulative literal still manage to enter the nogood via some alternate route. Table 6.28 shows us that this is a well represented risk within the nogoods that do contain a cumulative literal, around 50% of all nogoods with extended resolution contain such a form of duplication. We can also see how much more unlikely such a nogood was to propagate, typically propagating only 60% as likely. As this is consistent across all benchmarks it would lead us to believe the duplication is detrimental, but not conclusively yet.

config: free		naive-ext		bigstep-ext		pointwise-ext
	%NG-dup	%Prop-dup	%NG-dup	%Prop-dup	%NG-dup	%Prop-dup
suite						
CV	45.7	50.2	50.1	65.6	51.0	65.8
bl	25.3	49.2	27.1	73.1	27.1	62.6
j120	36.0	37.6	42.0	49.4	42.3	58.4
j90	19.4	22.4	23.9	28.5	23.1	29.1
pack	46.5	53.4	54.8	65.7	53.0	74.6
pack_d	44.2	52.1	43.7	69.9	44.0	71.6

Table 6.28: A comparison between nogoods that contain extended resolution contained some degree of duplication. The other columns describes how much more likely nogoods with duplication were to propagate compared to their non duplicate counterparts.

While the metric on its own is perhaps too rough we can still take a look at some specific cases to conclude that this form of duplication is detrimental. For this we utilize figure 6.15. Retaining only those instances where the likeliness of nogoods containing duplication to propagate is $\geqslant 90\%$ relative to the propagation of all nogoods containing extended resolution, normalized by the %NG factors we see a slight pattern. Namely that our outliers tend to shift to the lower right quadrant indicating scenarios beneficial to extended resolution. Here we see that on instances where duplication, in the aggregate, has no severe effect on propagation likelihood it is extended resolution which performs better. While this was initially performed on a free search benchmark, making exact claims about where the performance comes from difficult, pruning this way does remove almost all instances crossing the upper magnitude boundary at $x = 10y$ but retains most of the instances at the lower magnitude boundary of $x = 0.1y$. We therefore conclude that partial duplication with respect to the cumulative literal is a detrimental effect to the nogood quality and thus a solvers performance.

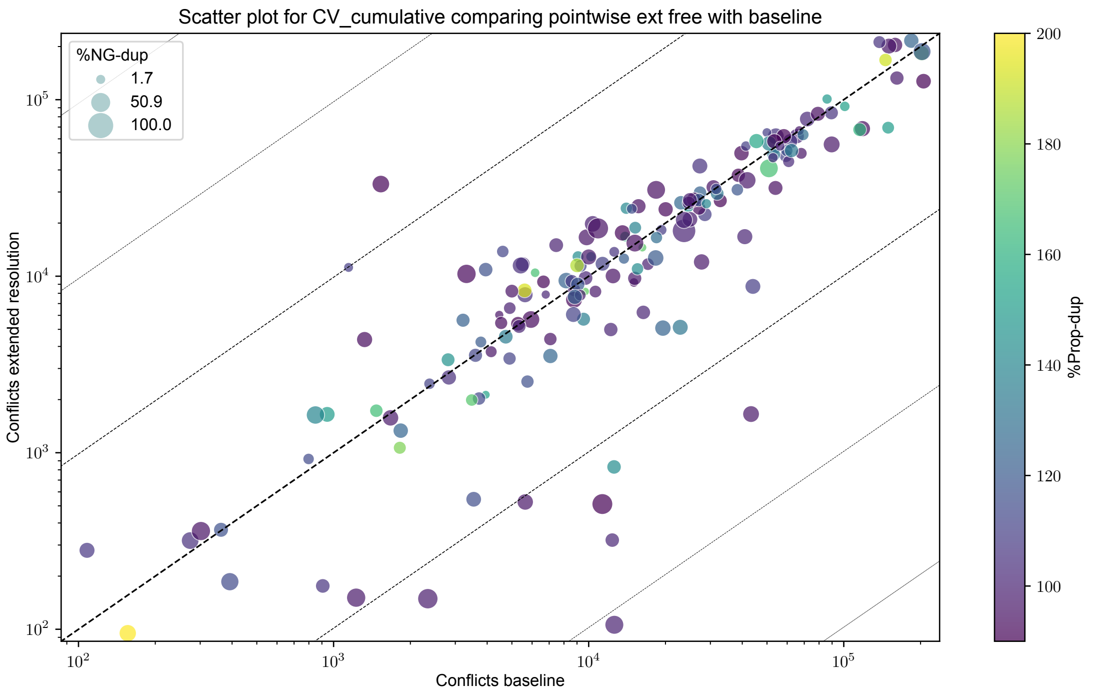


Figure 6.15: Conflicts scatter plot between extended resolution and the baseline for the naive, bigstep, and pointwise explanation on the CV dataset, utilizing free search. Instances where %Prop-dup-ext exceeded the $[90 \ldots 200]$ bounds were pruned.

