Applying Core-Guided Techniques to Constraint Programming

Cesar van der Poel

Applying Core-Guided Techniques to Constraint Programming

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Cesar van der Poel
born in Heemskerk, the Netherlands

Algorithmics Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Supported by the NWO/OCW,
as part of the Quantum Software
Consortium programme
(project number 024.003.037 / 3368)

Applying Core-Guided Techniques to Constraint Programming

Author: Cesar van der Poel
Student id: 4964780
Email: `C.B.vanderPoel-1@student.tudelft.nl`

Abstract

Core-guided search has been prevalent in the field of Maximum Satisfiability (MaxSAT), largely due to the application of additional techniques that improve performance. With core-guided search being recently applied to Constraint Programming (CP), the question emerges whether such additional techniques can be applied to CP as well. In this thesis, four such features are implemented: weight-aware core extraction (WCE), which extracts multiple disjoint cores, reducing core size and overhead to improve efficiency; stratification, which extracts high-weighing cores first, allowing it to give better estimates and often to require fewer cores to be found; hardening, which prunes suboptimal parts of the search space, guiding search closer to the optimal solution; and partitioning, which divides the terms in the objective function into disjoint, strongly intra-related groups, thereby causing smaller cores to be found efficiently. With the exception of hardening - which is used in conjunction with either WCE or stratification - these are individually combined with the four approaches used to apply core-guided search to CP. This evaluation results in an in-depth analysis on which features combine well with which approaches, and why, as well as the individual strengths of the approaches and features.

We observed that the variable-based approach generally perform better than the slice-based approach, with coefficient elimination increasing this difference. Hardened WCE has the largest positive effect, especially in the slice-based approach; it causes speedups in all four base approaches, resulting in an overall performance increase in three of those. The variable-based coefficient eliminating variant without additional features was able to solve the most instances; all additional features decreased performance. The current implementation of partitioning was able to improve on some key metrics, but decreased overall performance in all four cases.

Other important conclusions are that no solver strictly outperformed any other, advocating that different solvers are best suited for different problems. Secondly, the order of assumptions has been proven to cause significant changes in the performance of solvers. Finally, we concluded that considering relaxations generally helps prove unsatisfiability.

Thesis Committee:

Chair:	Dr. E. Demirović, Faculty EEMCS, TU Delft
University supervisor:	I.C.W.M. Marijnissen, Faculty EEMCS, TU Delft
Committee Member:	Dr. S.S. Chakraborty, Faculty EEMCS, TU Delft

Contents

Chapter 1

Introduction

Constraint Programming, or CP, is a valuable approach in computer science. It can be used to efficiently solve a wide range of problems, many of which are hard to solve using other methods - such as scheduling problems [20] [25]. The wide range of (global) constraints and available solvers are prime examples of reasons why this field is so versatile and impactful.

The solvers used for CP generally perform either of two approaches; local search [22] or linear search [23]. In the related field of MaxSAT, a different technique exists, called Core-Guided Search. It has for years proven to be effective and efficient [17] [24] [30]; solvers applying this technique are competitive and often outperforming linear or local search solvers in competitions, such as the MaxSAT Evaluation [34]. Recent research [18] has shown that core-guided search can also be applied to CP, with promising results.

Many competitive core-guided MaxSAT solvers make use of several additional functionalities to improve their efficiency. The solver used in [18] uses four such features in CP, of which only one is extensively evaluated. As such, it is unclear what effect the other features have on the performance of the solver. Additionally, much more features exist in MaxSAT, some of which may be valuable in CP as well; more research is needed to determine which ones can improve the performance of core-guided CP solvers.

In this thesis, we examine four techniques used in recent core-guided MaxSAT solvers; three of these have already been implemented in the solver from [18]. These features are implemented in a core-guided CP solver, and are individually evaluated on a large set of problems. This allows us to investigate the effect of these features, and to determine to what extent they are valuable additions to the core-guided CP paradigm. The main contributions compared to [18] are an individual evaluation for each of the features, and an increase in the number of instances used in the evaluation; as well as the implementation of a fourth feature: partitioning, based on [31].

This thesis is structured as follows. In chapter 2, several base techniques are discussed; an understanding of these techniques is required to better understand the information and ideas provided in this thesis. An in-depth description of the features implemented during this thesis is given by chapter 3. After this, the approaches used to evaluate these features and the experimental setup are described in chapter 4. The results of this evaluation, as well as the conclusions drawn from them, are described in chapter 5 and briefly summarised in chapter 6. Finally, future research directions are laid out in chapter 7.

Chapter 2

Preliminaries

To provide a basic understanding of the original techniques, as well as the field they are applied them to, this section describes the most important concepts used in this thesis.

SAT and MaxSAT are described in section 2.1; a rudimentary knowledge of these topics is required to understand the most important notions of core-guided search. Additionally, this knowledge is important in grasping the concepts of additional features discussed in chapter 3. Core-guided search is the root of the research performed here, as well as constraint programming. As such, it is important that these concepts are properly understood; they are extensively explained in section 2.2 and section 2.3, respectively. After this, Lazy Clause Generation (LCG) is covered in section 2.4, as this technique helps explain some of the topics covered later. Finally, the application of core-guided search to CP as it has been done previously is addressed in section 2.5. This provides the cornerstones of the research performed in this thesis, and therefore is vital to cover thoroughly.

2.1 SAT and (Partial) MaxSAT

We define a boolean variable x_k as a variable that can take on values 0 and 1. A literal l^i_j is either a boolean variable x_k, or its negated counterpart $\neg x_k$; $\neg$ is the boolean *not* operator, inverting the assignment. These literals l^i_j are combined into clauses $c_i = l^i_1 \vee l^i_2 \vee ... \vee l^i_n$, where $\vee$ is the boolean *or* operator. A clause may also be given as a set $c_i = \{l^i_1, l^i_2, ..., l^i_n\}$. These clauses c_i are in turn combined into a formula $S = c_1 \wedge c_2 \wedge ... \wedge c_m$, where $\wedge$ is the boolean *and* operator; this formula may again be given as a set $S = \{c_1, c_2, ..., c_m\}$. Formally, this results in the following identity:

$$S = \bigwedge_{c_i \in S} c_i = \bigwedge_{c_i \in S} \left(\bigvee_{l^i_j \in c_i} l^i_j \right)$$

We define set $Vars(S)$ as the set of all boolean variables x_k that are used in S, or formally:

$$Vars(S) = \{x_k \mid \exists c_i \in S \, : \, x_k \in c_i \vee \neg x_k \in c_i\}$$

Consider a mapping $V : Vars(S) \rightarrow \{0,1\}$, which we refer to as an assignment. Based on such a mapping, we can derive a value for each literal l^i_j, and in turn for each clause c_i and formula S. We extend the definition of this mapping to include these derived values:

$$V(\neg x_k) = \neg V(x_k)$$

$$V(S) = \bigwedge_{c_i \in S} V(c_i) = \bigwedge_{c_i \in S} \left(\bigvee_{l^i_j \in c_i} V(l^i_j) \right)$$

Finally, we define $\mathbb{V}$ to be the set of all possible assignments V.

The Boolean Satisfiability problem, abbreviated SAT, is the problem of determining whether there exists an assignment V, such that it satisfies a given propositional formula S:

$$\exists V \in \mathbb{V} : V(S) = 1$$

As a simple example, consider the XOR problem. The XOR problem has two variables and requires exactly one of these to be true. This formula could be expressed as follows:

$$Vars(S) = \{x_1, x_2\}$$

$$S = \{\{x_1, x_2\}, \{\neg x_1, \neg x_2\}\} = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

There are 4 possible assignments $V_i \in \mathbb{V}$.

$$V_1 = \{(x_1,0),(x_2,0)\};\ V_2 = \{(x_1,0),(x_2,1)\};\ V_3 = \{(x_1,1),(x_2,0)\};\ V_4 = \{(x_1,1),(x_2,1)\}$$

$$V_1(S) = (0 \vee 0) \wedge (\neg 0 \vee \neg 0) = 0 \wedge (1 \vee 1) = 0 \wedge 1 = 0$$

$$V_2(S) = (0 \vee 1) \wedge (\neg 0 \vee \neg 1) = 1 \wedge (1 \vee 0) = 1 \wedge 1 = 1$$

$$V_3(S) = (1 \vee 0) \wedge (\neg 1 \vee \neg 0) = 1 \wedge (0 \vee 1) = 1 \wedge 1 = 1$$

$$V_4(S) = (1 \vee 1) \wedge (\neg 1 \vee \neg 1) = 1 \wedge (0 \vee 0) = 1 \wedge 0 = 0$$

Since V_2 and V_3 result in satisfaction, S is satisfiable. These satisfying assignments are often called solutions. Most implementations proof whether such a solution V exists by finding one, which can subsequently be returned as output. If multiple solutions exist, different solvers may return different ones, based on their inner workings.

Partial MaxSAT, often referred to as MaxSAT[1] [29], is a generalisation of SAT. Rather than a single set of clauses S, this variation makes use of two sets of clauses: the hard clauses C_h, corresponding to S; and soft clauses C_s, which define an *objective* (or *cost*) function. Without loss of generality, we assume all clauses $c_i \in C_s$ to be unit clauses, i.e. consisting of a single literal. We also assume C_s and C_h to be disjoint, i.e. $C_h \cap C_s = \emptyset$. In most cases, the objective function is defined as a weighted sum of the satisfied (or, depending on the problem, violated) clauses, where each soft clause $c_i \in C_s$ corresponds to a weight w_i. The goal of MaxSAT is to find an optimal solution, i.e. an assignment which has the highest (or

[1]MaxSAT originally refers to the problem where all clauses are "soft" [21]. However, the literature generally uses this term to refer to Partial MaxSAT, as is done in this thesis.

lowest) objective value, among all assignments that fully satisfy the hard clauses C_h. In this thesis, we consider problems which minimise the total weight of violated clauses.

Consider a MaxSAT problem $\{C_h, C_s, W\}$, where W is the set of weights w_i associated with clauses $c_i \in C_s$, and the set of possible assignments $\mathbb{V}$. Formally, the goal of MaxSAT is to find assignment $V_{opt} \in \mathbb{V}$, such that the total weight of violated clauses $c_i \in C_s$ is minimised, while all hard clauses C_h are satisfied:

$$V_{opt} = \underset{V \in \mathbb{V}}{\operatorname{argmin}} \left\{ \sum_{c_i \in C_s} w_i * V(\neg c_i) \ s.t. \ V(C_h) = 1 \right\}$$

For convenience, assignments are henceforth given as a set of variables X. Given such a set X, the mapping of variables is the following:

$$V(x) = \begin{cases} 1 & x \in X \\ 0 & x \notin X \end{cases}$$

2.1.1 Incremental SAT solving [36]

When provided with a set of multiple, similar SAT problem instances which need to be solved iteratively, it is generally beneficial to reuse some of the search steps and calculations done by the solver in a previous iteration. This idea was first introduced in SATIRE [36]. Though originally created for solving several distinct yet similar problems, the practice of reusing learned clauses and the search tree has also proven beneficial in MaxSAT. Many solvers use (parts of) this approach.

2.1.2 Assumption interface [14]

A technique often used in incremental SAT solving, as well as core-guided search, is the Assumption interface. Through assumptions, the values of certain variables are assigned at the start of the search process[2], thus limiting the search space of the solver. This can be used to enable or disable certain clauses at will: by modifying c_i to $c_i \vee r_i$, assumptions on r_i determine whether the clauses is enabled. Alternatively, assumptions may be used to find mutually exclusive assignments: if a problem is unsatisfiable under a given set of assumptions, these assumptions may conflict with one another. Such conflicting subsets are referred to as cores, and are vital in core-guided search, as described in the next section.

A set of assumptions is equivalent to a partial assignment. Since this may contain unassigned variables, the aforementioned set notation used for full assignments is not sufficient. As such, the previously defined notation is extended to include negated variables where needed - indicating that the variable is assigned 0. Given a partial assignment X, this results in the following mapping:

$$V(x) = \begin{cases} 1 & x \in X \\ 0 & \neg x \in X \\ \textit{unassigned} & \text{otherwise} \end{cases}$$

[2]A minor note for assumptions in CP is that they assign variables through boolean literals, which are discussed later. Such assumptions can, and often do, take on the form $(x \leq n)$, rather than $(x = n)$.

2.2 Core-Guided Search [17]

Core-Guided Search is an approach originally developed for solving MaxSAT problems. It is based largely on the notion of a Minimum Unsatisfiable Subset, or MUS [26]. Given a set of clauses C, a MUS $C_{MUS} \subseteq C$ is an unsatisfiable set of clauses, such that $C_{MUS} \setminus \{c_i\}$ is satisfiable for any $c_i \in C_{MUS}$. Given a MaxSAT problem $\{C_h, C_s\}$, there may exist MUSes $C_{MUS} \subseteq C_h \cup C_s$; since C_h needs to be satisfied in all solutions, we generally concern ourselves with $\kappa = C_{MUS} \cap C_s$. Such a κ is called a *core*.

A solver applying core-guided search makes use of assumptions enforcing that the objective value $obj = l_{obj}$, its lower bound. These assumptions initially correspond to C_s being fully satisfied. If the problem has no solution under these assumptions, the solver extracts a (small) core κ; as mentioned before, κ contains the soft clauses of a MUS, meaning $\kappa \cup C_h$ is unsatisfiable. This core κ of conflicting soft clauses can be formally defined as follows:

$$\kappa \subseteq C_s \; s.t. \; \nexists V \; \forall c \in (C_h \cup \kappa) \; : \; V(c) = 1$$

Or, equivalently:

$$\kappa \subseteq C_s \; s.t. \; \forall V \; \exists c \in (C_h \cup \kappa) \; : \; V(c) = 0 \tag{2.1}$$

By its definition, we can see that $\kappa \cup C_h$ contains a MUS. As the assumptions in κ are necessary conditions for $obj = l_{obj}$, this core also serves as a proof that $obj > l_{obj}$.

After a core has been found, core-guided search reformulates the problem. This results in an updated lower bound, and a correspondingly updated set of assumptions. The exact reformulation approach depends on the used solver, but as an example, the approach from [17] is described. This approach uses relaxation variables, in combination with hard cardinality constraints of which the bounds do not change. Note that other solvers might have soft cardinality constraints [30], or not use cardinality constraints at all [5]; for the sake of this example and the corresponding explanation, these cases are not considered.

Given a core κ, each (soft) clause $c_i \in \kappa$ is replaced by $c'_i \leftrightarrow c_i \cup r_i$, where r_i is a newly created relaxation variable; after all $n = |\kappa|$ clauses have been replaced, and as such n relaxation variables have been created, a cardinality constraint is added to the problem to enforce that (at most) one of these variables can be assigned 1: $r_1 + r_2 + ... + r_n \leq 1$. This allows the solver to assign a single $V(r_i) = 1$, in turn allowing $V(c_i) = 0 \; s.t. \; c_i \in \kappa$. Intuitively, this changes the problem from $C_h \cup \kappa$ to $C_h \cup \kappa \setminus \{c_i\}$; when combining this information with the definition of a MUS and Equation 2.1, we see that this new problem is satisfiable if κ corresponds to at most one MUS.

Found cores may be non-minimal, i.e. contain several elements that are not present in the MUS corresponding to the core. Some algorithms have specific approaches to handle these, and even cores corresponding to multiple MUSes, as discussed in [26] and [3] respectively. However, such a specific approach is not required to ensure that such cores are handled correctly. For example, consider a non-minimal core corresponding to two disjoint MUSes $\kappa = \kappa^1_{MUS} \cup \kappa^2_{MUS} \; s.t. \; \kappa^1_{MUS} \cap \kappa^2_{MUS} = \emptyset$; after reformulation, the solver can assign $V(r_i) = 1$ only once due to the cardinality constraint, thereby causing $\forall j \neq i \; : \; V(r_j) = 0$. Without loss of generality, we assume $c_i \in \kappa^1_{MUS}$, resulting in $\forall c_j \in \kappa^2_{MUS} \; : \; V(c_j) = 0$. This means the assumptions causing κ^2_{MUS} are effectively unchanged, which eventually results in another core being found, containing these assumptions.

As an example, consider a minimisation problem, with associated objective function:

$$S = \{C_s, C_h\} = \{\{(\neg a), (\neg b), (\neg c)\}, \{(a \vee b \vee c)\}\}$$

$$obj = \sum_{c_i \in C_s} 1 * V(\neg c_i) = V(a) + V(b) + V(c)$$

The initial assumptions are $\{\neg a, \neg b, \neg c\}$. Due to the hard clause, a core containing all three assumptions is found: $\kappa = \{(\neg a), (\neg b), (\neg c)\}$. The problem is modified by adding CNF encodings of $\{(a' \leftrightarrow (\neg a \vee r_a)), (b' \leftrightarrow (\neg b \vee r_b)), (c' \leftrightarrow (\neg c \vee r_c)), (r_a + r_b + r_c \leq 1)\}$ to C_h, and replacing the set of soft clauses by $C'_s = \{(a'), (b'), (c')\}$. After this, the solver is run again with new assumptions $\{(a'), (b'), (c')\}$, which allow three solutions to be found: $X_1 = \{(a), (r_a)\}; X_2 = \{(b), (r_b)\}; X_3 = \{(c), (r_c)\}$. These are all optimal, with $obj = 1$.

Note that using a larger bound for the cardinality constraint can cause non-optimal solutions to be found before optimal ones, as is obvious when modifying the example: if $r_a + r_b + r_c \leq 2$ is used instead, several non-optimal solutions are found alongside the optimal ones, such as $X'_1 = \{(a), (b), (r_a), (r_b)\}$.

As previously mentioned, MaxSAT instances may have weights $w_i \in W$ associated with the soft clauses. These weights may be very diverse, meaning that the elements of core κ may have different weights associated with them as well. Let us define the set of weights associated with elements from the core, $W^\kappa = \{w_i \in W | c_i \in \kappa\}$, for convenience. In cases where W^κ contains several unique values, the smallest weight $w_{min} = \min_{w' \in W^\kappa} w'$ is selected, after which all weights $w_i \in W^\kappa$ are lowered by this value [1]. As a result, soft clauses c_i with associated weight $w_i = w_{min}$ have no weight left, while those with higher weights still have a residual, non-zero weight. The relaxation and substitution of soft clauses is mostly done as in unweighted MaxSAT, but the soft clauses with residual weight are kept in their original form as well. Consider the weighted version of our earlier example:

$$S = \{W, C_s, C_h\} = \{\{3,4,5\}, \{(\neg a), (\neg b), (\neg c)\}, \{(a \vee b \vee c)\}\}$$

$$obj = 3V(a) + 4V(b) + 5V(c)$$

After core $\kappa = \{(\neg a), (\neg b), (\neg c)\}$ is found, the assumptions $(\neg b), (\neg c)$ remain; the soft clauses and associated weights are now $C_s = \{(a'), (b'), (c'), (\neg b), (\neg c)\}, W = \{3,3,3,1,2\}$. Note that keeping the assumptions with residual weight active prevents the solver from finding suboptimal solutions $X_2 = \{(b), (r_b)\}; X_3 = \{(c), (r_c)\}$; these solutions have $obj = 4$ and $obj = 5$ respectively. The optimal solution is $X_1 = \{(a), (r_a)\}$ with $obj = 3$, which is not prevented under the current assumptions.

Conceptually, core-guided search quite intuitively reaches the optimal solution to a MaxSAT problem instance. It assumes an ideal situation and adjusts these assumptions once proven wrong, by a core. These adjustments are as small as possible, while still being effective. The first solution found is clearly optimal, as all assignments with lower cost have been proven unsatisfiable by said cores.

Several core-guided algorithms have been developed, including OLL [30] and PM1 [17]. Their general workings are largely similar, though their details are often unique. To provide a more concrete example of such an algorithm, the pseudocode for the unweighted

RC2 algorithm - one of the variations of the OLL algorithm - is denoted in algorithm 1. This prevalent core-guided algorithm is described in [24]. It works as follows:

1. Initialize the cost to 0 and the set of cardinality constraints to the empty set.
2. Check satisfiability of $C_h \cup C_s$, resulting in either solution V or core κ.
   - If V was found, report optimality with the current cost and said assignment. The infeasibility of solutions with lower costs has been proven by producing corresponding cores, meaning that the current solution has the lowest cost among all solutions.
   - If an empty core $\kappa = \emptyset$ was found, C_h contains a MUS. Report unsatisfiability.
   - If a non-empty core κ was found, continue to the next step.
3. Increase the cost by 1; κ is proof that the current lower bound in infeasible.
4. Initialize a set of new relaxation variables as the empty set; this will be used to create the new cardinality constraint.
5. Consider all elements of κ one by one.
   - If this is a cardinality constraint, add (a variable corresponding to) its negation to the set of new relaxation variables. After this, increase the bound of the original cardinality constraint by 1.
   - Otherwise, add a (new) relaxation variable to this clause. Convert the clause into a hard clause, and add the relaxation variable to the set of new relaxation variables.
6. Create a cardinality constraint which enforces that only a single element of the set of new relaxation variables can be set to 1. Add this new constraint to the soft clauses and to the set of cardinality constraints. The new constraint consists of relaxation variables, which directly correspond to the relaxation of a soft clause, and negated cardinality constraints, which are true if and only if the previous bounds are violated - corresponding to a single additional relaxation for that cardinality constraint. Both options correspond to the increase of 1 to the cost.
7. Repeat from step 2.

2.3 Constraint Programming

A Constraint Satisfaction Problem, or CSP, consists of a set of variables $\mathcal{X}$ with corresponding domains $\mathcal{D}$, and a set of constraints $\mathcal{C}$ [16]. In this thesis, we assume all domains $\mathcal{D}(x_i)\ s.t.\ x_i \in \mathcal{X}$ to be continuous integer ranges, i.e. $\mathcal{D}(x_i) = [l_i, u_i] \subset \mathbb{Z}$ where $l_i \leq n \leq u_i \leftrightarrow n \in \mathcal{D}(x_i)\ s.t.\ n \in \mathbb{Z}$. Note that this means that $|\mathcal{D}(x_i)| = u_i - l_i + 1$. The constraints $C_i \in \mathcal{C}$ define relations over (subsets of) $\mathcal{X}$. This relation may be either satisfied

Algorithm 1 The RC2 Algorithm, adapted from [24]

```
Require: A MaxSAT Formula $S = \{C_h, C_s\}$
 1: $cards \leftarrow \emptyset$                                   {Set of cardinality constraints}
 2: $obj \leftarrow 0$
 3: while True do
 4:   $(isSat, \kappa, V) \leftarrow Solve(C_h, C_s)$        {Retrieve core $\kappa \subseteq C_s$ or assignment $V$}
 5:   if isSat then
 6:     return $(obj, V)$                          {Return first satisfying assignment}
 7:   else if $\kappa = \emptyset$ then
 8:     return $(\infty, null)$                                {Report unsatisfiability}
 9:   else
10:     $obj \leftarrow obj + 1$              {Conflicting soft clauses were found; resolve this}
11:     $newRelax \leftarrow \emptyset$
12:     for $c_i \in \kappa$ do
13:       if $c_i \in cards$ then
14:         $newRelax \leftarrow newRelax \cup \{\neg c_i\}$
15:         $C_s \leftarrow C_s \setminus \{c_i\}$                              {Remove old clause}
16:         $cards \leftarrow cards \setminus \{c_i\}$
17:         $rhs(c_i) \leftarrow rhs(c_i) + 1$                        {Increase right hand side}
18:         $C_s \leftarrow C_s \cup \{c_i\}$                             {Insert adapted clause}
19:         $cards \leftarrow cards \cup \{c_i\}$
20:       else
21:         $C_s \leftarrow C_s \setminus \{c_i\}$
22:         $c_i \leftarrow c_i \cup \{r_i\}$                   {Add (unique) relaxation literal}
23:         $C_h \leftarrow C_h \cup c_i$                           {Make relaxed clause hard}
24:         $newRelax \leftarrow newRelax \cup \{r_i\}$
25:       end if
26:       $nc \leftarrow encode\left(\sum_{r_i \in newRelax} r_i \leq 1\right)$      {New cardinality constraint}
27:       $C_s \leftarrow C_s \cup \{nc\}$
28:       $cards \leftarrow cards \cup \{nc\}$
29:     end for
30:   end if
31: end while
```

or violated, depending on the values taken on by the variables $x_i \in \mathcal{X}$. Such constraints C_i can take on many forms, most commonly linear inequalities and global constraints [33]. Global constraints are one of the distinguishing features of CSPs and are used to model and propagate complex relations efficiently.

The goal of a CSP is to find a mapping $V : \mathcal{X} \to \mathbb{Z}$, which satisfies all constraints and domain memberships. Formally, V is subject to the following conditions: [16]

$$\forall x_i \in \mathcal{X} \, : \, V(x_i) \in \mathcal{D}(x_i)$$

$$\forall C_i \in \mathcal{C} \, : \, C_i(\mathcal{X}|\mathcal{D}, V) = 1$$

For convenience, we define $\mathbb{V}$ as the set of all assignments satisfying the first condition.

A CSP might be extended into a Constraint Optimisation Problem (COP) by supplying a function $f \, : \, \mathbb{V} \to \mathbb{Z}$, which defines the cost of assignment. The goal of a COP is to find the optimal assignment $V_{opt} = \underset{V \in \mathbb{V}}{\operatorname{argmin}} \Big\{ f(V) \; s.t. \, \forall C_i \in \mathcal{C} \, : \, C_i(\mathcal{X}|\mathcal{D}, V) = 1 \Big\}$. In this thesis, we assume f to be a weighted sum of (a subset of) the variables - this is the case for most COPs in practice[3]. Without loss of generality, we additionally assume the COP to be a minimisation problem, as has implicitly been done in the definition above.

Constraint Programming (CP) is a paradigm used to solve CSPs and COPs. It models constraints $C_i \in \mathcal{C}$ through corresponding propagators p_i. Propagators remove values which, given the current domains $\mathcal{D}$ and a partial assignment $V_{partial}$, are known to violate C_i:

$$\mathcal{D}' = p_i(\mathcal{D}|V_{partial})$$

$$\forall x_j \in \mathcal{X} \, : \, \mathcal{D}'(x_j) \subseteq \mathcal{D}(x_j)$$

$$\forall v_j \in \big(\mathcal{D}(x_j) \setminus \mathcal{D}'(x_j)\big) \, : \, C_i(\mathcal{X}|\mathcal{D}, V_{partial} \cup \{v_j\}) = 0$$

In other words, given $\mathcal{D}$ and $V_{partial}$, the values removed by a propagator cannot be part of a solution to the CSP or COP.

As an example, consider the following inequality, partial assignment and domains:

$$C_1 = (x_1 + 2x_2 + 3x_3 \leq 10)$$

$$V_{partial} = \{(x_2 = 2)\}$$

$$\mathcal{D}(x_1) = [0, 10], \; \mathcal{D}(x_3) = [1, 4]$$

The propagator corresponding to C_1 enforces $\mathcal{D}'(x_3) = [1, 2]$ and $\mathcal{D}'(x_1) = [0, 3]$, since assigning $x_3 \geq 3$ violates C_1 for any valid value of x_1, and $x \geq 4$ does so for both remaining values of x_3. These changed domains may trigger new propagators, associated with other constraints, to update $\mathcal{D}$ as well. A solver keeps track of which propagators have been triggered using a queue, adding newly triggered propagators to the back. This allows the solver to process the propagators in the order they are triggered.

[3]When this is not the case, we can define $f(V) = 1 * V(obj)$, where obj is a variable that corresponds to the original non-linear cost.

During the solving process, a situation may occur in which no propagator can change the domains based on the current information. This phenomenon is called a fixpoint. When this occurs, the solver selects a variable and divides its domain into disjoint parts, creating separate branches. In doing so, the solver also divides the search space, considering its disjoint parts in the individual branches. By selecting such a part, the solver enforces the constraints defining it, which possibly results in several new propagations. We refer to this procedure of division and selection as a decision.

By repeating the process of propagations and decisions, the solver will eventually reach either a solution - where all constraints are satisfied and all variables are assigned - or a conflict - where a propagations result in a variable having an empty domain, or (equivalently) in a constraint being violated. A solution may either be returned to the end user, or used to guide the remainder of the search process. A conflict on the other hand, proves that the current branch contains no solutions, in response to which the solver moves on to the next branch. Such a conflict may also be used for learning, which prevents the solver from making similar mistakes later in the search process.

As an example, consider a problem with $C_1 = (x_1 + x_2 \leq 10)$ and $C_2 = (x_1 - x_3 \geq 5)$, alongside several other constraints. For the sake of the example, we assume propagations based on x_1 to be non-trivial. At the first fixpoint, the solver limits $x_2 \geq 3$, resulting in $x_1 \leq 7$. At the next fixpoint, the solver limits $x_3 \geq 3$, resulting in $x_1 \geq 8$. These two conditions on the domain of x_1 together allow no valid assignments. The explanation of this conflict can be expressed in the following ways:

$$(x_2 \geq 3) \wedge (x_3 \geq 3) \rightarrow false$$

$$(x_2 \geq 3) \rightarrow (x_3 \leq 2)$$

This conflict proves the current branch has no solutions, and as such the solver continues to the next branch, which sets $x_3 \leq 2$ due to the conflict[4].

Even though global constraints can be decomposed into simpler constraints, their propagators are often more powerful than those of their decompositions combined. Take for example the constraint `alldifferent`$(x_1, ..., x_n)$; this constraint can naively be decomposed into $\frac{n}{2}(n-1)$ constraints of the form $x_i \neq x_j \ s.t. \ 1 \leq i < j \leq n$. With $n = 3$ and domains $x_1 \in \{1,2\}$; $x_2 \in \{1,2\}$; $x_3 \in \{1,2,3\}$, the naive decomposition cannot infer additional information; however, the propagator of the *alldifferent* constraint can infer $x_3 = 3$, using the techniques described in [35]. The ability to create such additional inferences is the reason global constraints are a powerful and versatile tool in CP.

Note that a SAT instance can be mapped to a CSP. Given a SAT formula S, we substitute all $\neg x_k$ by $1 - x_k$, and use the following definitions:

$$\mathcal{X} = Vars(S)$$

$$\forall x_i \in \mathcal{X} \ : \ \mathcal{D}(x_i) = [0,1]$$

[4]Note that this is no longer a decision; the conflict has taught the solver to infer it from the previous decision. As such, if a conflict is found in this branch, the previous decision $(x_2 \geq 3)$ is reverted.

$$C = \left\{ \sum_{l^i_j \in c_i} l^i_j \geq 1 \;\middle|\; c_i \in S \right\}$$

The result is a valid CSP. A MaxSAT instance can similarly be mapped to a COP.

2.4 Lazy Clause Generation [15]

Lazy clause generation, often abbreviated LCG, is an approach that combines the reasoning possible in CP with the learning capabilities of SAT. In order to properly describe its functions, it is first vital to understand how CP problems can be expressed in SAT. An integer variable x can be expressed using a set of boolean variables $(x \leq i)$ for integers i in its domain. This approach also requires a set of consistency constraints $(x \leq i-1) \rightarrow (x \leq i)$. In some formulations, variables $(x = i)$ and constraints $(x = i) \leftrightarrow (x \leq i) \wedge \neg(x \leq i-1)$ are also included. These representations are essential when learning new clauses.

During the search process, the CP solver updates domains based on inference. These updates are additionally encoded into clauses using the encoding above, and the resulting clauses are provided to a module which performs SAT reasoning - and, as a result, learning. This learning function may be able to perform additional unit propagations, which can update the domains in ways the CP solver may not have been able to discover.

Additionally, when a conflict is encountered, the SAT solver can use the provided and newly learned clauses to explain this conflict. Such an explanation is vital in guiding the search process. Such explanations can also serve different purposes: the cores used in core-guided search are explanations consisting only of assumptions.

2.4.1 Lifting explanations

An extracted explanation in CP can sometimes be tightened, or *lifted*, as it is called in [18]. This concept was introduced in [32] and is based on the idea that the explanations use the current domain, while the propagator can reason on larger domains. If the conflict occurs after one or several decisions, as have been described before, the domains of variables may have been artificially limited. The true reason for the conflict may result in a stricter conflict than the one encountered, while this stricter explanation may be more useful later in the search process. An example of this can be found in [18]:

$$2x + 3y + 4z \leq 27$$

$$(x \geq 5) \wedge (y \geq 4) \wedge (z \geq 5) \rightarrow false$$

Based on the constraint, the following, stricter explanation can be derived:

$$(x \geq -2) \wedge (y \geq 4) \wedge (z \geq 5) \rightarrow false$$

If $x \geq 0$, the new clause is shorter, since $(x \geq -2)$ is always true and can be left out. Otherwise, it may cause additional inferences in cases where the old clause could not. This stricter explanation is referred to as a *lifted* explanation. To avoid confusion, the original explanation may be referred to as an *unlifted* explanation.

2.5 Core-Guided Search in Constraint Programming

Based on the competitive results achieved by core-guided search in MaxSAT, Gange et al [18] investigated the efficacy of this approach on CP problems. A total of four different schemes were devised to process the extracted cores; the unique combinations of two proposed reformulation approaches and two proposed weight handling approaches. These approaches are discussed in the following paragraphs, and pseudocode for all four combinations is provided in algorithm 2. This pseudocode is explained at the end of this section.

Reformulation approaches

Slice-based reformulation The slice-based reformulation approach makes use of (binary) *slices* of integer variables, denoted as follows:

$$\lceil\lfloor x \rfloor\rceil_j^i = \max(0, \min(x,i) - j) = \begin{cases} 0 & x \leq j \\ x - j & j < x < i \\ i - j & i \leq x \end{cases}$$

$$\lceil\lfloor x \rfloor\rceil_j = \lceil\lfloor x \rfloor\rceil_j^\infty$$

$$\lceil\lfloor x \rfloor\rceil_{n-1}^n \equiv (x \geq n)$$

Slice-based reformulation works by slicing off the lowest values of variables present in a core; this is done by iteratively applying the following identity to those variables:

$$\lceil\lfloor x \rfloor\rceil_{j-1} = \lceil\lfloor x \rfloor\rceil_{j-1}^j + \lceil\lfloor x \rfloor\rceil_j \equiv (x \geq j) + \lceil\lfloor x \rfloor\rceil_j$$

Slices corresponding to a single value ($\lceil\lfloor x \rfloor\rceil_{j-1}^j$, for some j) are used, as the assumptions fix the variable to this single value. The second part of the identity is referred to as the *remainder*: $\lceil\lfloor x \rfloor\rceil_j$; these are not affected by reformulations[5]. Note that any variable can be interpreted as a remainder, as the following equivalence holds for $x \in [l_x, u_x]$:

$$x \equiv \lceil\lfloor x \rfloor\rceil_{l_x} + l_x$$

At every iteration, assumptions fix the variables in the objective function to their lower bound - which can be mapped to 0 using the identity above. As a result, any core κ contains assumptions of the form ($\lceil\lfloor x \rfloor\rceil_n \leq 0$). The corresponding slices $\lceil\lfloor x \rfloor\rceil_n^{n+1}$ are combined into a reformulation term o_κ, which has a lower bound corresponding to a single relaxation. This bound can again be trivially converted into a bias on the objective value.

As an example, consider the following (unweighted) minimisation problem:

$$obj = a + b + c + d + e \ s.t.$$

$$a, b, c, d, e \in \mathbb{N}$$

[5]Except for the fact that they become progressively smaller.

$$a+b+c \geq 1$$

This results in the following core and resulting reformulation:

$$\kappa = \{(a \leq 0), (b \leq 0), (c \leq 0)\}$$

$$o_\kappa = \lceil \lfloor a \rfloor \rceil_0^1 + \lceil \lfloor b \rfloor \rceil_0^1 + \lceil \lfloor c \rfloor \rceil_0^1 ;\ o_\kappa \geq 1$$

$$obj = \lceil \lfloor a \rfloor \rceil_1 + \lceil \lfloor b \rfloor \rceil_1 + \lceil \lfloor c \rfloor \rceil_1 + d + e + \lceil \lfloor o_\kappa \rfloor \rceil_1 + 1$$

Rerunning the CP solver with this reformulated objective results in an optimal solution.

Note that this approach most closely resembles core-guided search in MaxSAT: the integer variables in the objective function are decomposed into a list of boolean variables (variables $(x \geq n) \equiv \lceil \lfloor x \rfloor \rceil_{n-1}^n$), which are handled in the same way as boolean variables in a MaxSAT problem.

Variable-based reformulation As mentioned earlier in this section, propagators can lift explanations - such as cores - to find tighter bounds. Since this property cannot be exploited by the slice-based approach, [18] proposes an alternative reformulation approach, based on variables. The reformulation terms are a combination of several variables, with its lower bound being a single relaxation higher than the sum of lower bounds of its constituents. Lifted cores can at times modify the lower bound more strongly, decreasing the search space further.

Considering the same example as before:

$$obj = a+b+c+d+e \ s.t.$$

$$a, b, c, d, e \in \mathbb{N}$$

$$a+b+c \geq 1$$

This results in the following:

$$\kappa = \{(a \leq 0), (b \leq 0), (c \leq 0)\}$$

$$o_\kappa = a+b+c \ ;\ o_\kappa \geq 1$$

$$obj = o_\kappa + d + e$$

Rerunning the CP solver with this reformulated objective results in an optimal solution.

To show an example of core lifting, we modify the linear constraint:

$$a+b+c \geq 3$$

The core $\kappa = \{(a \leq 0), (b \leq 0), (c \leq 0)\}$ can now be lifted using this constraint, resulting in an increased lower bound on the new variable:

$$o_\kappa = a+b+c \ ;\ o_\kappa \geq 3$$

$$obj = o_\kappa + d + e$$

This results in an optimal solution after only a single core is extracted, while slice-based reformulation would require additional cores.

Weight handling approaches

Weight Splitting Weight Splitting in CP is a direct translation of the MaxSAT approach to handling weights. It finds the minimum weight w_{min} associated with an element in the core, and decreases the weights associated with all elements by this value: $\forall c_i \in \kappa \,:\, w_i' = w_i - w_{min}$. After reformulation, the weight w_{min} is associated with reformulation term o_κ, which has a lower bound of $l_{o_\kappa} = 1 + \sum\limits_{x \in \kappa} l_x$. This 1 corresponds to the previously mentioned single relaxation.

As an example, consider a minimisation problem with $obj = 2x + 3y + 5z$, for which core $\kappa = \{(x \leq 0), (y \leq 0), (z \leq 0)\}$ is found. This results in the following:

$$w_{min} = \min(w_z, w_y, w_z) = 2$$

$$o_\kappa = x + y + z \,;\, o_\kappa \geq 1$$

$$obj = y + 3z + 2o_\kappa$$

Notice that, if remainders are generated due to a slice-based approach being used, these remainders retain the original weight:

$$o_\kappa = \lceil \lfloor x \rfloor \rceil_0^1 + \lceil \lfloor y \rfloor \rceil_0^1 + \lceil \lfloor z \rfloor \rceil_0^1 \,;\, o_\kappa \geq 1$$

$$obj = \lceil \lfloor y \rfloor \rceil_0^1 + 3 \lceil \lfloor z \rfloor \rceil_0^1 + 2o_\kappa + 2 \lceil \lfloor x \rfloor \rceil_1 + 3 \lceil \lfloor y \rfloor \rceil_1 + 5 \lceil \lfloor z \rfloor \rceil_1$$

Which, after converting non-zero lower bounds to bias terms, results in:

$$obj = \lceil \lfloor y \rfloor \rceil_0^1 + 3 \lceil \lfloor z \rfloor \rceil_0^1 + 2 \lceil \lfloor o_\kappa \rfloor \rceil_1 + 2 \lceil \lfloor x \rfloor \rceil_1 + 3 \lceil \lfloor y \rfloor \rceil_1 + 5 \lceil \lfloor z \rfloor \rceil_1 + 2$$

A drawback of this approach occurs in problems with many diverse weights, where many terms with small residual weights remain in the objective function. These residual weights may also appear in cores, causing the new reformulation variables to have low weights as well. These low weights cause the lower bound increase from each core to be minor, thereby requiring many cores to be extracted before a given lower bound - such as the optimal value - is reached.