6.3.5. Free Search

Thus-far we have seen that the cumulative literal poses almost no benefit. Before discarding the concept entirely we wish to share some extra insights that were obtained. We answer if VSIDS weightings should perhaps be changed and how many unique profiles are even considered when solving an RCPSP problem.

Free search is not capable of exploiting the cumulative literal to achieve a better performance. For the sake of brevity not all tables will be shown here as statistics regarding cumulative literal presence in no goods or decisions were tempered versions of their VSIDS counterparts. Statistics and patterns regarding decisions on the cumulative literal were again also similar yet slightly higher, $\pm 1\%$. This comes as no surprise as free search is essentially a seeded VSIDS run as the initial solution is reached quickly. The goal being to nudge VSIDS in a better direction than randomly guessing a start position. This means that free search tends to outperform both annotated search and regular VSIDS yet it has little extra impact on the search process involving the cumulative literal.

explanation	naive		bigstep		pointwise	
variant	basic	ext	basic	ext	basic	ext
suite						
CV	32K	**29K**	27K	27K	28K	**26K**
bl	**3340**	3465	**2760**	3456	**3244**	3549
j120	**6206**	8169	**5904**	6894	**5978**	6011
j90	**2899**	3921	**2731**	2802	**2520**	2907
pack	13K	**11K**	12K	**10K**	**10K**	11K
pack_d	23K	**20K**	17K	**16K**	17K	**15K**

Table 6.29: The number of conflicts encountered when utilizing free search. Comparing between the baseline and a solver utilizing extended resolution.

However, the cumulative literals potentially mark non-interesting decision variables. When digging into the BL suite data by hand for free search there were instances where the number of decisions made on cumulative literals was correlated to the increase in conflicts. We therefore ran another test with a slight variation, one where VSIDS discards the cumulative literals entirely. This means that the predicates over which the cumulative literal is an abstraction are no longer utilized for the VSIDS procedure. Table 6.30 shows us that blocking VSIDS access to the cumulative literals tends to increase performance, sometimes tremendously. Assuming the nogood strength with extended resolution is still relatively identical to that of the baseline implementation, as we have seen in the previous sections, we can therefore conclude that there may be better ways to inform VSIDS within the context of RCPSP. One argument in favour of this is that the cumulative literal tends to abstract the upper bound predicates away in most explanations, as the upper bounds are only used for determining that a mandatory part is present but it is the lower bounds that determine the eventual strength of the propagation it would make sense that pushing VSIDS more towards those lower bound variables can be beneficial.

explanation	naive		bigstep		pointwise	
variant	default	blocked	default	blocked	default	blocked
suite						
CV	31K	**27K**	28K	**25K**	29K	**25K**
bl	**3340**	3478	2760	**2194**	3244	**2468**
j120	12K	**8015**	10K	**7755**	11K	**7651**
j90	4320	**3362**	4340	**3108**	4313	**2803**
pack	15K	**12K**	13K	**10K**	11K	11K
pack_d	22K	**20K**	**15K**	17K	17K	**14K**

Table 6.30: The number of conflicts encountered when utilizing free search. Default denotes a solver without extended resolution. Blocked denotes a solver with extended resolution where VSIDS access to the cumulative literal is blocked.

To support the effectiveness of blocking VSIDS access to the cumulative literals we also have table 6.31. This table shows how many unique profiles and polarities the timetable propagator has propagated on. What we see is that by blocking VSIDS access to extended resolution the number of unique profiles also diminishes. While we do not claim that fewer unique profiles means we have an easier problem it does seem to indicate that our problem tends to be focused more on specific conflicts. It could therefore be an indication that VSIDS, by removing access, tends to search in a more concentrated area.

explanation variant suite	extended	naive blocked	extended	bigstep blocked	extended	pointwise blocked
CV	2022	**1947**	1932	**1870**	1893	**1844**
bl	1317	**1283**	1188	**1183**	1257	**1147**
j120	3114	**2239**	2873	**2188**	2584	**2340**
j90	774	**571**	576	**507**	577	**578**
pack	1863	**1849**	1724	**1611**	1700	**1315**
pack_d	915	**873**	**836**	913	**754**	1009

Table 6.31: The number of unique profiles and shift directions encountered between extended free search and free search where VSIDS was blocked access to the cumulative literal.