Coefficient Elimination Coefficient Elimination aims to reduce the number of variables in the objective function more effectively, by incorporating the weights into the reformulation term o_κ. This has several implications. First of all, the domain of o_κ may be non-continuous. Additionally, the lower bound increase of o_κ is no longer exactly 1, but is instead w_{min}. Furthermore, o_κ has weight $w_{o_\kappa} = 1$.

As an example, consider again the problem from before, with $obj = 2x + 3y + 5z$ and core $\kappa = \{(x \leq 0), (y \leq 0), (z \leq 0)\}$. For a variable-based reformulation approach, this results in the following:

$$o_\kappa = 2x + 3y + 5z \,;\, o_\kappa \geq 2$$

$$obj = o_\kappa$$

In the slice-based paradigm, the following reformulation is done instead:

$$o_\kappa = 2\lceil\lfloor x\rfloor\rceil_0^1 + 3\lceil\lfloor y\rfloor\rceil_0^1 + 5\lceil\lfloor z\rfloor\rceil_0^1 \ ; \ o_\kappa \geq 2$$

$$obj = o_\kappa + 2\lceil\lfloor x\rfloor\rceil_1 + 3\lceil\lfloor y\rfloor\rceil_1 + 5\lceil\lfloor z\rfloor\rceil_1$$

Or, equivalently, after converting lower bounds to bias:

$$obj = \lceil\lfloor o_\kappa\rfloor\rceil_2 + 2\lceil\lfloor x\rfloor\rceil_1 + 3\lceil\lfloor y\rfloor\rceil_1 + 5\lceil\lfloor z\rfloor\rceil_1 + 2$$

In the slice-based case, we indeed see a non-continuous domain: $o_\kappa \in \{2,3,5,7,8,10\}$, i.e. $o_\kappa \notin \{4,6,9\}$, despite these values being between the bounds $l_{o_\kappa} = 2, u_{o_\kappa} = 10$.

A drawback of this approach is that reformulation variables o_κ occur in cores more often, since residual terms are no longer present. This means that if a certain high-weighing term forms multiple cores, weight splitting would be able to keep these partially disjoint; in coefficient elimination, these all depend on one another. This results in more complex relations between the assumptions and the original variables, increasing overhead and computational power required. Additionally, the non-continuous domains may cause many unit cores with weight 1 to occur, requiring relatively many iterations to increase the lower bound by a relatively small amount. However, it does not suffer from low residual weights, and may be able to effectively handle weights that are close, but not equal, to one another.

Pseudocode The pseudocode given in algorithm 2 specifically describes the handling of a core κ, which we assume to be provided as input. Additionally, we assume the objective function $f(V) = \sum_{x_i \in X} V(x_i) * w_i$ to be provided, where the weights w_i can be considered (and modified) separately from their variables.

In algorithm 2, we have made use of two auxiliary functions whose inner workings are not important. Firstly, CREATEVARIABLE returns a newly created variable, of which the value is enforced to be equal to that of the argument (through a hard constraint). Secondly, SETLOWERBOUND overwrites the inferred lower bound of the variable supplied as its first argument, enforcing it to be its second argument.

One should note that, in the slice-based approach, all variables can be adapted to have lower bound 0 using the equivalence $x \equiv \lceil\lfloor x\rfloor\rceil_{l_x} + l_x$. This knowledge has not explicitly been used in the pseudocode, nor in its explanation. Additionally, it should be noted that the value of w_{min} is calculated before any branch is entered, and is used in all branches. w_{min} is the minimum weight associated with a variable x_i $s.t.$ $(x_i \leq l_i) \in \kappa$, and is the fraction of the weight incorporated into the reformulation variable.

Slice-based weight splitting

1. Use the slices $\lceil\lfloor x_i\rfloor\rceil_{l_i}^{l_i+1}$, where $(x_i \leq l_i) \in \kappa$, to define a new reformulation variable o_κ; the value of this variable is equal to the sum of the values of these slices. It has a lower bound of 1, corresponding to a single relaxation. Note that a slice is the inverse of an assumptions, and thus o_κ corresponds to the number of violated assumptions in κ: $\lceil\lfloor x_i\rfloor\rceil_{l_i}^{l_i+1} \equiv (x_i \geq l_i + 1) \equiv \neg(x_i \leq l_i)$

2. Lower the weight of all slices used in o_κ by w_{min}. Those with residual weight remain in the objective function. The remainders also remain present, with original weight.

3. Replace the objective function $f(X)$ with its reformulated version; remove the elements represented by o_κ, retain the slices and remainders as described before, and add the reformulation variable. Note that the lower bound of o_κ has been lowered to 0 by introducing a bias.

Slice-based coefficient elimination

1. Use the slices $\lceil \lfloor x_i \rfloor \rceil_{l_i}^{l_i+1}$, where $(x_i \leq l_i) \in \kappa$, to define a new reformulation variable o_κ; the value of this variable is equal to the *weighted* sum of the values of these slices. It has a lower bound of w_{min}; any single relaxation will incur its associated weight, which is at least w_{min}. Note that the remainders are to remain in the objective function.

2. Replace the objective function $f(X)$ with its reformulated version; remove the elements represented by o_κ, retain the remainders as described before, and add the reformulation variable. Note that the lower bound of o_κ has been lowered to 0 by introducing a bias.

Variable-based weight splitting

1. Define a new reformulation variable o_κ, the value of which is equal to the sum of the values of the variables represented in κ. This variable has a lower bound that is 1 above the sum of the lower bounds of its components, corresponding to a single relaxation.

2. Lower the weights of all variables used in κ by w_{min}. Those with residual weight remain in the objective function.

3. Replace the objective function $f(X)$ with its reformulated version; remove the elements represented by o_κ, retain the variables with residual weight as described before, and add the reformulation variable.

Variable-based weight splitting

1. Define a new reformulation variable o_κ, the value of which is equal to the *weighted* sum of the values of the variables represented in κ. This variable has a lower bound that is w_{min} above the sum of the lower bounds of its components; any single relaxed element will incur its associated weight, which is at least w_{min}.

2. Replace the objective function $f(X)$ with its reformulated version; remove the elements represented in the core, and add the reformulation variable.

Algorithm 2 Pseudocode for all 4 combinations of the proposed approaches in this section; adapted from textual descriptions in [18]

```
Require: A (possibly lifted) core $\kappa$ whose elements are of the form $(x_i \leq l_i)$, and a linear objective function $f(X)$, which defines a weight $w_i$ for every objective variable $x_i$
 1: $w_{min} \leftarrow \min_{(x_i \leq l_i) \in \kappa} w_i$
 2: if slice-based weight splitting is used then
 3:     $o_\kappa \leftarrow \textsc{CreateVariable}\left(\sum_{(x_i \leq l_i) \in \kappa} \lceil \lfloor x_i \rfloor \rceil_{l_i}^{l_i+1}\right)$
 4:     $\textsc{SetLowerBound}(o_\kappa, 1)$
 5:     $res \leftarrow \sum_{(x_i \leq l_i) \in \kappa} (w_i - w_{min}) \lceil \lfloor x_i \rfloor \rceil_{l_i}^{l_i+1} + \sum_{(x_i \leq l_i) \in \kappa} w_i * \lceil \lfloor x_i \rfloor \rceil_{l_i+1}$
 6:     $f(X) \leftarrow f(X) - \left(\sum_{(x_i \leq l_i) \in \kappa} w_i * x_i\right) + res + w_{min} * \lceil \lfloor o_\kappa \rfloor \rceil_1 + w_{min}$
 7: else if slice-based coefficient elimination is used then
 8:     $o_\kappa \leftarrow \textsc{CreateVariable}\left(\sum_{(x_i \leq l_i) \in \kappa} w_i * \lceil \lfloor x_i \rfloor \rceil_{l_i}^{l_i+1}\right)$
 9:     $\textsc{SetLowerBound}(o_\kappa, w_{min})$
10:     $res \leftarrow \sum_{(x_i \leq l_i) \in \kappa} w_i * \lceil \lfloor x_i \rfloor \rceil_{l_i+1}$
11:     $f(X) \leftarrow f(X) - \left(\sum_{(x_i \leq l_i) \in \kappa} w_i * x_i\right) + res + \lceil \lfloor o_\kappa \rfloor \rceil_{w_{min}} + w_{min}$
12: else if variable-based weight splitting is used then
13:     $o_\kappa \leftarrow \textsc{CreateVariable}\left(\sum_{(x_i \leq l_i) \in \kappa} x_i\right)$
14:     $\textsc{SetLowerBound}\left(o_\kappa, 1 + \sum_{(x_i \leq l_i) \in \kappa} l_i\right)$
15:     $res \leftarrow \sum_{(x_i \leq l_i) \in \kappa} (w_i - w_{min}) x_i$
16:     $f(X) \leftarrow f(X) - \left(\sum_{(x_i \leq l_i) \in \kappa} w_i * x_i\right) + res + w_{min} * o_\kappa$
17: else if variable-based coefficient elimination is used then
18:     $o_\kappa \leftarrow \textsc{CreateVariable}\left(\sum_{(x_i \leq l_i) \in \kappa} w_i * x_i\right)$
19:     $\textsc{SetLowerBound}\left(o_\kappa, w_{min} + \sum_{(x_i \leq l_i) \in \kappa} w_i * l_i\right)$
20:     $f(X) \leftarrow f(X) - \left(\sum_{(x_i \leq l_i) \in \kappa} w_i * x_i\right) + o_\kappa$
21: end if
```

Chapter 3

Core-Guided techniques in CP

Over the years, various MaxSAT solvers applying core-guided search have been developed and evaluated, often applying additional techniques. This is seen, for example, in the yearly MaxSAT Evaluation [34]. Many of these techniques have significantly improved the performance the solver applying them, raising the question if these same features can also improve the performance of solvers in CP.

This section describes several of such features, which have been selected for implementation in a CP solver, Pumpkin [11]. Note that only a limited number of features was selected, based on expected effectiveness. These techniques are WCE, explained in section 3.1; Stratification, explained in section 3.2; Hardening, explained in section 3.3; and finally Partitioning, explained in section 3.4.

For each feature, the corresponding section first describes how it functions in core-guided search for MaxSAT. This knowledge is then used to explain how it can be applied to CP. After this, the effects of the reformulation and weight handling approaches, as described in section 2.5, are discussed where applicable. Finally, any additional important implementation details are mentioned, if applicable.

The Geas solver [18] The first three features described in this section have been previously implemented in CP, as part of the Geas solver [18]. However, in this paper, the features were not evaluated individually. One of the goals of this thesis is to perform this individual, comparative evaluation. Geas also implemented a technique called Core-Boosting [7]; this technique has been evaluated in [18], and as such has been left out of this thesis.

3.1 Weight-Aware Core Extraction (WCE) [6]

Weight-aware Core Extraction, or WCE, is an approach seen in numerous solvers from the yearly MaxSAT Evaluation [34]. It is referred to as Independent Core Extraction in [18][1]. Since WCE is the more prevalent name in MaxSAT, this thesis refers to the feature as such.

[1]The phrasing in [18] seems to imply that for slice-based variants, WCE is adapted to remove the slices entirely rather than split the weight, in the case of weight splitting. In our implementation, we also apply weight splitting (when specified) in the slice-based case, rather than only in the variable-based case.

In MaxSAT WCE is a technique where multiple cores are extracted before a reformulation is performed. A solver applying WCE postpones the addition of cardinality constraints until several cores have been found, while still performing the associated relaxations. The effect is that the extracted cores are independent, and that a relaxed version of the problem is considered during much of the solving process. Once the considered relaxation becomes satisfiable, no more independent cores can be extracted, and the cardinality constraints are finally added to the problem - which returns to an unrelaxed version this way. Note that this step generates an intermediate solution, which may be returned to the user if desired, or if the time limit is reached.

The postponing of cardinality constraints allows all derived soft clauses ($(a'), (b'), (c')$ in previous examples) to be trivially satisfiable through the relaxation variables; all corresponding relaxation variables can be assigned 1 without issue. This prevents the derived soft clauses from occurring in cores until later in the search process. The eventual effect is that the relations between these derived variables and the original variables are less complex, which in turn reduces overhead.

Note that postponing cardinality constraints is one example of how WCE can be performed. If cardinality constraints are soft, these can be encoded immediately, while only enforcing the bound later. Alternatively, the assumptions which enforce that the derived soft clauses are satisfied may be left out.

Disjoint Core Extraction [12] or DCE is a variation of WCE. It applies a similar approach, but operates on unweighted MaxSAT instances. For the remainder of this thesis, DCE is considered a special case of WCE, and not be mentioned explicitly.

In CP WCE in CP is quite similar to WCE in MaxSAT: once a core is found, the relaxation proceeds as normal, but no assumptions are placed on the reformulation variable - as in MaxSAT, this means no constraints are placed on the values of the associated variables. Once an intermediate solution is found, all reformulations are completed by adding the previously omitted assumptions.

Effect of reformulation approach A slice-based approach is likely to result in the extraction of many cores before allowing a solution to exist, as each core removes relatively little terms from the objective function: the constraint $x + y \geq 5$ alone already causes three disjoint cores before the first reformulation. Note also that many slice-based cores would be heavily intertwined, which is partially prevented by WCE: again looking at constraint $x + y \geq 5$, we see that of the five cores needed to resolve it, four can realistically contain the previous reformulation variable[2]. As such, WCE would cause less complex relations between the assumptions and the original variables. Additionally, the remainders causes the considered relaxations to remain close to the original problem, presumably resulting in good intermediate solutions.

[2]Both cores $\{(x \leq 1), (y \leq 1)\}$ and $\{(x \leq 1), (o_\kappa \leq 1)\}$ *s.t.* $o_\kappa = \lceil \lfloor x \rfloor \rceil_0^1 + \lceil \lfloor y \rfloor \rceil_0^1$ correspond to a MUS; either may be found, and it may even be the case that both appear in a non-minimal core. Similar situations occur for higher values, and codependency of the reformulation variables is, to some extent, guaranteed.

A variable-based approach is likely to extract far fewer cores, which are individually more informative. The total amount of information discovered before reformulation is expected to decrease, as entire variables are removed from consideration when appearing in a core. Consider for example a toy problem with constraints $(a+b \geq 1) \wedge (a+c \geq 8)$; if the core $\{(a=0),(b=0)\}$ is processed first, variable a cannot form cores with c until the reformulation is performed; something that was possible in the slice-based approach. This can result in more derived terms being present in subsequent cores, which in turn causes more complex relations between the input variables and objective function. Additionally, the variable-based approach is expected to provide worse intermediate solutions, since it lacks remainders and thus results in a much more relaxed problem.

The variable-based approach is expected to benefit from applying WCE, though presumably less so than the slice-based approach. In both approaches, WCE is expected to decrease complexity of the relations between original and reformulation variables, causing a significant speedup.

Effect of weight handling approach When using weight splitting, a single variable can appear in several independent cores; after appearing once, it may have residual weight, allowing additional cores containing it to be extracted. Additionally, we expect that the assumptions corresponding to these terms cause the relaxations to more closely resemble the original problem, which may contribute to better intermediate solutions. These factors are expected to improve the solver to a notable extent.

Coefficient elimination allows slightly fewer cores to be extracted, as each variable can only occur in a single core. This may decreases the total amount of information found before a reformulation occurs. Additionally, since the terms are fully removed, the problem is more strongly relaxed after each core, thereby resulting in slightly worse intermediate solutions. However, WCE is still expected find several cores between reformulations, resulting in a moderate speedup.

While both weight splitting and coefficient elimination are expected to benefit from WCE, weight splitting is likely to be much more suitable for this approach.

Implementation All found cores are processed normally. However, when the assumptions are updated, the new assumption $(o_\kappa \leq l_{o_\kappa})$ is held back. Both the variable o_κ and the bound l_{o_κ} are stored until a solution is found. When satisfiability is reported, these unadded assumptions are added to complete the reformulation. Note that l_{o_κ} may have changed in the meantime; this is accounted for by comparing the stored lower bound to the current one, and updating the problem if necessary.

3.2 Stratification [2]

Similar to WCE, this technique is used by numerous solvers in numerous years of the MaxSAT evaluation [34], and has been implemented for CP in the Geas Solver [18].

In MaxSAT Stratification is a technique which aims to extract high-weighing cores first, thereby causing the lower bound on the objective value to increase faster. This is done by splitting the objective variables into strata, based on the absolute value of their weight. Such a stratum may contain any number of variables; if many variables have the same weight w_i, the associated stratum will be large, but if a variable has unique weight w_j, it will be the sole representative of its stratum. The strata are added from largest to smallest associated weight, advancing when no more cores can be found using the currently added variables. Note that intermediate solutions are found before a new stratum is added, as this is the logical result of checking satisfiability after removing all cores. Like in WCE, this allows the solver to return the best solution so far if a time limit is reached.

Note that stratification enforces a (partial) precedence on the cores; one that allows the lower bound to be more rapidly increased, presumably resulting in the optimal solution being reached faster. As an example, consider the following problem, where we assume that the solver considers constraints in order:

$$obj = 10a + 10b + c$$

$$S = (b \vee c) \wedge (a \vee b)$$

Without stratification, two cores are discovered; $\kappa_1 = \{b, c\}$ of weight 1, and $\kappa_2 = \{a, b\}$ of weight 9. When stratification is active, the first iteration considers only a, b as soft clauses, resulting in the discovery of $\kappa_1' = \{a, b\}$ with weight 10. After this, the optimal solution is discovered - having required fewer cores than the stratified approach. Though more complex, such situations may also occur in many real-life MaxSAT problems.

In CP Stratification in CP functions in nearly exactly the same way as in MaxSAT; the objective function is split up by weight, and the strata resulting from this are added from largest to smallest corresponding weight, proceeding whenever satisfiability is detected. This allows cores with high weight to be found earlier in the process, possibly resulting in fewer cores being needed to solve the problem.

However, because of the increased domain sizes, the implications of stratification are fundamentally different in CP: in MaxSAT, violating a single soft clause with weight 1 has a much lower effect than violating a soft clause with weight 10. In CP however, a variable with weight 1 may still have a very large domain; if it is assigned a large value, its effect may be much larger than that of other variables, such as a variable with weight 10 that is assigned a low value. Strata may thus functionally be less disjoint, making it likely that, in general, CP benefits less strongly from stratification.

Effect of reformulation approach The slice-based approach is expected to benefit from stratification. It effectively increases the weight associated with the early cores, while the slice-based character allows for cases where a low-weighing variable conflicts with a high-weighing remainder - the core may not need to contain reformulation variables in such cases.

Stratification is likely to improve the variable-based approach as well, though slightly less than the slice-based variant. This is mainly because of the latter case; a low-weighing variable may conflict with the reformulation variable, since the remainder is incorporated in it. The former case, however, is still likely to improve performance.

Both reformulation approaches are expected to experience a moderate benefit from stratification, though the slice-based approach presumably benefits slightly more.

Effect of weight handling approach Stratification with weight splitting results in some high-weighing cores being found, but may cause low residual weights to appear earlier in the solving process. These low residual weights could diminish the effectiveness of stratification slightly. However, this effect is expected to be much smaller than in cases where these lower weights are already present from the start; meaning stratification will still have a positive effect.

Whether coefficient elimination benefits from stratification depends on how strongly the values of reformulation variables can be inferred. The residual weights discussed before do not occur; instead, all reformulation variables have weight 1, which would intuitively decrease the effect of stratification. However, the domain of said variable may often be non-continuous, especially in earlier strata; if the values in said domain can be correctly inferred, the lower bound on the objective function may still rapidly increase. If such inferences cannot be done effectively, stratification loses most of its effect. Since the implementation will exclusively make use of continuous domain encodings, the effect is expected to be small.

Weight splitting is likely to improve due to stratification, but some problems may experience only little benefit. The benefit coefficient elimination experiences will presumably be low because of the reformulation variables with weight 1.

Implementation Stratification is straightforward to implement. The objective variables are divided into several sets, based on their weights; one set is created for each unique absolute weight, which contains all variables associated with that weight. The assumptions corresponding to the first of these sets are added at the start of the process; the assumptions corresponding to the other sets are iteratively added, in order, whenever the solver detects satisfiability. As with WCE, the stored lower bounds of future strata may change throughout this process; these are also checked, and appropriately handled.

3.3 Hardening [2]

Hardening is the third of the three features used in the Geas Solver [18] as well as in this thesis. As with WCE and stratification, it has often been used in solvers submitted to the MaxSAT Evaluation [34].

In MaxSAT Hardening prunes parts of the search space, based on the upper and lower bounds u_{obj}, l_{obj} on the objective value. The upper bound is generally determined by intermediate solutions found during search; these provide a value much lower than the inferred upper bound, and it is a given that the optimal solution has at most the same objective value.

As such, hardening is generally combined with WCE or stratification, which both produce intermediate solutions. Note that any technique producing intermediate solutions suffices.

Hardening compares the upper bound u_{obj} to the weights $w_i \in W$ of soft clauses $c_i \in C_s$. Once it detects that $u_{obj} - l_{obj} < w_i$, it *hardens* c_i from a soft clause to a hard clause. This is done because violating c_i causes the objective value to be at least $l_{obj} + w_i$; this thus exceeds u_{obj}, meaning that any solution doing so is worse than the current solution, and as such automatically proven not to be optimal.

In CP In CP, hardening can generally not fix variables to a given value, but it can effectively limit the domains of certain variables through inference. Consider for example a problem with $obj = 5a + 2b$ and initial domains $a, b \in [0, 100]$, for which a solution with $f(V) = 30$ has been found. This hardens $(obj \leq 30) \rightarrow (5a + 2b \leq 30)$, limiting the domains to $a \in [0, 6]; b \in [0, 15]$. These bounds on a and b can also be directly enforced.

As previously stated, hardening depends on tight upper bounds, generally provided by intermediate solutions. This is no less true in CP than in MaxSAT.

Effect of reformulation and weight handling approaches For all combinations of reformulation and weight handling approaches, the hardened bounds are applied directly to the original objective variables, as well as to the reformulation variables. The direct application to the objective variables is likely to have an equal effect in all cases. However, the effectiveness of the application to reformulation variables depends on the underlying approaches.

In the slice-based approach, the hardening of reformulation variables is unlikely to have a large impact, due to the limited domains. In the variable-based approach, the domains are larger, making impact more likely. Among these, coefficient elimination is likely to experience a bigger impact than weight splitting, as the domains are often slightly larger - depending on the weights of the variables.

Implementation Whenever an intermediate solution is reported, its corresponding objective value is compared to that of the best solution found so far. If it is better, the upper bound on the objective value is tightened to this new value by adding the corresponding constraint. Additionally, this new upper bound u_{obj} is used to calculate the interval $u_{obj} - l_{obj}$, which is then compared to the sizes of the scaled domains of individual variables. This is done for both original variables and reformulation variables, limiting the domain of variable x if $w_x * (u_x - l_x) \geq u_{obj} - l_{obj}$. For example, if variable x has weight $w_x = 10$, with upper and lower bound $u_{obj} = 50, l_{obj} = 0$, we know that $x \leq u_{obj}/w_x = \frac{50}{10} = 5$, and thus $x \leq l_x + 5$. We know this regardless of the other terms in the objective function, and we know this is true even if the current residual weight of x is lower than w_x. To incorporate this knowledge into the solver, we add a constraint encoding the literal $(x \leq l_x + 5)$, thereby limiting its domain.

3.4 Partitioning

Partitioning [31] is a technique introduced by Open-WBO [27]. Contrary to the other techniques used in this thesis, this technique is used by only a very limited number of solvers - several variants of Open-WBO. Despite this, the solvers applying it have proven effective, making it a suitable candidate to implement in CP.

In MaxSAT Based on the input problem, a graph representation is created. This graph is partitioned by applying the Louvain community-finding algorithm [8]. For each partition $\gamma_k\ s.t.\ \gamma_k \cap C_s \neq \emptyset$, core-guided search is performed. During this search, the solver does not consider soft clauses that are not in the partition; i.e. it considers the relaxed problem $\{C_h, \gamma_k \cap C_s\}$, rather than the full problem $\{C_h, C_s\}$. As such, all γ_k correspond to different relaxations, considering disjoint sets of soft clauses - since each clause can only occur in a single partition. Once all γ_k have been solved to optimality, one or several merges are performed. These merges result in new, larger partitions, which may contain additional cores, and thus need to be optimised further. These solving and merging steps are repeated until a single partition remains, representing the entire problem.

Intuitively, partitioning divides the problem into several relaxed versions, all of which consider a much smaller set of soft clauses. The community-finding algorithm ensures that elements within such a set are relatively closely related, while being less closely related to the elements of other sets. These conditions theoretically cause relatively many cores to be found within partitions, with these cores being more likely to be minimal - the assumptions from other partitions are not present, and thus cannot interfere. Additionally, these cores may be found more efficiently due to the search space being less constrained. The importance of small cores is emphasised in [4]. Merging allows increasingly larger cores to be found, as more assumptions are combined, while ensuring that cores are always found as early as possible. Note that, this way, the largest cores are postponed until relatively late in the search process.

The graph representation used for partitioning may be either a resolution-based graph (RES, as used in practice by the Open-WBO solvers), or a clause-variable incidence graph (CVIG, as is used in our CP implementation). Additionally, the merging may be done in a sequential (not used) or balanced (used both by Open-WBO and in our CP implementation) fashion. Both approaches for both elements are extensively described in [31]; only the CVIG representation and the balanced merging variant are briefly described here. In [31], the various approaches are evaluated, showing that the balanced merging scheme performs better with both graph representation. Furthermore, it is shown that the RES representation is more effective than the CVIG representation.

CVIG representation In a CVIG, both clauses and variables are represented by vertices in the graph. Edges exist between a variable vertex v_i and a clause vertex c_j if and only if $v_i \in c_j$. The weight of such an edge is proportional to $\frac{1}{|c_j|}$[3]. Note that no edges

[3]As can be read in [31], the weight is larger if v_i occurs in one (or several) soft clauses. However, this element is not present in the RES variant, nor is it explained in the text, and as such has been left out of our

exist between two variable vertices, nor between two clause vertices, resulting in a bipartite graph. A clause vertex representing c_j has $|c_j|$ edges, and a variable vertex representing v_i has $|\{c_{i'} \in C_h | v_i \in c_{i'}\}|$ edges.

Balanced merging scheme In a balanced merging scheme, the number of partitions is approximately halved every iteration. At the start of the process, the connection strengths d_{ij} are calculated for each pair of partitions γ_i, γ_j. Given $w(u,v)$, which returns the weight of edge (u,v), or 0 if no such edge exists, we define d_{ij} as follows:

$$d_{ij} = \sum_{u \in \gamma_i} \sum_{v \in \gamma_j} w(u,v)$$

Note that these only need to be calculated from scratch once, as $d_{il} = d_{ij} + d_{ik}$ for $\gamma_l = \gamma_j \cup \gamma_k$ (subject to $\gamma_i, \gamma_j, \gamma_k$ being disjoint partitions). Once all d_{ij} are known, the pair of partitions with the strongest connection is selected, merged, and removed from consideration. These steps are repeated until either one or zero partitions remain; if one remains, it is left unchanged, as if it were merged with an empty partition. The newly merged partitions, along with the possibly remaining unmerged partition, are used in the next iteration. Solving and merging alternate until the final partition is solved to optimality.

In CP In order to apply partitioning to CP, a reasonable weighted graph representation needs to be defined for input problems. According to [31], the RES representation performs better than its alternative; however, this representation is much harder to define for CP problems than for SAT problems. As such, the CVIG representation is used. The weighing scheme for the edges is loosely based on the CVIG weighing scheme from [31].

The community-finding algorithm can be directly reused, as this operates independently of the input problem; it operates directly on the weighted graph. The implementation from [19] has been used to this end. The merging approach is also mostly problem independent, and thus the balanced merging scheme can be directly applied.

Graph representation weighing scheme In CP, several types of constraints exist, which need to be considered separately by the weighing scheme. We consider two distinct types: linear inequalities, which are of the form $c_i = \sum_{v_j \in c_i} r^i_j * x_j \,\square\, n$ for some $\square \in \{\leq, =, \geq\}$ and $n \in \{0,1\}$[4]; and global constraints, which we assume to model arbitrarily complex relationships on unweighted variables. Any constraint that is not a linear inequality is considered a global constraint in this weighing scheme.

In [31], the weight of an edge conceptually corresponds to the contribution of v_i to c_j, and as such a weight proportional to coefficient r^j_i seems appropriate for linear constraints. To ensure all edges have positive weight, the absolute value $|r^j_i|$ is used. Notice that, to accurately represent the influence, these coefficients need to be normalised. Cases where

version of partitioning.

[4]Note that in many cases, the values of r^i_j need to be normalised to achieve this form.

$c_i = \sum_{v_j \in c_i} r^i_j * v_j \ \leq \ n \ s.t. \ n \neq 0$ can be normalised by defining $r^{i\prime}_j = \frac{r^i_j}{n}$, which results in $c'_i = \sum_{v_j \in c_i} r^{i\prime}_j * v_j \ \leq \ 1$ - as mentioned before. In cases where $c_i = \sum_{v_j \in c_i} r^i_j * v_j \ \leq \ 0$, n cannot be used as normalisation factor; as such, $m = \sum_{v_j \in c_i} |r^i_j|$ is used instead, resulting in $r^{i\prime}_j = \frac{r^i_j}{m}$. This definition is based on the intuition that all variables combined exert an influence of 1. Using these definitions, the weight function for linear constraints can be defined as follows:

$$w_{linear}(x_i, c_j) = \left| r^{j\prime}_i \right|$$

No factors comparable to $r^{j\prime}_i$ exist for global constraints. Since the propagators for global constraints are relatively strong, we assume the variables in such a constraint to be quite closely related; although an increase in the degrees of freedom causes this relation to become weaker as more variables are considered. To model this, we use a weight near - but not equal to - 1, which exponentially decreases with the number of variables used in the global constraint. 0.9 has been chosen for this purpose. This results in the following weighing function:

$$w_{global}(x_i, c_j) = 0.9^{|c_j|}$$

A special case to consider is that of reified constraints. The effect of the reification variable does not depend on the weights or number of other variables, and is very large. Reification variables directly correspond to the truth value of the constraint itself, giving it complete control rather than influence. As such, the weight of the edge between these two vertices should be 1. This value exceeds the weights of all global and most linear constraints, thereby strongly increasing the probability of the constraint and variable being part of the same community.

A more suitable weighted graph representation may exist, and a qualitative evaluation of different representations could provide valuable insights. However, such an evaluation is outside the scope of this thesis, and presents a possible topic for future research.

Example To better understand the described partitioning approach, we consider the following problem. This problem is fully solved by a slice-based weight splitting solver applying partitioning, to illustrate both partitioning and its effects on solving.

$$obj = 3a + 4b + c + 12d + e + 7f$$

$$c_1 : 5a + b \geq 10$$

$$c_2 : b + 2e \leq 5$$

$$c_3 : alldifferent(b, c, d)$$

$$c_4 : 3c - 4d \leq 2e$$

$$c_5 : 4e + 7f \geq 7$$

After normalisation of the weights, The problem is as follows:

$$obj = 3a + 4b + c + 12d + e + 7f$$

$$c_1' : \frac{1}{2}a + \frac{1}{10}b \geq 1$$

$$c_2' : \frac{1}{5}b + \frac{2}{5}e \leq 1$$

$$c_3' : alldifferent(b, c, d)$$

$$c_4' : \frac{1}{3}c - \frac{4}{9}d - \frac{2}{9}e \leq 0$$

$$c_5' : \frac{4}{7}e + f \geq 1$$

The graph representation of this problem has been visualised in Figure 3.1.

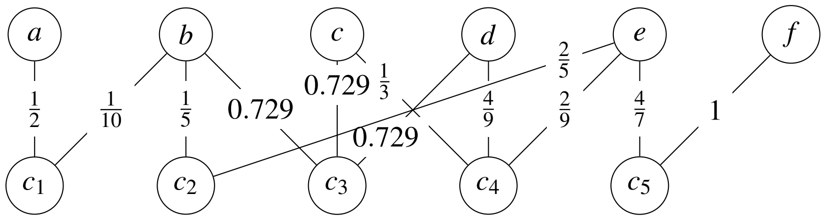


Figure 3.1: A graph representation of an example problem. The edge weights correspond to the normalised weights in linear constraints, or to $0.9^{|c_3|} = 0.9^3 = 0.729$ for the three edges incident to the node representing global constraint c_3.

The partitioning algorithm from [8] will not be iterated in this thesis, but results in the following partitions of soft constraints: $\gamma_1 = \{a\}, \gamma_2 = \{b, c, d\}, \gamma_3 = \{e, f\}$.

When considering partition γ_1, the solver only finds unit core $\kappa_1^1 = \{(a = 0)\}$ before a solution is found. γ_2 produces three cores: $\kappa_1^2 = \{(b = 0), (c = 0)\}$, $\kappa_2^2 = \{(d = 0)\}$ and $\kappa_3^2 = \{(b = 0), (c = 1), (d = 1)\}$, after which it becomes satisfiable. γ_3 becomes satisfiable after two cores: $\kappa_1^3 = \{(e = 0), (f = 0)\}$ and $\kappa_2^3 = \{(e = 1), (f = 0)\}$.

During the merging phase, we greedily select γ_2, γ_3 to be merged into new partition γ_4, since $d_{23} = \frac{1}{5} + \frac{2}{9} = \frac{19}{45} > d_{12} = \frac{1}{10} > d_{13} = 0$. This leaves γ_1 unmerged. After this, partitions γ_3 and γ_4 are again solved to optimality, both producing a solution before encountering any cores. These two partitions are subsequently merged into γ_5, which represents the entire problem. γ_5 produces a single core, $\kappa_1^5 = \{(a = 1), (f = 0)\}$, after which a solution with cost 22 is found. This is the optimal solution.

This example shows that most cores are already found in the relaxed problems represented by the individual partitions, presumably allowing for more efficient search. A second effect, not shown in this example, is that large cores are found later in the search process, since only then all elements are considered at once.

Effect of reformulation approach Although the variable-based approach is expected to experience a performance increase, it may not be the most suitable approach. There may be cases in which a non-partitioned approach finds a single core containing multiple overlapping MUSes, after which unit cores resolve the conflicts efficiently; the partitioned approach may find the individual MUSes in separate partitions, causing a later core to depend on the resulting reformulation variables. This slightly increases overhead.

The slice-based approach suffers less from the aforementioned problem, due to remainders being present. As such, the slice-based approach is expected to benefit from the more efficient search and smaller cores resulting from partitioning, with no noteworthy downsides.

Overall, both reformulation approaches are expected to experience a performance increase, though the slice-based approach is presumably more suitable for partitioning.

Effect of weight handling approach Weight splitting is likely to cause terms with residual weights to remain in the objective functions of partitions. After a merge, these terms can appear in new cores, instead of a reformulation variable. This keeps overhead low and can increase performance.

While coefficient elimination is still likely to experience a speedup, the aforementioned benefit is absent; after a merge, many of the new cores are likely to depend on one or multiple reformulation variables, slightly increasing overhead over time. Note that this overhead is still expected to be lower than in cases where partitioning is not applied.

Both weight handling approaches are expected to benefit from partitioning, though weight splitting appears to be a more suitable paradigm to apply partitioning to.

Chapter 4

Experimental setup

In order to determine the effectiveness of the features implemented in the solver, an extensive evaluation is required, in which the different versions of the solver are compared to one another on a set of benchmark problems. This evaluation shows which features have an effect on the efficiency of the solver, and whether that effect is positive or negative. In order to draw more general conclusions, several different metrics are used in the evaluation; this allows us to draw conclusions even in cases where the overall effectiveness of the solver is unchanged, as well as explain the performance changes. This section describes the workflow and resources used to perform said evaluation.

The implemented features allow for different possible configurations of the solver. Not all of these combinations are evaluated, as this is an intractable amount; the ones that are evaluated are described in section 4.1. These variants are all run on the same set of benchmarks, detailed in section 4.2. The Gourd-Test framework [9] is used to set up and execute the runs necessary to perform the evaluation. In section 4.3, relevant details of its setup are provided. Several of the metrics needed to perform the evaluation are not provided by Gourd-Test, and are implemented to be monitored by Pumpkin itself. These metrics are described extensively in section 4.4. The raw data points are processed into a more manageable output by a script, as explained in section 4.5.

The experiments are run on the DelftBlue supercomputer [13], with 16 gigabytes of memory for each run, and a timeout of 10 minutes enforced by the scheduler. DelftBlue makes use of Intel Xeon E5-6448Y 32C 2.1GHz processors. Pumpkin is single-threaded.

Unfortunately, results from DelftBlue are not perfectly consistent; repeated experiments have shows that the exact same setup can lead to different amounts of time being taken. Note that this only affects the time, not the other metrics; the code execution itself is unaffected. For the sake of this thesis, we assume this effect to be minor.

4.1 Solver variants

A total of four different features have been implemented in Pumpkin over the course of this thesis; as mentioned in chapter 3, these are WCE (section 3.1), Stratification (section 3.2), Hardening (section 3.3) and Partitioning (section 3.4). The different reformulation and weight handling approaches seen in section 2.5 provide four base variants, to which these features can be added. The features are mostly considered individually, with a single exception: since hardening needs intermediate solutions to take effect, this feature must be combined with another which generates these. To this end, hardening is combined with both WCE and stratification. Note that the effect of hardening largely depends on the quality of the intermediate solutions, and as such the differences between these two variants may be significant.

In total, twenty-four variants of the solver are considered; six for each combination of a reformulation approach and a weight handling approach. The six variants for a given base solver are:

- No additional features
- WCE
- Stratification
- Hardening using WCE
- Hardening using stratification
- Partitioning

The code is available at [10]. The release *Core-guided pumpkin: v1.1.0* contains the version of the code used during the evaluation of twenty of these twenty-four variants, but lacks partitioning. The finalisation of partitioning resulted in release *Core-guided pumpkin: v1.1.1*, which contains the version of the code used during the evaluation of the final four variants, those applying partitioning.

4.2 Benchmark files

In order to evaluate the solver, a set of benchmark problems is required. The MiniZinc Challenge Benchmarks repository [28] is used as a source for such problems. Benchmarks from this challenge provide a large range of suitable problems, and a (limited) selection has previously been used in [18]. In this thesis, a larger set of problems is used, including more recent problems.

Not all benchmark problems found on [28] are suitable; some problem families are CSPs, while core-guided search specifically performs optimisation - thus requiring COPs. For each of the COP families, a total of ten problem instances are randomly selected by the bash `shuf` command. This process has resulted in a total of 782 instances. These instances are flattened and given as input to the different variants of the solver. The flattened files files have been included in [10].

Note that the flattener normalises all files to have a single objective variable. In most cases, the constraint defining this variable can be easily extracted. If no such constraint exists, or the constraint is not a linear weighted sum, core-guided search is performed on this single variable - thereby effectively performing lower-bounding linear search. These cases can still be valuable, as they help ascertain whether core-guided search is suitable for this type of problems - making it a good general purpose solver. These problem families are explicitly considered in section 5.11.