7 Conclusions and Future Work

Lazy Clause Generation (LCG) solvers are usually improved to provide better and more general explanations in their propagators. However, such explanations are constrained to only reason about variables already present in the problem. This prevents universally maximally general explanations from being utilized. Previous work [7, 23] has shown that introducing extra variables, known as structural extended resolution, to model relationships between variables has been very potent. This thesis therefore aims to investigate if it is possible to achieve better nogoods when solving the RCPSP problem by utilizing extended resolution to model more complex relationships.

The first four research questions considered the concept of linear inequalities and extended resolution. When comparing linear extended resolution on a broader set of benchmarks we found that a lower conflict score happens often but is not guaranteed. Initially we looked at linear extended resolution comparing the baseline solver with a sequential encoding and a totalizer encoding. We have seen that the sequential model is capable of providing the most general explanations while the totalizer model is more robust in uncontrolled environments. We wanted to investigate if linear extended resolution was a good approach for the decomposition of cumulative, here we saw that this approach did not work, most likely due to partial overlap between literals which we could not remove. Finally we asked what characterizes the quality of a nogood with linear extended resolution. We have seen that literals which cover larger parts of a linear inequality, explicitly or implicitly, are much more beneficial to have in a nogood. In addition to this we have seen that the aforementioned partial overlap is detrimental to the quality of a nogood but this effect can be overcome by the inherent strength of the technique in specific problem suites and or instances.

The last two research questions focused on the cumulative literal. Here we saw the the cumulative literal had little to no effect and that most of the effects were detrimental to the performance. The small impact was attributed to the new literal being an intermediate variable which had great difficulty being present in nogoods. Different search strategies managed to achieve extended resolution presence in the nogoods but performance remained at the same level. A deep-dive showed us that the nogood quality may be affected by another type of partial overlap. We conclude that the cumulative literal provides no benefit when used with typical RCPSP search strategies. Results achieved with VSIDS remain inconclusive but altering the VSIDS procedure has potential.

A correlation between both sections is that for both types of extended resolution, in cases were extended resolution failed to perform, there was a notion of semantic duplication. Nogoods often contained both the extended resolution predicate and some part of the decomposition of that extended resolution predicate. To our knowledge this was not discovered before and creates new paths to explore. These findings tell us that extended resolution requires more work than previously thought. While a direct implementation can provide improvements to reach its full potential more research has to go into the infrastructure surrounding extended resolution before an accurate assessment can be made.

7.1. Future Work

While the initial results of this work did not show an increase of performance for the RCPSP benchmarks the work is far from done.

We have shown that extended resolution suffers a great weakness when it comes to potential duplication, with the increase in different types of literals come different semantics that we fail to minimize properly. One must consider solving this duplication problem to be highly relevant as we have shown that even on benchmarks were performance was high duplication was thoroughly present. The main goal of future research should be to create new minimization techniques that support nogoods containing extended resolution.

The work of this research was mainly theoretical: We looked at conflicts, nogood quality, and extended resolution presence. Anybody wanting to apply this in a practical setting will mainly care about runtime. As the engineering efforts here focused largely on the theoretical benefit these were not invested in practical computation efficiency. For linear extended resolution this meant that the consistency of the encodings was enforced inefficiently by sweeping the data structures. The notification system to help prevent enqueuement of propagators still at fixpoint was not utilized, nor were any trailed auxiliary variables that would help speed-up propagation. Alternatively lazy structures could be introduced to only create and retain the relevant variables when needed for resolution. How much faster extended resolution may be in practice would be a suitable goal for future research.

Not all of the questions on linear extended resolution were answered. Not every variable ordering was tested, nor was it possible to test different methods to balance the totalizer encoding. One could consider solving the first thousand conflicts, order the variables by relevancy and then create the encodings. And currently it is always the last node that is not full in both encodings. Perhaps creating a more balanced tree or shifting the incomplete nodes in the encoding can be beneficial. We therefore suggest future research to look into more configurations of these hyper-parameter to the technique.

Finally we have shown how VSIDS interacts with extended resolution. In these scenarios we have treated our extended variables as regular variables. However seeing that not all variables are created equally in the relevance histograms we may suggest alternative weightings for VSIDS. After all the extended resolution variables are typically more powerful variables to make decisions on. It would not be difficult to increase the VSIDS score of such a variable in correlation to the size of partial sum covered in the case of linear extended resolution. As the final intrigue we therefore suggest future research to look into weighting extended resolution variables in relation with VSIDS or other search strategies focusing explicitly on extended resolution variables.