In addition to specific consideration in the case of a single objective variable, several quantities are extracted from the input files, to see how these influence the performance of each solver. The following features are extracted to this end:

- Number of objective variables
- Number of unique weights associated with objective variables
- Number of variables
- Number of constraints

These metrics roughly indicate the size of the problem, and the size of the objective function. They help determine which solver variants are most effective on very large instances, which suffer from a large number of unique weights, and which experience a performance decrease when dealing with a larger objective function.

4.3 Gourd-Test framework

Gourd-Test [9] is an evaluation suite for CP solvers, designed with Pumpkin [11] in mind. This framework is thus suitable to evaluate the core-guided version of Pumpkin as well.

Gourd-Test receives a list of solver executables and their respective arguments, alongside a list of input files. It schedules a run for every combination of a solver and an input file - either to be run as soon as the resources are available, or through a scheduler on a supercomputer. All associated outputs are stored in separate files to allow later in-depth analysis of each run.

The Gourd-Test framework is able to provide the following data points:

- **Exit status** - There are several possible ways in which a run can end. It may either reach optimality or prove unsatisfiability, if the run is successful; it may also exceed its time or memory limits, and be unable to solve the problem appropriately. Gourd reports whether a run was successful, and what the reason for failure was, if any.

  We expect solver variants with additional features to result in successful runs more often than the base variants. This because we expect them to be more efficient, mainly in terms of time taken, thereby presumably reaching optimality in cases where base variants experience a timeout.

- **Time taken** - Most problems cannot be solved instantly, and thus take some time before the correct result can be returned. The time taken before proving optimality or unsatisfiability is a useful metric, defining the efficiency of a solver on the most basic level. The faster a solver is, the better it must be at finding cores, and subsequently solutions.

Solvers with additional functionalities are expected to take less time than solvers without additional functionalities. However, the amount of time saved depends heavily on several factors; some features may perform better, or worse, depending on the reformulation and weight handling approaches, and depending on the underlying problem instance. Different solvers are expected to experience different amounts of speedup, though it is hard to predict a priori which ones benefit to which extent.

Based on these data points, the solvers can be roughly compared to one another, and early conclusions about the implemented features can be drawn. For example, a significant speedup, or lower number of timeouts, would already allow us to draw conclusions on which features are effective, and which are not.

4.4 Collection of additional data

While the data points described above allow for a simple evaluation, a more detailed one is desirable. As such, several additional data points are collected for every solver-input combination that ran successfully. These additional measurements allow stronger conclusions to be drawn on the various techniques, the reformulation and weight handling approaches, and on core-guided search for CP in general.

The following general data points are extracted:

- **Objective value** - In order to verify correctness of the solver variants, the optimal objective values found are collected and compared. These are not used further in the evaluation, but only serve as an additional verification.

- **Number of lower bound steps** - During the search process, the lower bound on the problem is constantly updated. Although the lower bound eventually reaches the optimal objective value, information can be retrieved from the number of steps; this corresponds roughly to the number of cores extracted. If fewer steps are taken to reach the optimal solution, this means fewer cores were processed to solve the problem, indicating that the cores were of higher quality. Additionally, this smaller number of cores may indicate less search, and thus explain improvements on the time taken.

  The number of steps is expected to decrease mainly for stratified approaches, which specifically focus on extracting high-weighing cores. Stratification also causes relaxations to be considered most of the time, which causes fewer assumptions to be present and thus increases the chances of cores being minimal - preventing non-minimal cores with unnecessary low-weighing element. WCE and partitioning are also likely to slightly decrease the number of steps due to the second factor, but to a lesser extent than stratification. For consistency across variants, we only count the lower bound step if it is induced by a core; if this would not be done, WCE, stratification and partitioning would experience an increase in the number of steps, resulting from potential lower bound increases after adding new information.

- **Lower bound step size** - As with the number of lower bound steps, the step size roughly indicates the quality of the found cores. These metrics are expected to very strongly correlate, though they may be interesting to look at separately.

  This number is expected to most strongly increase in stratified approaches, and to a lesser extent in WCE and partitioning. The reason for this is the same as the one for the decrease in number of steps.

- **Average Size of Cores** - [4] emphasises the importance of small cores. In the general case, cores that are smaller cause the search space to be smaller (on average), which in turn results in the solver requiring less time to search it[1]. As such, a decrease in this number may be the reason for a decrease in time taken. Additionally, this number can serve as a predictor of performance on instances larger than those in this evaluation; the lower the average core size, the smaller the overhead, and thus the more efficient the search is expected to be.

  This metric is expected to be mostly affected by partitioning, since this technique specifically aims to group highly interconnected objective variables, in order to find smaller cores. WCE and stratification are also expected to improve this metric, due to the increased chance of minimal cores mentioned before. Note that this metric may be biased in cases where many unit cores are found, as occurs especially in the variable-based approach, and potentially in coefficient eliminating approaches.

- **Time Spent on each Task** - All additional features in this thesis require some amount of computational resources when applied. The total time taken provides some information on overall efficiency, but a comparison of the time spent on one of the additional features and that spent traversing the search space may provide new insights. Even if the overall time increases, it might be the case that the search time decreases, meaning the additional feature has a large positive effect. In these cases, a more efficient implementation is worth pursuing, as this can significantly improve the performance.

  Most additional features are expected to be relatively efficient, with the exception of partitioning. Especially on larger instances, the time spent generating the partitions is expected to increase sharply, but result in a sharp decrease in search time. The other features are unlikely to consume a significant amount of time themselves, though they are expected to decrease the search time.

- **Number of Visited Nodes** - During search, many nodes of the search tree are visited, and many other nodes are pruned. The number of visited nodes is strongly related to the number of propagations done by the solver, and can as such be used as an indicator of the quality of the constraints and variables used, as well as the clauses learned based on these factors. A lower number of visited nodes corresponds to stronger

[1]To see why this is the case, consider two reformulation variables $o^1_\kappa = x_1 + x_2$; $o^2_\kappa = x_1 + x_2 + ... + x_{10}$. When assuming $o^1_\kappa = 1$, only two partial assignments are possible: $\{(x_1 = 1), (x_2 = 0)\}$ and $\{(x_1 = 0), (x_2 = 1)\}$. When assuming $o^2_\kappa = 1$ on the other hand, ten partial assignments are possible, thereby creating more branches to be searched.

inferences; this indicates the solver can better extract valuable information from the problem, presumably making it more effective.

Especially hardening is expected to decrease this number, as it efficiently decreases domains of certain variables whenever a solution is found. The other three features commonly consider relaxations, thereby prompting an increase, though their positive effects might outweigh this number - especially in the case of WCE, where relaxations are often relatively close to the original problem throughout the entire process.

Additionally, the following feature-specific data points are extracted:

- **WCE: Number of cores before reformulation** - WCE is able to extract multiple, disjoint cores before a reformulation occurs. This number of cores between reformulations is a good quantifier of the effect WCE has. An average of one core per reformulation indicates no effect at all[2]; higher values signify a larger (positive) effect. This average is used to determine the efficacy of WCE in the different approaches.

- **Stratification: Slope of lower bound steps** - The goal of stratification is to discover cores with high weight earlier than cores with low weight. To measure this, the slope of the step sizes - which are already monitored - is calculated using a linear regression model[3]. A non-stratified approach presumably results in a slope near 0, while a stratified approach is expected to have a strongly negative slope. A steeper (negative) slope indicates a larger success.

- **Hardening: Unhardened fraction of domain** - When a (suboptimal) solution is found, hardening adds constraints to limit variable domains appropriately. In order to quantify this effect, the fraction of the domain that remains is calculated (the unhardened, or "soft" fraction), using the product of these fractions as a final value. A smaller fraction indicates a more effective hardening step. This value is, of course, 1 in non-hardened approaches.

- **Partitioning: Slope of core sizes** - Partitioning allows large cores to be discovered only quite late in the solving process, by splitting the objective variables into small sets. The effect of this is quantified by calculating the slope of the core sizes, as was done for the lower bound steps. This slope is likely to be near 0 for non-partitioned instances, and increasingly positive in cases where partitioning is more effective.

4.5 Aggregation of evaluation results

The results of every unique run, i.e. every unique solver-instance combination, are combined into a single file to facilitate its analysis. Based on this file, several graphics are created; these are described in this section, and used in chapter 5.

[2]This even indicates a negative effect; the search time is partially spent finding solutions instead of cores.

[3]Note that the model fits a function of form $ax+b$, where only a is presented; the bias term is desired, as the early steps are expected to be very large (and thus far from 0), slowly decreasing over time.

Based on the times taken to reach a correct conclusion, a cactus plot can be created; such a plot visualises the elapsed time (Y-axis) as a function of number of instances which have been correctly solved (X-axis). This plot helps to directly compare the performance of the (twenty-four) variants of the core-guided solver, as well as compare their performance to a baseline solver. We expect this plot to show a notable variation between the solvers, based on which we can conclude what variants perform the best, and which perform worse. This divergence is expected to be bigger than that in [18], since the variants used here are more heterogenous; a solver with no additional features is likely to be significantly slower than one with additional features. Additionally, the different reformulation and weight handling approaches are expected to cause a notable divergence among solvers with the same features (or lack thereof), causing even more diversity. The baseline solver is expected to perform slightly better than most, though not all, core-guided solvers; [18] suggests comparable performance, a result most likely applicable to our research as well. An additional factor might present itself in the selected instances, as certain problem families are presumably less suitable for core-guided search, as described briefly in section 4.2.

The remaining data points are shown in two different types of table. The first type of table shows the exit statuses returned by the different solvers. This allows us to perform a quantitative comparison of the solver variants, and compare these to the baseline solver. Based on this, we can quantify the positive or negative effects of the additional features on the most basic level: the number of solved instances.

The second type of table shows the mean of the collected data points, for some set of core-guided solver variants, aggregated over those problem instances which were solved by all variants in the table. This allows an easy comparison between the different approaches and techniques, on key metrics which are expected to improve (or deteriorate) for their specific configuration. These comparisons allow us to draw more qualitative conclusions on the different features, as well as on the different approaches used for reformulation and for weight handling. The expected results from these tables have been previously described in section 4.4 and section 4.3, where the used data points are described.

Finally, the input file metrics and solver performance can be combined into a box plot. This plot shows whether solvers are, on average, better at solving larger or smaller problems, and problems with a larger or smaller objective function. The data points collected for this purpose are described in section 4.2.

Chapter 5

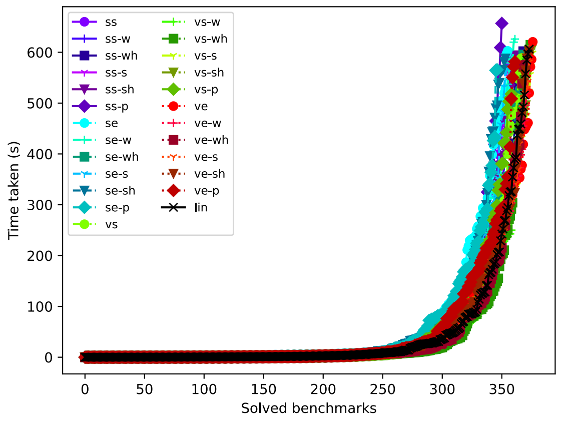
Results

This chapter contains the results from the experiments described in chapter 4, as well as their corresponding conclusions. First, the cactus plots are used to derive some general conclusions in section 5.1. Secondly, tables are used to arrive at more precise conclusions about various solver variants; we consider the base variants (section 5.2), the performance of each individual feature over all four base variants (section 5.3 for WCE, section 5.4 for stratification, section 5.5 for hardening, section 5.6 for partitioning), and the performance of the four base variants over all features (section 5.7 for slice-based weight splitting, section 5.8 for slice-based coefficient elimination, section 5.9 for variable-based weight splitting, and section 5.10 for variable-based coefficient elimination). Finally, we look at several file characteristics and their effect on the solver. This is done by first looking at files with single-variable objective functions in section 5.11, after which other factors and files are considered in section 5.12.

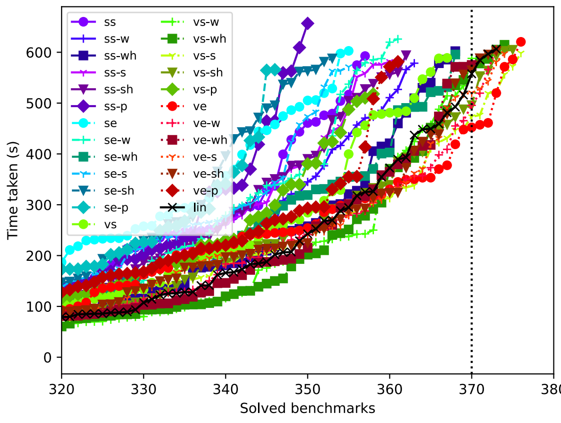
Most tables used in the following sections are modified fragments of Table A.1, available in the appendix. In these tables, the solvers are denoted by a string of two to five letters. The linear search solver is denoted "lin"; the core-guided variants of the solver are denoted by characters representing their reformulation (first letter, "s" for slice-based and "v" for variable-based) and weight handling (second letter, "e" for coefficient elimination and "s" for weight splitting) approaches, as well as additional features (after the dash; "s" for stratification, "w" for WCE and "h" for hardening). Data from the full table may be referenced throughout this chapter, in cases where it lightly supports a conclusion, but is not fully relevant otherwise. In order to draw stronger conclusions, the tables in this chapter use data over only the problem instances which were solved by all solvers in the table, causing values to differ slightly from those in Table A.1.

Note that the time spent on each task is mentioned in section 4.4. Several tasks were identified for this purpose, of which only two are represented in a number of tables - the other tasks either took approximately constant time in each case, or took too little time to draw relevant conclusions on. The tasks represented in the table are the time spent traversing the search space - the "solver time" - and the time spent executing additional features - the "special time". The latter is only relevant in the case of partitioning, where most cases still spend less than one second on it.

5.1 General conclusions



(a) The plot at normal scale, showing comparable performance for all solvers.



(b) An enlarged fragment, highlighting the individual differences.

Figure 5.1: A cactus plot, showing the time taken (Y-axis) as a function of the number of instances solved before this time (X-axis). In this plot, an instance is considered solved if either optimality or unsatisfiability was proven. Most core-guided solver variants perform comparable to one another. We can clearly see that variable-based variants outperform slice-based variants. Linear search performs comparable to the better core-guided variants. Most variants experience a speedup from additional features, but some suffer from decreased performance when features are added. A vertical dotted line has been added to (b) to visually compare between these results, and those later presented in Figure 5.2(b).

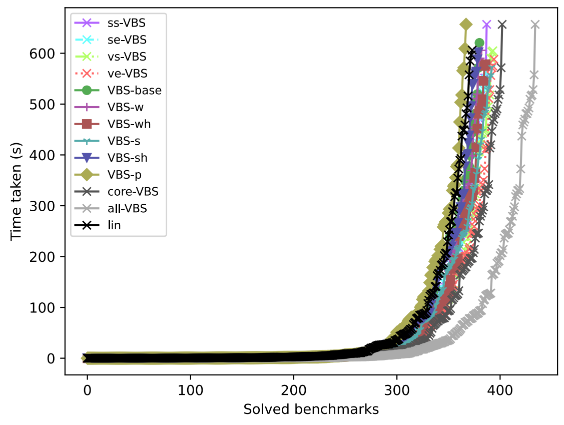
In Figure 5.1(a), it is clear to see that most variants of the core-guided solver perform somewhat comparable to one another, and to the linear search solver. When enlarging a part of the plot in Figure 5.1(b), some general conclusions can be drawn.

The first and foremost observation from Figure 5.1 is the fact that several of the better performing solvers exceed the linear search solver. This shows that core-guided search is indeed a competitive solving paradigm, even when few additional features are used to improve performance - a promising result for the future of core-guided search in CP.

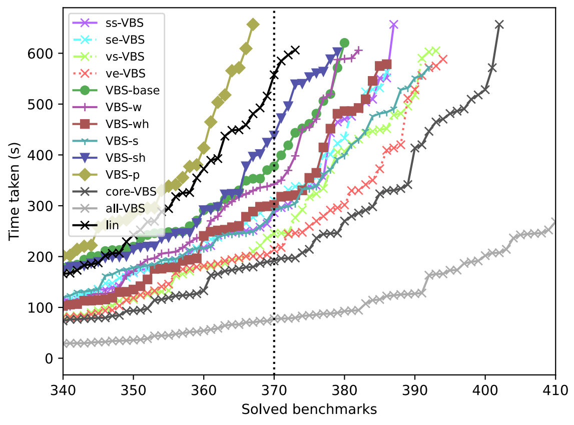
A second observation is the fact that, quite consistently, slice-based variants perform worse than variable-based variants. This is in line with the conclusions from [18]; similar to the fact that between the two slice-based variant, weight splitting performs better on average. However, it stands out that this effect seems to be inverted for the variable-based variants; in [18], variable-based weight splitting outperforms variable-based coefficient elimination, while Figure 5.1 seems to show the opposite. A closer look shows that additional features can decrease this difference; it stands to reason that these features combined, as in [18], cause the weight splitting variant to outperform the coefficient eliminating variant. The four approaches are compared to one another in section 5.2, and individually discussed in the context of additional features in section 5.7 through section 5.10.

Thirdly, we see that most of the used features are beneficial to the core-guided solver variants, with two notable exceptions. This is thus largely in line with our expectations. We see a significant performance improvement for three of the variants when WCE is applied, which is increased further when hardening is added. Stratification, even when combined with hardening, seems to contribute only minor performance improvements to these variants. As the first of the two mentioned exceptions, partitioning decreases performance nearly consistently; contrary to our expectations. Secondly, the variable-based coefficient eliminating variant seems to experience a performance decrease from all features, to some extent. These findings indicate that features are not always beneficial. The additional features are further discussed in section 5.3 through section 5.6.

An additional fact to note is the fact that intersections occur in Figure 5.1. These occur when one variant is more effective on smaller instances, while the other variant outperforms it on larger instances. A prime example of this is the variable-based coefficient eliminating variant; this variant is relatively slow (compared to its variants with additional features) on the first 350 instances, but is eventually able to solve more instances than any other variant.



(a) The plot at normal scale, showing that the ‘all-VBS‘ solver exceeds all others, by a margin.



(b) An enlarged fragment, highlighting the individual differences between the other VBSs.

Figure 5.2: A cactus plot showing several virtual best solvers (VBSs) and the linear search solver. It stands out “all-VBS” exceeds all others, including “core-VBS”, by a margin; this shows that the linear search solver solves different types of problems than the core-guided solver variants. The VBSs for different features are further apart than those for different reformulation and weight handling approaches, indicating that the features have a larger effect on which instances can be solved. A vertical dotted line has been added to (b), at the same location as the one in Figure 5.1(b).

To provide a different perspective on these results, Virtual Best Solvers (VBSs) have been generated over a total of twelve sets of solvers. These are plotted in Figure 5.2. Four

VBSs are shown for the combinations of reformulation and weight handling approaches, and six VBSs are shown for the used (combinations of) additional features. These ten VBSs adhere to the same naming scheme as used for the regular solvers, using "VBS" in place of the varying properties; with the exception of "VBS-base", representing the VBS over variants with no additional features. Additionally, a VBS is shown over all core-guided variants ("core-VBS"), and one over all solvers used in this evaluation ("all-VBS"). For comparison, the linear search solver is also shown.

The first observation from Figure 5.2 is the fact that "all-VBS" exceeds all other (virtual best) solvers, including "core-VBS", by a margin. This shows that the sets of instances solved by the linear search solver and core-guided variants have a relatively large symmetric difference. This advocates that different types of problems are best solved by different types of solvers. Note again that "all-VBS" is the VBS over a single linear search solver and "core-VBS", meaning that the entire discrepancy is caused by one solver; other VBSs are relatively closer to each of their constituents. This conclusion is covered further in section 5.11, in the context of specific input files.

A close look at Figure 5.2(b) shows that all VBSs, except "VBS-p", outperform all actual solvers. This indicates that the components of the VBSs solve quite diverse sets of problems. As such, we can conclude that the used reformulation and weight handling approaches, as well as the used features, affect which problem instances are solved. Since the feature-based VBSs are much farther apart than those based on reformulation and weight handling approaches, we can also conclude that the set of solved instances depends more strongly on the used features.

5.2 Base variants

Solver	ss	se	vs	ve	lin
# Solved	347	345	357	**366**	359
# Timed out	400	399	385	**378**	385
# Out of RAM	24	27	29	27	**23**
# Proven UNSAT	11	11	11	11	**15**

Table 5.1: A comparison of the four variants with no additional features, including the linear search solver, in terms of reasons for termination. The best value in each column is marked by a bold font. Note that in the first and fourth columns, this is the largest value, while in the second and third columns, it is the smallest.

From Table 5.1, we can confirm the earlier conclusion that slice-based variants perform worse than variable-based variants. It is worth noting that the same eleven instances were proven unsatisfiable by all four core-guided variants.

Looking at Table 5.2, we see that variable-based variants perform better because of the relatively low average size of cores. We have previously briefly explained the importance of small cores in section 4.4 (see also [4]). Slice-based variants cause remainders to be present in the objective function, thereby using more assumptions and thus allowing larger

Solver	ss	se	vs	ve
# of LB steps	107.9	191.3	**107.4**	188.5
LB step size	**10.8**	1.10	**10.8**	1.15
Core size	8.72	6.29	6.02	**3.68**
Total time (s)	31.3	38.5	**24.8**	27.7
Solver time (s)	30.2	37.1	**24.0**	26.9
# of nodes	72k	75k	60k	61k

Table 5.2: The data points describing the optimisation process of the four base variants. For each metric, except for the number of nodes, the best value has been made bold; this may be either the largest or the smallest value, depending on the metric. The number of nodes has been excluded, as changes to it are not necessarily improvements, nor deteriorations. Data from 334 instances was used; this is the number of instances solved to optimality by all variants in this table.

cores to be found. Additionally, certain relations are harder to efficiently extract in slice-based variants[1]. In turn, the resulting reformulation variables may allow for more unique assignments; this increases the number of decisions, and thus the number of nodes to be searched.

Note that the average core sizes from the coefficient eliminating variants are significantly lower than their weight splitting counterparts, while the time taken increases. This is because these variants produce variables with non-continuous domains, which may produce many unit cores when encoded as continuous domains - as is done in our core-guided solver. Choosing to encode these variables using sparse domains may improve performance when relatively little elements are in the domain - as inference can be done more efficiently - but may decrease performance when many elements are present - since the domain representation then takes up more space. The effect extends to the number of lower bound steps and lower bound step size; these unit cores increase the bound by only 1.

Another important factor to note in the coefficient eliminating variants is the performance: the variable-based variant performs better, while the slice-based variant performs worse than their respective weight splitting counterparts. This is because the workings of coefficient elimination combine particularly well with variable-based reformulations, while combining badly with slice-based reformulations. Despite possibly creating variables with non-continuous domains, variable-based coefficient elimination can directly model cases where a fixed set of variables is responsible for several separate MUSes - something that is more complex for all other variants. Meanwhile, slice-based coefficient elimination can introduce very sparse domains; the slice-based approach is specifically bad at handling these, since each value corresponds to a slice, even if the variable cannot be assigned the represented value. This strongly increases overhead.

When instead considering weight splitting, we see that those variants often cause reformulation variables to have weight higher than one, allowing them to increase the lower

[1]Consider the relation $a+b+c \geq n$. A variable-based approach would find a single multi-valued core, while the slice-based approach require n multi-valued cores.

bound by more than one when appearing in a core. In coefficient elimination, the increase is often only one; this is due to reformulation variables, which have weight one, appearing in cores. As a result, solver variants applying weight splitting experience a larger step size, and consequently smaller number of steps.

A different observation from Table 5.1 is that core-guided solvers seem to be worse in proving unsatisfiability than the linear search solver. This is most likely due to the assumptions constraining the search space, requiring many cores - with the resulting additional overhead - to be found before the proof of unsatisfiability can be found. Note that this could be largely mitigated by performing an initial feasibility check before placing any assumptions, something which is currently not done. Adding this would increase performance on unsatisfiable instances, at the cost of a slight performance decrease.

Strengths and weaknesses In order to better understand which core-guided approaches are suitable for what problem instances, we have analysed the problem instances which were solved by one variant, but not by the other. The findings of these comparisons have been summarised below.

Reformulation approach The most evident benefit slice-based reformulations offer over variable-based reformulations is the possibility of remainders appearing in cores, which would otherwise require reformulation variables. This can significantly reduce overhead in certain cases, especially if the effect occurs multiple times on the same (original) variables - requiring multiple codependent reformulation variables in a variable-based reformulation approach.

Besides this effect, the main difference in performance is due to the advantage of the variable-based approach, and is the result of the way each approach handles the occurrence of multiple MUSes on the same set of variables. In the variable-based case, these variables are simply combined into a single new variable - especially if they have equal weights, or if coefficient elimination is active. In the slice-based case, this can be much less straightforward, as different combinations of values are encoded by different variables. As an example, consider $a+b \geq 2$ where $w_a = w_b = 1$:

$$o_{\kappa_1} = \lceil \lfloor a \rfloor \rceil_0^1 + \lceil \lfloor b \rfloor \rceil_0^1 \; ; \; o_{\kappa_2} = \lceil \lfloor a \rfloor \rceil_1^2 + \lceil \lfloor b \rfloor \rceil_1^2 + \lceil \lfloor o_{\kappa_1} \rfloor \rceil_1^2$$

Depending on the bound, whether the cores are minimal, and other factors, subsequent cores may become increasingly more complex. Also note that this effect may be amplified by coefficient elimination, as the non-continuous domains can cause several slices to appear which correspond to infeasible values.

Weight handling approach Weight splitting is especially effective when the weights of variables share a divisor. In such cases, all reformulation variables have a multiple of this divisor as weight, instead of 1 - as introduced by coefficient elimination. Coefficient elimination would additionally cause sparse domains[2] in this type of problem, thereby often

[2]As mentioned, an alternative approach would be to encode the domains of coefficient elimination variables sparsely; however, this could cause domain encodings to become overly large in some cases.

requiring additional time to find the meaningless unit cores introduced by these domains. Of course, an alternative approach for these cases would be to divide all weights beforehand, thereby mitigating this effect.

Coefficient elimination exceeds weight splitting if the weights of objective variables are close, but not equal. In such cases, weight splitting introduces terms with low (residual) weight. These low-weighing terms cause the lower bound to increase relatively slowly, and prevent the cores found to utilize large splits of the weights of the other variables. Coefficient elimination may need to spent some additional time on the aforementioned cores due to non-continuous domains; however, these do not introduce additional overhead, and may thus be more effective in the long run.

In addition to these effects, we saw in Figure 5.1 that coefficient elimination outperforms weight splitting in the variable-based context, while being surpassed by it in the slice-based context. As briefly mentioned, this is in part because its combination with the variable-based approach allows it to capture multiple MUSes effectively. On the other hand, the slice-based approach generates reformulation variables with strongly non-continuous domains when combined with coefficient elimination; as mentioned, this causes many slices which do not directly correspond to a valid value, incurring additional overhead as these need to be handled in some way.

A different factor occasionally affecting performance, both positively and negatively for both approaches, is the order of assumptions. This can affect the cores that are found, as well as their order. On one hand, weight splitting causes variables with residual weight to remain in the list of assumptions, which causes them - and, as a result, the propagators they trigger - to be processed earlier than the new assumptions[3]. If the found core was non-minimal, these residual assumptions may still result in the (harder to find) minimal core. In other cases, these assumptions are considered before those of the reformulation variables, and may thus cause additional cores using only original variables, decreasing overhead. On the other hand, coefficient elimination removes all assumptions associated with a core, making sure early cores are functionally more independent by not sharing elements. Additionally, this prevents low residual weights from appearing in cores, thereby possibly affecting the lower bound step size of these cores.

5.3 WCE

From Table 5.3, we observe that all solvers except variable-based coefficient elimination are able to solve more instances when equipped with WCE. Additionally, we see that all solvers were able to prove unsatisfiability for three additional instances - which are again the same instances for each of the four variants. Note that the instances which no longer ran out of memory in variable-based weight splitting were not solved; they instead resulted in timeouts. To explain these observations, we look at Table 5.4.

In Table 5.4, we see that WCE extracts the most disjoint cores in slice-based coefficient elimination, followed by slice-based weight splitting. This is to be expected, due to

[3]This depends on the implementation details. Solvers using a set or map of assumptions, rather than a list, may experience this effect differently, as there may not be a set order in which the assumptions are considered.

Solver	ss	ss-w	se	se-w	vs	vs-w	ve	ve-w
# Solved	347	350	345	**348**	357	360	366	360
# Timed out	400	**394**	399	394	385	384	378	381
# Out of RAM	24	24	27	26	29	**24**	27	27
# Proven UNSAT	11	**14**	11	**14**	11	**14**	11	**14**

Table 5.3: A comparison of the variants applying WCE to their base variants, in terms of reasons for termination. The largest relative improvements have been marked by a bold font. Note that the white columns are base variants, and the gray columns are variants applying WCE.

Solver	ss	ss-w	se	se-w	vs	vs-w	ve	ve-w
# of LB steps	77.3	73.1	163.1	153.6	76.8	**69.9**	160.2	152.7
LB step size	11.1	11.4	1.08	1.44	11.1	13.3	1.14	**2.49**
Core size	8.55	8.07	6.21	**5.85**	5.85	5.79	3.62	3.66
Total time (s)	23.8	20.4	30.8	**24.3**	17.3	17.8	21.2	21.1
Solver time (s)	23.0	19.6	29.7	**23.3**	16.5	17.0	20.3	20.2
# of WCE cores	-	21.2	-	**49.2**	-	5.24	-	4.09
# of nodes	56k	74k	59k	82k	49k	60k	49k	147k

Table 5.4: A comparison of the variants applying WCE to their base variants, showing several important metrics. The largest relative improvements have been marked by a bold font, where applicable. Note that the white columns are the base variants, and the gray columns are the variants applying WCE. Data from 323 problem instances was used; this was the number of instances solved to optimality by all variants in this table.

remainders being present, and the fact that relatively little information is removed at once; only single slices are removed, while the variable-based approach remove entire variables at once. Note again that coefficient elimination causes non-continuous domains, which can generate relatively many slices; if these appear in small, non-unit cores, they can amplify the number of independent cores compared to weight splitting. The smaller cores resulting from WCE can utilize their weight slightly better, as reformulation variables with low weights are absent from the assumptions, and thus from the cores. This can increase the lower bound step size and consequently decrease the required number of steps.

The described removal of assumptions causes cores to be disjoint and (on average) smaller, decreasing overhead. Additionally, this relaxes the problem, which expands the search space, and in turn increases the chance of finding a proof of unsatisfiability; this causes an increase in instances proven unsatisfiable. The increased freedom has a less positive impact on the instances solved to optimality, as observed in the variable-based coefficient eliminating solver: the removed assumptions heavily expand the search space, causing a spike in the number of nodes, which can in some cases not be traversed within the time limit. This has been verified using the raw data; most of the instances which timed out due to WCE being enabled were able to remove most, if not all, variables before an intermediate solution is found.

Our expectation that the slice-based approach would benefit more from WCE than the variable-based approach seems to be correct; however, the expectation that the variable-based approach would still benefit is partially wrong. Furthermore, we expected weight splitting to be able to extract more cores than coefficient elimination; this is the case in the variable-based variants, but slice-based variants show the opposite.

Strengths and weaknesses WCE can significantly improve performance by preventing cores containing reformulation variables from being found until later in the search process. This reduces overhead by causing reformulation variables to be as independent of one another as possible. Additionally, the removal of assumptions over time increases the chance of finding minimal cores - especially ones that would normally contain an (unnecessary) reformulation variable; this can reduce overhead even further. Additionally, these smaller cores may have higher weighing elements, or lack low-weighing reformulation variables, thereby aiding the lower bound increase. A separate though less prominent factor affecting the performance of WCE is the fact that, in some cases, the solver is able to increase the bound of the delayed assumption. This can save some time, as the corresponding unit cores do not need to be extracted and handled separately, and are guaranteed to not be part of other, non-minimal cores.

Instead of these improvements, WCE may also be detrimental; by relaxing the assumptions, the search space increases. This means that more nodes need to be searched, taking more time. One example of this is when large cores are found, resulting in many assumptions being removed at once, thus relaxing the problem much further than helpful; another example is when feasible solutions are very scarce, as the time needed to find a solution to the relaxed problem is high compared to the time normally taken to extract additional cores, in a restricted search space. A similar situation can also occur with cores being present, as these can also be hard to find in the expanded search space.

Finally, a specific problem type worth mentioning is that with an objective function of the form $N * x + M * y$ with $N >> M$. There are several cases in which performance on such a file improves due to WCE, but also several cases in which performance decreases. In some cases, the application of WCE prevents the aforementioned effect where reformulation variables immediately appear in new cores, reducing overhead and thus speeding up the process. However, there are cases in which a solver is able to find many of such cores without WCE, meaning WCE only contributes the negative impact of the expanded search space.

5.4 Stratification

In Table 5.5, we see that only the variable-based weight splitting variant was able to solve strictly more instances due to stratification. Most variants ran out of time on several instances for which they previously ran out of memory, and in the variable-based weight splitting case even for two instances it had previously solved. It should be noted that all

Solver	ss	ss-s	se	se-s	vs	vs-s	ve	ve-s
# Solved	347	347	345	345	357	**363**	366	364
# Timed out	400	397	399	402	385	**380**	378	383
# Out of RAM	24	24	27	24	29	**25**	27	24
# Proven UNSAT	11	**14**	11	11	11	**14**	11	11

Table 5.5: A comparison of the stratified variants to their base variants, in terms of reasons for termination. The largest relative improvements have been marked by a bold font. Note that the white columns are the base variants, and the gray columns are the variants applying stratification.

Solver	ss	ss-s	se	se-s	vs	vs-s	ve	ve-s
# of LB steps	94.8	87.9	180.0	**130.9**	94.3	87.4	177.2	132.3
LB step size	11.0	16.4	1.11	**12.2**	11.0	16.4	1.17	12.2
LB step slope	-0.45	-18.7	-0.15	**-18.1**	-0.46	-18.7	-0.15	-18.1
Core size	8.75	**8.04**	6.29	6.21	6.03	5.62	3.66	3.62
Total time (s)	27.4	25.8	35.0	**29.3**	21.5	21.9	25.3	23.7
Solver time (s)	26.3	24.8	33.5	**28.1**	20.6	21.1	24.4	22.8
# of nodes	63k	148k	66k	149k	56k	140k	56k	141k

Table 5.6: A comparison of the stratified variants to their base variants, showing several important metrics. The largest relative improvements have been marked by a bold font, where applicable. Note that the white columns are the base variants, and the gray columns are the variants applying stratification. Data from 327 problem instances was used; this was the number of instances solved to optimality by all variants in this table.

solvers were, due to stratification, able to prove unsatisfiability on three[4] additional instances; however, the coefficient eliminating solvers reached timeouts on another three[5] unsatisfiable instances. The other observed changes are explained by looking at Table 5.6.

We clearly see that, in Table 5.6, the lower bound slope became strongly negative for all four stratified variants variants, indicating that stratification had a large effect. This effect mainly manifests in an increased lower bound step size - and associated decreased number of steps - and, to a lesser extent, in smaller cores. Additionally, the number of nodes increased significantly due to relaxations being considered. The changes are approximately equal in size for all variants, with the exception that the lower bound step size increased much more for the coefficient eliminating variants. This last factor is because it may now be able to find unit cores on original objective variables, instead of finding these after reformulation; this allows the solver to increase the lower bound by the original weight, rather than by 1.

The aforementioned larger step size and smaller core size are mostly because stratified solvers are able to find small cores consisting of only high-weighing variables, instead of

[4]These were the same three instances across all four solvers.

[5]These were again the same three instances for both solvers.

the larger (non-minimal) cores found by non-stratified solvers, containing low-weighing elements as well. This affects mainly variable-based weight splitting, causing a performance increase. It should be noted that the effect is also present in the slice-based variant; the time taken decreased. However, the performance increase is not enough to make a difference on the larger instances. Besides this effect, stratification also considers a relaxation from the start, which only becomes less relaxed when new solutions are found. In the case of unsatisfiable instances, this can be a large benefit, as fewer assumptions are present to prevent the proof of unsatisfiability from being found.

Despite the described benefits, a performance improvement is small or even absent in three of the variants. For the variants applying coefficient elimination, this is due to the reformulation variables. In the first few strata, when the variables of most problems still have high weights, coefficient elimination can already introduce variables with weight 1. These low weights can quickly diminish all positive effects introduced by stratification. In fact, the appearance of these terms caused three timeouts on instances that were otherwise proven unsatisfiable. Additionally, the fact that the problem is relaxed can also cause an increase in search time, due to the additional nodes needing to be searched. This effect has also been covered in section 5.3; the node increase is clearly observed in Table 5.6, and the associated time increase is the reason for the additional timeouts of the variable-based coefficient elimination solver in Table 5.5.

Our expectation concerning stratification were largely incorrect. The coefficient eliminating variants experienced no positive (slice-based) or even a negative (variable-based) effect, while we expected these to benefit more than the (somewhat successful) weight splitting variants. It is also incorrect that slice-based variants experience a larger benefit; the advantages over variable-based variants are not strong enough to counteract the additional overhead incurred at other points of the solving process.

Strengths and weaknesses Most of the performance decrease introduced by stratification is either because the search space is too unrestricted, or because it is restricted in an ineffective way. An example of the first case is when the initial stratum contains only one variable, causing the solver traverse a large part of the search space before finding either a solution or unit core. An example of the second case is when the solver takes a long time to find a solution under the current assumptions, while the addition of the next stratum would allow it to very easily extract one or several cores, after which a (different) solution can be found more easily. A less prominent reason for stratification decreasing performance, mostly in coefficient elimination, is the existence of reformulation variables (or residual terms) with a low weight.

On the other hand, stratification can also increase performance; this happens because in these cases, different - often smaller - cores are found. These cores lack elements with a low weight, thus causing the lower bound to increase by a larger amount than if all strata had been present. This effect is mostly present in the variable-based weight splitting variant, as this variant has no remainders with original weights late in the process, nor does it introduce reformulation variables with low weight early in the process.

One additional factor that should be noted is that several instances experienced a performance change despite stratification not modifying the instance in any meaningful way. The reason for this is an implementation detail, which changes the ordering of variables[6] when stratification is active. This could have either a positive, negative or insignificant effect.

5.5 Hardening

Since hardening requires intermediate solutions, it can only be considered in the context of an additional feature that generates those. As such, we have combined it with both WCE and stratification. In order to draw general conclusions based on both variants of hardening, these first need to be inspected individually.

5.5.1 WCE-based hardening

Solver	ss-w	ss-wh	se-w	se-wh	vs-w	vs-wh	ve-w	ve-wh
# Solved	350	355	348	**355**	360	362	360	357
# Timed out	394	389	394	**387**	384	383	381	384
# Out of RAM	24	24	26	26	24	24	27	27
# Proven UNSAT	14	14	14	14	14	13	14	14

Table 5.7: A comparison of the variants applying hardened WCE to their non-hardening variants, in terms of reasons for termination. The largest relative improvements have been marked by a bold font, if any improvement was present. Note that the gray columns are the variants applying hardening, and the white columns are the variants applying only WCE.