References

[1] Ignasi Abío and Peter J. Stuckey. “Conflict Directed Lazy Decomposition”. en. In: *Principles and Practice of Constraint Programming*. Ed. by Michela Milano. Berlin, Heidelberg: Springer, 2012, pp. 70–85. isbn: 978-3-642-33558-7. doi: 10.1007/978-3-642-33558-7_8.

[2] Ignasi Abío and Peter J. Stuckey. “Encoding Linear Constraints into SAT”. en. In: *Principles and Practice of Constraint Programming*. Ed. by Barry O’Sullivan. Cham: Springer International Publishing, 2014, pp. 75–91. isbn: 978-3-319-10428-7. doi: 10.1007/978-3-319-10428-7_9.

[3] Ignasi Abío et al. “To Encode or to Propagate? The Best Choice for Each Constraint in SAT”. en. In: *Principles and Practice of Constraint Programming*. Ed. by Christian Schulte. Berlin, Heidelberg: Springer, 2013, pp. 97–106. isbn: 978-3-642-40627-0. doi: 10.1007/978-3-642-40627-0_10.

[4] Gilles Audemard, George Katsirelos, and Laurent Simon. “A Restriction of Extended Resolution for Clause Learning SAT Solvers”. en. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 24.1 (July 2010). Number: 1, pp. 15–20. issn: 2374-3468. doi: 10.1609/aaai.v24i1.7553. url: https://ojs.aaai.org/index.php/AAAI/article/view/7553 (visited on 11/02/2024).

[5] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*. English. Second edition. Frontiers in artificial intelligence and applications v. 336. Amsterdam: IOS Press, 2021. isbn: 978-1-64368-161-0. url: https://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=2934688 (visited on 12/05/2024).

[6] Arthur Bit-Monnot. “Enhancing Hybrid CP-SAT Search for Disjunctive Scheduling”. In: *ECAI 2023*. IOS Press, 2023, pp. 255–262. doi: 10.3233/FAIA230278. url: https://ebooks.iospress.nl/doi/10.3233/FAIA230278 (visited on 11/19/2024).

[7] Geoffrey Chu and Peter J. Stuckey. *Structure Based Extended Resolution for Constraint Programming*. arXiv:1306.4418. June 2013. doi: 10.48550/arXiv.1306.4418. url: http://arxiv.org/abs/1306.4418 (visited on 10/24/2024).

[8] José Coelho and Mario Vanhoucke. “Going to the core of hard resource-constrained project scheduling instances”. In: *Computers & Operations Research* 121 (Sept. 2020), p. 104976. issn: 0305-0548. doi: 10.1016/j.cor.2020.104976. url: https://www.sciencedirect.com/science/article/pii/S0305054820300939 (visited on 06/28/2025).

[9] Martin Davis, George Logemann, and Donald Loveland. *A machine program for theorem-proving*. eng. New York: Courant Institute of Mathematical Sciences, New York University, 1961. url: http://archive.org/details/machineprogramfo00davi (visited on 01/14/2025).

[10] Martin Davis and Hilary Putnam. “A Computing Procedure for Quantification Theory”. In: *J. ACM* 7.3 (July 1960), pp. 201–215. issn: 0004-5411. doi: 10.1145/321033.321034. url: https://dl.acm.org/doi/10.1145/321033.321034 (visited on 01/07/2025).

[11] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. “Towards a Better Understanding of the Functionality of a Conflict-Driven SAT Solver”. en. In: *Theory and Applications of Satisfiability Testing – SAT 2007*. Ed. by João Marques-Silva and Karem A. Sakallah. Berlin, Heidelberg: Springer, 2007, pp. 287–293. isbn: 978-3-540-72788-0. doi: 10.1007/978-3-540-72788-0_27.

[12] 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. Berlin, Heidelberg: Springer, 2009, pp. 352–366. isbn: 978-3-642-04244-7. doi: 10.1007/978-3-642-04244-7_29.

[13] Robert M. Haralick and Gordon L. Elliott. “Increasing tree search efficiency for constraint satisfaction problems”. In: *Artificial Intelligence* 14.3 (Oct. 1980), pp. 263–313. issn: 0004-3702. doi: 10.1016/0004-3702(80)90051-X. url: https://www.sciencedirect.com/science/article/pii/000437028090051X (visited on 01/10/2025).

[14] Jinbo Huang. “Extended clause learning”. In: *Artificial Intelligence* 174.15 (Oct. 2010), pp. 1277–1284. issn: 0004-3702. doi: 10.1016/j.artint.2010.07.008. url: https://www.sciencedirect.com/science/article/pii/S000437021000130X (visited on 11/02/2024).

[15] Jorik Jooken, Pieter Leyman, and Patrick De Causmaecker. “A new class of hard problem instances for the 0–1 knapsack problem”. In: *European Journal of Operational Research* 301.3 (Sept. 2022), pp. 841–854. issn: 0377-2217. doi: 10.1016/j.ejor.2021.12.009. url: https://www.sciencedirect.com/science/article/pii/S037722172101016X (visited on 06/28/2025).

[16] Philippe Laborie. “Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results”. In: *Artificial Intelligence* 143.2 (Feb. 2003), pp. 151–188. issn: 0004-3702. doi: 10.1016/S0004-3702(02)00362-4. url: https://www.sciencedirect.com/science/article/pii/S0004370202003624 (visited on 01/14/2025).

[17] Eduardo Lalla-Ruiz and Stefan Voß. “Improving solver performance through redundancy”. en. In: *Journal of Systems Science and Systems Engineering* 25.3 (Sept. 2016), pp. 303–325. issn: 1861-9576. doi: 10.1007/s11518-016-5301-9. url: https://doi.org/10.1007/s11518-016-5301-9 (visited on 01/10/2025).

[18] Matthew W. Moskewicz et al. “Chaff: engineering an efficient SAT solver”. In: *Proceedings of the 38th annual Design Automation Conference*. DAC ’01. New York, NY, USA: Association for Computing Machinery, June 2001, pp. 530–535. isbn: 978-1-58113-297-7. doi: 10.1145/378239.379017. url: https://dl.acm.org/doi/10.1145/378239.379017 (visited on 12/02/2024).

[19] 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.

[20] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. “Propagation via lazy clause generation”. en. In: *Constraints* 14.3 (Sept. 2009), pp. 357–391. issn: 1572-9354. doi: 10.1007/s10601-008-9064-x. url: https://doi.org/10.1007/s10601-008-9064-x (visited on 11/20/2024).

[21] Knot Pipatsrisawat and Adnan Darwiche. “A Lightweight Component Caching Scheme for Satisfiability Solvers”. en. In: *Theory and Applications of Satisfiability Testing – SAT 2007*. Ed. by João Marques-Silva and Karem A. Sakallah. Berlin, Heidelberg: Springer, 2007, pp. 294–299. isbn: 978-3-540-72788-0. doi: 10.1007/978-3-540-72788-0_28.

[22] Christian Schulte and Peter J. Stuckey. “Efficient constraint propagation engines”. In: *ACM Trans. Program. Lang. Syst.* 31.1 (Dec. 2008), 2:1–2:43. issn: 0164-0925. doi: 10.1145/1452044.1452046. url: https://dl.acm.org/doi/10.1145/1452044.1452046 (visited on 12/02/2024).

[23] Andreas Schutt and Peter Stuckey. “Explaining Producer/Consumer Constraints”. In: vol. 9892. Sept. 2016, pp. 438–454. doi: 10.1007/978-3-319-44953-1_28.

[24] Andreas Schutt et al. “Explaining the cumulative propagator”. en. In: *Constraints* 16.3 (July 2011), pp. 250–282. issn: 1572-9354. doi: 10.1007/s10601-010-9103-2. url: https://doi.org/10.1007/s10601-010-9103-2 (visited on 11/29/2024).

[25] Andreas Schutt et al. “Why Cumulative Decomposition Is Not as Bad as It Sounds”. en. In: *Principles and Practice of Constraint Programming - CP 2009*. Ed. by Ian P. Gent. Berlin, Heidelberg: Springer, 2009, pp. 746–761. isbn: 978-3-642-04244-7. doi: 10.1007/978-3-642-04244-7_58.

[26] Peter Smith. *An introduction to formal logic*. Cambridge University Press, 2003. url: https://books.google.com/books?hl=nl&lr=&id=udELAQAAQBAJ&oi=fnd&pg=PA1&dq=An+introduction+to+formal+logic&ots=RwJYxmkpeu&sig=KbeizyUeMyLepG_NcuoVgu7F014 (visited on 12/10/2024).

[27] G. S. Tseitin. “On the Complexity of Derivation in Propositional Calculus”. en. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer, 1983, pp. 466–483. isbn: 978-3-642-81955-1. doi: 10.1007/978-3-642-81955-1_28. url: https://doi.org/10.1007/978-3-642-81955-1_28 (visited on 11/27/2024).

[28] Lintao Zhang et al. “Efficient conflict driven learning in a Boolean satisfiability solver”. In: *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*. ISSN: 1092-3152. Nov. 2001, pp. 279–285. doi: `10.1109/ICCAD.2001.968634`. url: `https://ieeexplore.ieee.org/abstract/document/968634` (visited on 12/10/2024).