Solver	ss-w	ss-wh	se-w	se-wh	vs-w	vs-wh	ve-w	ve-wh
# of LB steps	75.7	75.5	148.2	144.6	72.8	**67.4**	148.8	140.4
LB step size	11.6	11.6	1.52	1.84	13.8	14.0	2.63	**4.03**
Core size	7.94	7.79	5.75	**5.35**	5.69	5.65	3.52	3.45
Total time (s)	22.1	19.4	26.9	**23.5**	19.6	19.4	24.4	21.8
Solver time (s)	21.3	18.6	25.9	**22.6**	18.8	18.7	23.5	21.0
# of WCE cores	23.8	**26.6**	45.3	50.3	5.04	5.06	4.07	4.06
Soft fraction	-	0.74	-	0.74	-	0.67	-	**0.66**
# of nodes	61k	62k	70k	66k	74k	74k	162k	151k

Table 5.8: A comparison of the variants applying hardened WCE to their non-hardening variants. The largest relative improvements have been marked by a bold font, where applicable. Note that the gray columns are the variants applying hardening, and the white columns are the variants applying only WCE. Data from 332 problem instances was used; this was the number of instances solved to optimality by all variants in this table.

The main differences in Table 5.7 are in the first two rows; the latter two show (almost) no differences. The slice-based variants experience a notable improvement; the variable-

[6]The original list of variables is traversed from back to front, while building the stratum from front to back. This thus inverts the order of assumptions.

based variants change less, with the weight splitting variant experiencing an increase, and the coefficient eliminating variant experiencing a decrease in performance.

According to Table 5.8, the changes in slice-based variants are in part caused because hardening allows even more disjoint cores to be extracted through WCE. This effect is hardly present in variable-based variants, as cores remove a larger amount of information (as mentioned in section 5.3), often leaving too little information for the effect to occur. The extraction of additional cores is possible because the intermediate solutions normally found are pruned, as they have a higher objective value than the best solution so far. This may result in no solutions being present in the search space at that point, thereby causing the solver to exhaust the search space further before re-adding information. This effect occurs in approximately one hundred instances, for each of the four variants. As a result, the core sizes decrease, and the lower bound steps change slightly - the smaller cores have a lower chance of of containing a small weight, thereby increasing step size and decreasing number of steps.

The pruning of intermediate solutions is also the main reason for the performance decrease of the variable-base coefficient eliminating solver. In fact, this performance decrease is present in all four variants, though it is obscured by a larger performance increase in three of the variants. The aforementioned further exhaustion may result in an increase in the number of nodes - as opposed to the decrease that is expected when parts of the search space are pruned. The additional cores may be relatively hard to discover, meaning much search is done before they are found, in turn resulting in the aforementioned node increase, and a corresponding increase in time taken. This conclusion conflicts with the data in Table 5.8; however, the effect is observed in manually investigated instances. For each of the four variants, approximately five instances resulted in a timeout due to intermediate solutions being pruned directly.

Despite these effects of hardening, Table 5.8 also shows that hardening affected the variable-based approach more strongly than the slice-based approach. This is because of the hardening of reformulation variables. The domains of slice-based reformulation variables are often too limited to be hardened, while variable-based reformulation variables can have much larger domains - especially when combined with coefficient elimination.

5.5.2 Stratification-based hardening

According to Table 5.9, the stratified variants experience a relatively minor performance change from hardening. This can be easily explained by looking at the idea behind stratification: high-weighing variables, which are most affected by hardening, are considered first, and the associated assumptions are never fully removed.

The fact that high-weighing variables are considered first also generally causes intermediate solutions to become progressively better; only approximately ten solved instances used intermediate solutions which would be pruned by hardening, for each of the four variants. This means that the first effect of hardening as mentioned in subsection 5.5.1 is much less prevalent, resulting in a minimal number of cases where the search space is exhausted further due to hardening[7]. Increases in the number of nodes are thus much less common,

[7]The effect is not absent; a single instances resulted in timeouts for all four hardening variants, while being

Solver	ss-s	ss-sh	se-s	se-sh	vs-s	vs-sh	ve-s	ve-sh
# Solved	347	**349**	345	343	363	362	364	363
# Timed out	397	**396**	402	404	380	383	383	384
# Out of RAM	24	23	24	24	25	**23**	24	24
# Proven UNSAT	14	14	11	11	14	14	11	11

Table 5.9: A comparison of the variants applying hardened stratification to their non-hardening variants, in terms of reasons for termination. The largest relative improvements have been marked by a bold font, if any improvement was present. Note that the gray columns are the variants applying hardening, and the white columns are the variants applying only stratification.

Solver	ss-s	ss-sh	se-s	se-sh	vs-s	vs-sh	ve-s	ve-sh
# of LB steps	86.8	86.0	130.2	129.6	86.3	**85.1**	131.4	131.6
LB step size	16.9	17.2	12.7	**13.0**	16.9	17.2	12.7	**13.0**
LB step slope	-18.1	-17.8	-17.5	-17.3	-18.1	-17.8	-17.5	-17.3
Core size	7.24	7.20	6.03	**5.67**	5.12	4.90	3.54	3.40
Total time (s)	34.1	34.3	36.6	38.9	28.3	27.9	29.7	**28.4**
Solver time (s)	32.9	33.1	35.3	37.6	27.2	27.0	28.8	**27.3**
Soft fraction	-	0.78	-	0.78	-	0.77	-	**0.76**
# of nodes	162k	141k	163k	149k	151k	125k	152k	130k

Table 5.10: A comparison of the variants applying hardened stratification to their non-hardening variants. The largest relative improvements have been marked by a bold font, where applicable. Note that the gray columns are the variants applying hardening, and the white columns are the variants applying only stratification. Data from 338 problem instances was used; this was the number of instances solved to optimality by all variants in this table.

while parts of domains are still pruned often. These factors combined result in a more structural decrease on this metric - as observed in Table 5.10. Additionally, in the small number of cases in which additional cores could be extracted, these cores were smaller; they were discovered before the addition of the new stratum, meaning fewer terms were available to appear in the core. This caused a minor decrease in average core size. As the new stratum was not yet added, these smaller cores also contained fewer low-weighing terms, resulting in a minor increase in lower bound step size, and associated decrease in number of steps.

Besides the decreased number of nodes and smaller cores, a notable effect of hardening is that the new constraints change which conflicts are found. In a small number of instances, this resulted in a noteworthy effect visible in Table 5.10. However, this effect is not unequivocally positive or negative, and relatively minor.

solved by the non-hardening variants.

5.5.3 Hardening overall

We have seen that hardening is able to improve slice-based variants with WCE quite well, but otherwise has relatively little effect. This is most likely because slice-based WCE generates relatively tight intermediate solutions, which amplify the effect of the underlying WCE. The remainders prevent the search space from becoming unnecessarily large, while the small amount of information extracted with each core allows for a high rate of independency. Stratification experiences little effect from hardening, We have also seen that the additional constraints from hardening can cause additional inferences, which may be either positive or negative - depending on the problem and approach.

The reason hardening was less effective than expected for most of the solvers is twofold: for WCE, hardening may cause the search space to be exhausted further by pruning suboptimal solutions, thereby increasing search time; for stratification, hardening has little effect at all, since both focus mainly on high-weighing variables.

Strengths and weaknesses By pruning the domains of variables, hardening is able to cause a notable speedup in several cases. However, there are also many cases in which search space expansions have a strong negative effect, or in which other, less suitable conflicts are found. Overall, the effect of hardening depends on many different factors of the input problem as well as the solver.

5.6 Partitioning

Solver	ss	ss-p	se	se-p	vs	vs-p	ve	ve-p
# Solved	347	338	345	335	357	346	366	350
# Timed out	400	407	399	411	385	398	378	396
# Out of RAM	24	24	27	24	29	**25**	27	24
# Proven UNSAT	11	**13**	11	12	11	**13**	11	12

Table 5.11: A comparison of the variants applying partitioning to their base variants, in terms of reasons for termination. The largest relative improvements have been marked by a bold font, if any improvement was present. Note that the white columns are base variants, and the gray columns are variants applying partitioning.

At first glance, Table 5.11 shows a relatively strong decrease in number of solved instances, accompanied by an increase in the number of instances that timed out. Several cases that previously ran out of memory now ran out of time first, further increasing this number. The number of instances proven to be unsatisfiable does show a positive impact from partitioning; similar to WCE and stratification, this is the effect of considering a relaxation, namely a single partition. For these instances, a similar effect occurs as seen in stratification: the same two instances were additionally proven unsatisfiable by all four variants, and both coefficient eliminating variants encounter an additional timeout - again on the same instance for both variants.

Solver	ss	ss-p	se	se-p	vs	vs-p	ve	ve-p
# of LB steps	92.4	86.2	179.6	**141.3**	91.9	86.7	176.5	139.6
LB step size	11.4	16.4	1.12	**12.3**	11.4	16.8	1.18	12.4
Core size	7.26	**6.39**	5.96	5.28	4.94	4.56	3.49	3.09
Core size slope	-0.44	-0.15	-0.51	**-0.17**	-0.71	-0.37	-0.79	-0.45
Total time (s)	20.3	23.3	26.2	**25.9**	16.9	22.0	19.0	22.3
Solver time (s)	19.9	21.5	25.4	**24.0**	16.4	20.2	18.6	20.6
# of nodes	57k	96k	59k	99k	49k	88k	50k	86k

Table 5.12: A comparison of the variants applying partitioning to their base variants, showing several important metrics. The largest relative improvements have been marked by a bold font, where applicable. Note that the white columns are the base variants, and the gray columns are the variants applying partitioning. Data from 314 problem instances was used; this was the number of instances solved to optimality by all variants in this table.

Although the effect of partitioning seems somewhat negative, Table 5.12 shows that its application improves some of the key metrics: cores are slightly smaller, which - as seen before - leads to an increase in lower bound step size and associated decrease in number of lower bound steps. Additionally, more unit cores on original variables are found, strongly impacting the initial step sizes in coefficient eliminating variants. We also see that the core size slope is less negative than without partitioning, indicating somewhat smaller cores being found early in the process. A closer inspection of individual files shows that these positive effects warranted a notable speedup in several (smaller) instances; this is also visible from intersections in Figure 5.1(b).

Besides these positive effects, Table 5.11 shows that the overall performance decreased. The reason is that each partition is solved individually, while representing a strong relaxation. As such, a large search space needs to be traversed, which may take a long time - as described in section 5.4 and section 5.3. It may even be the case that a partition contains only a single variable, requiring much search to be done before any usable information is discovered. Furthermore, there are cases in which the initial partitions contain little or no cores, meaning a lot of the early processing time is spent finding solutions; note that these solutions need to be different for most partitions, as all have different and disjoint objective variables.

An additional factor to consider is the amount of time taken by the creation of the partitions. In many cases, this was a relatively small amount; of the solved instances, only a single case was found to exceed one minute (taking 80 seconds), and a total of nine cases exceeded one second. Several of the unsolved instances took a longer time; six instances resulted in a timeout, and an additional seven exceeded one minute. Of the six instances that timed out, only two were solved by any (non-partitioning) variant. Of the seven exceeding one minute, three (taking 7, 5.5 and 5 minutes) resulted in timeouts while non-partitioning variants found a solution. A more efficient approach of creating the partitions may thus slightly improve performance, but even after this the variants would still perform worse than most others, and the overall speedup would be insignificant in most cases.

Contrary to our expectations, partitioning decreased performance for all four variants. We did not consider the performance decrease incurred by the increased freedom and resulting large search space. Note that, as previously mentioned in section 3.4, the used weighing scheme may not be sufficiently suitable; a more appropriate one may improve performance.

Strengths and weaknesses The performance decrease caused by partitioning is in large part due to the found partitions containing little information, as briefly mentioned. This can manifest in two different ways: either the search space is restricted too little, thereby causing the solver to take a lot of time traversing it; or the variables in a single partition do not occur in cores together, meaning a merging step is required before multi-valued cores can be discovered - while solving the individual partitions may still be non-trivial and time-consuming, especially due to the conflicting assumptions requiring unique solutions. In both cases, we argue the partitions are unsuitable for the problem used to create them.

Despite the aforementioned effects, partitioning is able to increase performance for several instances. In these cases, partitioning operates mostly as hypothesized; many small (unit) cores are extracted at the start, and larger cores can be extracted after a merge. There was also a small number of cases where all cores were found within the original partitions, before any merge took place. In these cases, the partitions are much more suitable than most other cases, where partitions contained no cores or were much smaller or larger than expected. The fact that some problem instances result in suitable partitions while others result in unsuitable partitions is most likely due to the weighing scheme not representing the different types of relations accurately. The weighing scheme used is a modified version of one designed for MaxSAT, where relations are much more monotonous than in CP, and variables have domains with a fixed size of two, rather than a size specified by the input. Because of this, the same constraint can have very different effects in different problems. If a more appropriate weighing scheme were to be designed, partitioning could provide a valuable tool for core-guided CP solvers in the future. Such a weighing scheme could incorporate domain sizes, more accurately model global constraints, and take into account the signs as well as the weights in a linear inequality,

5.7 Slice-based weight splitting variants

Solver	ss	ss-w	ss-wh	ss-s	ss-sh	ss-p
# Solved	347	350	**355**	347	349	338
# Timed out	400	394	**389**	397	396	407
# Out of RAM	24	24	24	24	**23**	24
# Proven UNSAT	11	**14**	**14**	**14**	**14**	13

Table 5.13: A comparison of the slice-based weight splitting variants, in terms of reasons for termination. The best value in each column is made bold. Note that in the first and fourth columns, this is the largest value, while in the second and third columns, it is the smallest.

Solver	ss	ss-w	ss-wh	ss-s	ss-sh	ss-p
# of LB steps	85.5	72.8	72.7	70.4	**69.7**	71.5
LB step size	11.6	12.3	12.3	17.8	**18.1**	16.8
Core size	7.52	6.91	6.83	7.01	7.19	**6.46**
Total time (s)	24.1	18.8	**17.9**	24.4	25.0	26.1
Solver time (s)	23.7	18.3	**17.4**	24.0	24.6	24.3
# of nodes	67k	52k	54k	105k	86k	100k

Table 5.14: The data points describing the optimisation process for the slice-based weight splitting variants. For each metric, except for the number of nodes, the best value has been made bold; this may be the largest or smallest value, depending on the metric. The number of nodes has been excluded, as changes to it are not necessarily improvements, nor deteriorations. Data from 321 instances was used; this is the number of instances solved to optimality by all variants in this table.

The data in Table 5.13 shows that this variant strongly improves when equipped with hardened WCE, and experiences smaller positive effects when equipped with the other features - with the exception of partitioning. Note that all features consider quite strong relaxations, resulting in an increase in instances proven unsatisfiable. One reason for the performance changes is seen in Table 5.14: all additional features decrease the average core size - with the associated changes in lower bound steps, as explained in several of the previous sections. However, these changes are not sufficient to explain the performance differences.

This solver variant combines particularly well with hardened WCE because the remainders keep the search space relatively compact (reflected by a decrease in number of nodes in Table 5.14), while the disjoint cores decrease overhead. These factors combined decrease the time spent searching. By pruning suboptimal solutions, hardening amplifies the effect of disjoint cores, as it causes more cores to be extracted before adding reformulation variables. Stratification has a relatively small effect, despite having the least lower bound steps; the speedup is nullified by the fact that the used relaxations heavily expand the search space, thereby increasing time spent traversing it. Hardening can again slightly amplify the positive effects while decreasing the search space, thereby slightly improving performance. Partitioning is able to extract very small cores, but suffers heavily from the expanded search space. Additionally, numerous problems generate many, small partitions, which are non-trivial to solve while containing little or no cores - thereby increasing time taken without contributing to the problem.

5.8 Slice-based coefficient eliminating variants

From Table 5.15, we can see that only WCE seems to have a significant positive effect; this feature, and its hardened variant, caused a notable performance increase. The other approaches - stratification and partitioning - mainly increased the number of timeouts. Note that the number of instances proven unsatisfiable is relatively low compared to the weight splitting variant, for both stratification and partitioning; this is because of the reformulation

Solver	se	se-w	se-wh	se-s	se-sh	se-p
# Solved	345	348	**355**	345	343	335
# Timed out	399	394	**387**	402	404	411
# Out of RAM	27	26	26	**24**	**24**	**24**
# Proven UNSAT	11	**14**	**14**	11	11	12

Table 5.15: A comparison of the slice-based coefficient eliminating variants, in terms of reasons for termination. The best value in each column is made bold. Note that in the first and fourth columns, this is the largest value, while in the second and third columns, it is the smallest.

Solver	se	se-w	se-wh	se-s	se-sh	se-p
# of LB steps	176.8	156.8	155.0	112.9	**112.6**	133.5
LB step size	1.09	1.54	1.87	13.3	**13.6**	12.5
Core size	5.98	5.61	5.25	5.84	5.54	**5.24**
Total time (s)	31.1	21.6	**20.8**	25.4	27.2	29.0
Solver time (s)	30.3	20.9	**20.2**	24.9	26.7	27.0
# of nodes	69k	61k	58k	108k	91k	101k

Table 5.16: The data points describing the optimisation process for the slice-based coefficient eliminating variants. For each metric, except for the number of nodes, the best value has been made bold; this may be the largest or smallest value, depending on the metric. The number of nodes has been excluded, as changes to it are not necessarily improvements, nor deteriorations. Data from 313 instances was used; this is the number of instances solved to optimality by all variants in this table.

variables, as discussed further later. In Table 5.16, we see that the core sizes and number of steps decrease for each variant, combined with an increase in step size; despite these seemingly positive changes, performance decreases.

WCE combines well with the slice-based nature of this variant, as discussed in section 5.7, keeping search space small and overhead low. Stratification improves on key metrics, while decreasing overall performance; this is explainable when considering that the solvers intersect in Figure 5.1. In many cases, stratification aids performance, hence the improved metrics; however, in certain problems, stratification introduces low-weighing variables in early strata - as reformulation variables have weight 1 - which can cause a delay by forcing the solver to extract many (unit) cores in a relaxed setting, while the addition of later strata could help discover this information more easily. This effect is slightly amplified by hardening, as pruned intermediate solutions can cause more such variables to be introduced, in the few cases where these occur. The effect of partitioning is comparable to that in section 5.7; it results in smaller cores, but suffers from the large search spaces and partitions with little or no cores, as well as from the low-weighing cores mentioned before.

5.9 Variable-based weight splitting variants

Solver	vs	vs-w	vs-wh	vs-s	vs-sh	vs-p
# Solved	357	360	362	**363**	362	346
# Timed out	385	384	383	**380**	383	398
# Out of RAM	29	24	24	25	**23**	25
# Proven UNSAT	11	**14**	13	**14**	**14**	13

Table 5.17: A comparison of the variable-based weight splitting variants, in terms of reasons for termination. The best value in each column is made bold. Note that in the first and fourth columns, this is the largest value, while in the second and third columns, it is the smallest.

Solver	vs	vs-w	vs-wh	vs-s	vs-sh	vs-p
# of LB steps	99.4	89.7	**84.1**	93.5	92.3	88.1
LB step size	11.0	13.2	13.5	16.2	**16.5**	16.3
Core size	4.89	4.93	4.89	4.60	4.63	**4.46**
Total time (s)	24.5	**19.9**	20.0	24.0	24.2	26.9
Solver time (s)	24.1	**19.5**	19.6	23.7	23.8	25.1
# of nodes	77k	77k	77k	106k	81k	106k

Table 5.18: The data points describing the optimisation process for the variable-based weight splitting variants. For each metric, except for the number of nodes, the best value has been made bold; this may be the largest or smallest value, depending on the metric. The number of nodes has been excluded, as changes to it are not necessarily improvements, nor deteriorations. Data from 328 instances was used; this is the number of instances solved to optimality by all variants in this table.

From Table 5.17, we see that most features have a positive impact on the variant - only partitioning underperforms. All variants increase the number of instances proven unsatisfiable. Unhardened stratification has the largest positive effect. Similar to the slice-based variant, unsatisfiability is more easily proven due to a relaxation being considered.

According to Table 5.18, stratification combines particularly well with this variant due to the increased lower bound steps. It has a moderate effect, allowing it to be slightly more efficient on certain problems, leading to an overall performance increase[8]. Hardening slightly amplifies the positive effects, as well as decreasing the search space; however, this again at the cost of pruning suboptimal solutions[9], thereby experiencing a minor increase in time and an additional timeout. Surprisingly, the large step size in stratification does

[8]There is no particular reason why this variant experiences a performance increase while the slice-based variant does not; due to variable-based variants being slightly more efficient, a small number of problems was close to being solved, and the minor improvement incurred by stratification allowed these to be solved; in the slice-based case, the improvement is comparable, but such files are not present, and a significant improvement is needed to experience performance increase.

[9]In the few cases that used them.

not result in the lowest number of steps; WCE is able to decrease this metric further. The reason is that stratification has more variance: some problems, for example those with low weights, experience little benefit, while those with weights in different orders of magnitude experience an enormous benefit; WCE causes a more moderate yet structural decrease in step size. Overall, WCE is able to curate a significant speedup in many cases due to the disjoint cores; however, the search space expands relatively more than in the slice-based variants, somewhat diminishing the positive effect. Hardening allows slightly more cores to be extracted per reformulation, which slightly increases time due to additional (relaxed) search in some cases, but decreases overhead and time in other cases. Partitioning is able to find the smallest cores, while also having a low number of large steps; yet it underperforms because of the aforementioned partitions with little or no cores. In many cases, a lot of time is required to solve individual partitions, before a single core is even present in such a partition.

5.10 Variable-based coefficient eliminating variants

Solver	ve	ve-w	ve-wh	ve-s	ve-sh	ve-p
# Solved	**366**	360	357	364	363	350
# Timed out	**378**	381	384	383	384	396
# Out of RAM	27	27	27	**24**	**24**	**24**
# Proven UNSAT	11	**14**	**14**	11	11	12

Table 5.19: A comparison of the variable-based coefficient eliminating variants, in terms of reasons for termination. The best value in each column is made bold. Note that in the first and fourth columns, this is the largest value, while in the second and third columns, it is the smallest.

Solver	ve	ve-w	ve-wh	ve-s	ve-sh	ve-p
# of LB steps	189.0	175.6	167.4	147.8	148.1	**137.1**
LB step size	1.20	2.67	4.06	12.6	**13.0**	12.1
Core size	3.32	3.41	3.34	3.30	3.16	**2.95**
Total time (s)	31.2	26.1	**24.4**	27.2	27.4	29.3
Solver time (s)	30.8	25.7	**23.9**	26.8	26.9	27.6
# of nodes	93k	161k	157k	126k	106k	112k

Table 5.20: The data points describing the optimisation process for the variable-based coefficient eliminating variants. For each metric, except for the number of nodes, the best value has been made bold; this may be the largest or smallest value, depending on the metric. The number of nodes has been excluded, as changes to it are not necessarily improvements, nor deteriorations. Data from 334 instances was used; this is the number of instances solved to optimality by all variants in this table.

The numbers in Table 5.19 show that all features decrease the performance of the variable-based coefficient eliminating variant. The variants using WCE do increase the number of problems proven unsatisfiable, again because of the relaxation being considered. The number of instances solved to optimality decreases notably in these variants; more so than in the stratified variants. Partitioning again causes the most significant performance decrease.

Interestingly, Table 5.20 shows that several of the features do combine well with this variant; not only the core size or lower bound steps show improvements, but so does the time taken. WCE is the fastest by comparison, especially when combined with hardening; this is because, in many cases, the overhead significantly decreases. The cost, however, is that the search space expands very rapidly, much more so than in any other variant - due to the lack of both remainders and residual weights. In cases where this happens, the result is often a timeout; this causes the time taken according to Table 5.20 to remain relatively low, as it only incorporates the solved instances, and thus over-represents those with a positive effect. This is in line with the intersection visible in Figure 5.1. Stratification causes a minor speedup due to the large steps, but again introduces reformulation variables with low weight early in the process; this keeps the effects moderate, while the relaxations still heavily expand the search space, thereby increasing search time. Hardening slightly amplify most positive effects, while negatively affecting the small number of instances dependent on pruned intermediate solutions. Partitioning again suffers from the expanded search space as well as partitions containing little or no cores, despite finding by far the smallest cores and requiring the smallest number of steps.

5.11 Input files with a single objective term

As previously mentioned, several input problems have only a single term in their objective function (after flattening). When this is the case, core-guided search is performed on only a single variable, severely limiting its functionalities. As such, we shall briefly consider these cases separately.

From Table 5.21(a), we first observe that the performance of all core-guided variants is (nearly) equal. This is because, with a single objective term, only unit cores can be found; these are processed in the same way for each approach[10]. The small differences are most likely due to variance introduced by the scheduler program, or the underlying hardware. The linear search solver, on the other hand, performs much better on these single-term instances. The reason is that the strength of core-guided solvers lies in the extraction of multi-valued cores, and the resulting refinements. In instances with a single objective term, the core-guided variants perform constrained linear search; this naturally results in lower performance.

When instead looking at Table 5.21(b), it is clear that core-guided solver variants perform much more competitively when multi-valued cores are present; most outperform the

[10]This statement is only true when considering cases where *only* unit cores were found; if a unit core is found later in the search process, slice-based weight splitting resets its weight, something other variants do not do.

Solver	Correct outcome	Unsolved
ss	107	99
ss-w	107	99
ss-wh	107	99
ss-s	107	99
ss-sh	108	98
ss-p	109	97
se	108	98
se-w	106	100
se-wh	108	98
se-s	108	98
se-sh	108	98
se-p	108	98
vs	108	98
vs-w	107	99
vs-wh	108	98
vs-s	108	98
vs-sh	108	98
vs-p	108	98
ve	107	99
ve-w	108	98
ve-wh	107	99
ve-s	108	98
ve-sh	107	99
ve-p	108	98
lin	**123**	**83**

(a) For objective functions with a single term.

Solver	Correct outcome	Unsolved
ss	251	325
ss-w	257	319
ss-wh	262	314
ss-s	254	322
ss-sh	255	321
ss-p	242	334
se	248	328
se-w	256	320
se-wh	261	315
se-s	248	328
se-sh	246	330
se-p	239	337
vs	260	316
vs-w	267	309
vs-wh	267	309
vs-s	269	307
vs-sh	268	308
vs-p	251	325
ve	**270**	**306**
ve-w	266	310
ve-wh	264	312
ve-s	267	309
ve-sh	267	309
ve-p	254	322
lin	251	325

(b) For weighted linear objective functions.

Table 5.21: Tables showing the (simplified) termination conditions for input files with objective functions consisting of (a) a single term, and (b) a weighted linear sum of multiple variables. "Solved" indicates that either optimality or unsatisfiability was proven, "unsolved" indicates that neither was proven within the allotted time or memory bounds. Note that the linear search solver performs especially well on the input files whose objective function has a single term, while conceding to many core-guided variants on those using a weighted linear sum. There are 206 input files in the single term category, and 576 files in the weighted linear sum category.

linear search solver, with the other variants performing only slightly worse. This is in line with our expectations. To visually compare the performance on this type of problems, Figure 5.3 shows again the performance of all solvers, using only data from instances with multi-valued objective functions. It is clear that the linear search solver appears much less

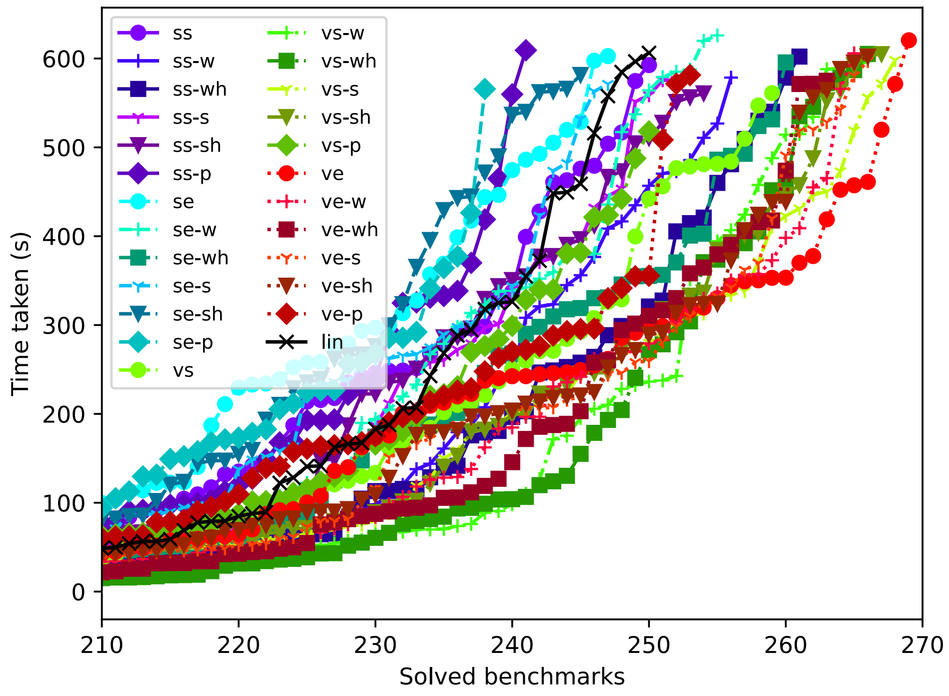


Figure 5.3: A zoomed cactus plot, showing the time taken (Y-axis) as a function of the number of instances solved before this time (X-axis), considering only the problems with a weighted linear sum as the objective function. Compared to Figure 5.1, more core-guided solvers outperform the linear search solver.

powerful than it did in Figure 5.1.

Based on this data, we can thus conclude that core-guided search requires a somewhat larger number of objective variables to perform optimally. It still performs relatively well on problems with only a single objective term, but is easily surpassed by linear search solvers in these cases. This also advocates the earlier conclusion that different types of solvers are most adequate for different types of problems, as all approaches have their individual strengths and weaknesses.

5.12 Influence of input problem metrics

In order to analyse which types of problems are most suited for core-guided solvers, several features were extracted for each input file. These features have been visualised in Figure A.1. These images have allowed us to draw some minor conclusions on the performance of each variant, for problems of different sizes.

The main conclusion that can be drawn from this data is that, on average, all solvers tend to perform better on smaller instances than on larger instances; the means and distributions of unsolved instances are higher than those of the solved instances. Additional conclusions can be drawn by looking at each plot individually.

When looking at the number of objective variables, we see that the mean of the linear search solver is lower than the other means. This is mostly due to the aforementioned fact that linear search performs significantly better on single-term objective functions, while core-guided search performs better when more terms are present. We also see minor decreases in the means for variants applying partitioning, and the variable-based solvers applying hardened WCE. Partitioning performs better on smaller objective functions as a smaller number of objective variables correlates with a smaller number of partitions to consider, resulting (on average) in a faster process. The problems solved by variable-based variants with hardened WCE tend to have fewer objective variables, as these benefit the most from disjoint cores. With relatively little variables, it is more likely to find non-minimal cores with reformulation variables; as such, overhead is reduced when cores are forced to be disjoint.

For the number of unique objective weights, we again observe that linear search is slightly biased to input files with a single term, and as such has a lower average. We see that stratified variants - especially when combined with hardening - solve problems with, on average, a smaller number of unique weights than other variants; this is mostly because the unique weights all need their own stratum, resulting in many strata and thus additional overhead when many unique weights are present. This effect is most present in the coefficient eliminating variants due to the reformulation variables having a weight of 1. It stands out that the average of variants with hardened WCE lies just above the overall average, in all cases except variable-based weight splitting. This specific variants excels when many variables have the same weight, due to the weight splitting nature and the absence of remainders. Hardened amplifies this effect by solving slightly more (difficult) problems with a low number of unique weights.

The number of variables shows that stratified approaches perform slightly better on larger problems; this is most likely due to the increased lower bound steps. Compared to the number of decisions on non-objective variables, the increased number of nodes incurred by stratification is insignificant, while the lower number of steps effectively decreases the number of cores that need to be extracted. Partitioning has a lower average, which can be explained by looking at graph size; the more variables are present, the larger the graph, most likely resulting in a large number of partitions. The linear search solver presumably performs worse on large instances due to the search space being unconstrained, but can solve additional small instances for the very same reason. The number of constraints shows approximately the same results.

Chapter 6

Conclusions

In this thesis, we have investigated the efficacy of several additional features for core-guided search in the context of CP. We have done so by first implementing the four reformulation and weight handling approaches described in [18], after which four additional features were implemented: WCE, stratification, hardening and partitioning. Of these, the first three have been previously implemented in [18], while partitioning has been newly adapted to CP. A total of twenty-four solver variants was considered; for each of the base variants, i.e. variants applying a given reformulation and weight handling approach, we considered a variant with no additional features, a variant applying partitioning, and variants applying either WCE or stratification, both with and without hardening. Note that several additional combinations could have been considered, such as partitioning with hardening, or a combination of WCE and stratification; these have been left out in favour of extensive individual evaluations for each feature and approach. Each of these twenty-four solver variants, in addition to a single linear search solver, were evaluated on a dataset of 782 problem instances.

The most important conclusion drawn from this research is the fact that solvers often solve different problems from one another; we have seen on multiple occasions that two solvers performed nearly identical in quantitative terms, while the selections of solved problems showed notable differences between the two. This was most clear when comparing the core-guided solver variants to the linear search solver as done in section 5.11, but also occurred for various combinations of core-guided solver variants. Despite the overall performance of most solvers being comparable - and the solvers completing largely the same set of instances - this shows that a lot can be gained by picking the right solver for a given problem. Additionally, we have seen that the order of assumptions can play a notable role in the performance of a core-guided solver.

For each individual solver variant, we analysed the performance in comparison to those applying the same feature, and to those using the same reformulation and weight handling approach. These conclusions were different for each variant, but for satisfiable instances, we mainly saw that variable-based coefficient elimination is most efficient when no additional features are used; that WCE improves the other variants, with slice-based approaches experiencing significant additional benefits from hardening; and that partitioning decreases overall performance in most cases, despite increasing several metrics. For unsatisfiable instances, we concluded that WCE quite strongly improved performance in all four variants;

that stratification allowed unsatisfiability to be proven in some instances, while causing additional timeouts in the coefficient eliminating instances; and that partitioning was able to improve performance to a lesser extent.

Chapter 7

Future research directions

Based on this research, several future research directions can be set out. This research has shown that, depending on the solver, certain additional features can significantly improve performance. This raises the question which other features can improve performance in a similar fashion. This may either be additional features already present in MaxSAT solvers, which are converted to work on CP, as has been done in this thesis; or research could be done to develop additional features specifically for CP-based core-guided solver, potentially even focusing on a specific reformulation or weight handling approach. Though this is the more promising option, it may prove more difficult, and may cause some promising features from MaxSAT to be overlooked.

A second research direction, related to the first one, is to investigate how much value can be found in altering the order in which assumptions are added. Though less prevalent in variable-based approaches, this has caused the slice-based weight splitting variant to outperform the slice-based coefficient eliminating variant on multiple occasions. Additionally, certain variants which are expected to perform comparably have shown divergence due to this factor. It may very well be the case that a proper heuristic for assumption ordering is able to increase performance by a notable amount, in both reformulation approaches, as well as across different features.

An alternative research direction could be the development of a more suitable partitioning approach. In this research, the implementation of partitioning was based on the MaxSAT version presented in [31]. The weighing scheme was made to approximately correspond to this MaxSAT version as well, and as such did not consider the unique features of CP. If a more appropriate graph representation and corresponding weighing scheme are designed specifically for CP, the partitions found in this graph may be much more adequate, potentially increasing overall performance significantly.

Bibliography

[1] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Solving (weighted) partial maxsat through satisfiability testing. In *International conference on theory and applications of satisfiability testing*, pages 427–440. Springer, 2009.

[2] Carlos Ansótegui, Maria Luisa Bonet, Joel Gabas, and Jordi Levy. Improving sat-based weighted maxsat solvers. In *International conference on principles and practice of constraint programming*, pages 86–101. Springer, 2012.

[3] Carlos Ansótegui, Maria Luisa Bonet, Joel Gabas, and Jordi Levy. Improving wpm2 for (weighted) partial maxsat. In *International Conference on Principles and Practice of Constraint Programming*, pages 117–132. Springer, 2013.

[4] Florent Avellaneda. A short description of the solver evalmaxsat. *MaxSAT Evaluation 2020*, page 8, 2020.

[5] Fahiem Bacchus. Maxhs in the 2020 maxsat evaluation. *MaxSAT Evaluation 2020*, pages 19–20, 2020.

[6] Jeremias Berg and Matti Järvisalo. Weight-aware core extraction in sat-based maxsat solving. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming*, pages 652–670, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66158-2.

[7] Jeremias Berg, Emir Demirović, and Peter J Stuckey. Core-boosted linear search for incomplete maxsat. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4–7, 2019, Proceedings 16*, pages 39–56. Springer, 2019.

[8] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[9] ConSol-Lab. Github - consol-lab/gourd: a command-line tool for configuring, running, and analysing algorithm comparison experiments on supercomputers. `https://github.com/ConSol-Lab/gourd`, 2024. Accessed: 2024-06-11.

[10] ConSol-Lab. Github - cesar48/pumpkin-cgs: A core-guided lazy clause generation constraint solver written in rust. `https://github.com/Cesar48/Pumpkin-CGS/tree/core-guided-v1.0.1`, 2024. Accessed: 2024-11-20.

[11] ConSol-Lab. Github - consol-lab/pumpkin: A lazy clause generation constraint solver written in rust. `https://github.com/ConSol-Lab/Pumpkin`, 2024. Accessed: 2024-06-11.

[12] Jessica Davies. *Solving MaxSAT by decoupling optimization and satisfaction*. PhD thesis, University of Toronto, 2013.

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

[14] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[15] Thibaut Feydy and Peter J Stuckey. Lazy clause generation reengineered. In *International Conference on Principles and Practice of Constraint Programming*, pages 352–366. Springer, 2009.

[16] Maarten Flippo, Konstantin Sidorov, Imko Marijnissen, Jeff Smits, and Emir Demirović. A multi-stage proof logging framework to certify the correctness of cp solvers. In *30th International Conference on Principles and Practice of Constraint Programming*, page 11. Schloss Dagstuhl-Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, 2024.

[17] Zhaohui Fu and Sharad Malik. On solving the partial max-sat problem. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 252–265, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37207-3.

[18] Graeme Gange, Jeremias Berg, Emir Demirović, and Peter J. Stuckey. Core-guided and core-boosted search for cp. In Emmanuel Hebrard and Nysret Musliu, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 205–221, Cham, 2020. Springer International Publishing. ISBN 978-3-030-58942-4.

[19] GraphText. Github - graphext/louvain-rs: Louvain clustering algorithm implementation in rust. `https://github.com/graphext/louvain-rs`, 2020. Accessed: 2024-12-03.

[20] Andy M Ham and Eray Cakici. Flexible job shop scheduling problem with parallel batch processing machines: Mip and cp approaches. *Computers & Industrial Engineering*, 102:160–165, 2016.

[21] Pierre Hansen and Brigitte Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44(4):279–303, 1990.

[22] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2009. ISBN 026251348X.

[23] Federico Heras, Javier Larrosa, and Albert Oliveras. Minimaxsat: A new weighted max-sat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 41–55. Springer, 2007.

[24] Alexey Ignatiev, António Morgado, and João Marques-Silva. Rc2: an efficient maxsat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, 2019.

[25] Elena Kelareva, Kevin Tierney, and Philip Kilby. Cp methods for scheduling and routing with time-dependent task costs. *EURO Journal on Computational Optimization*, 2(3):147–194, 2014.

[26] Joao Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications. In *2010 40th IEEE International Symposium on Multiple-Valued Logic*, pages 9–14. IEEE, 2010.

[27] Ruben Martins, Miguel Terra-Neves, Saurabh Joshi, Mikoláš Janota, Vasco Manquinho, and Inês Lynce. Open-wbo in maxsat evaluation 2017. *MaxSAT Evaluation 2017*, page 17, 2017.

[28] MiniZinc. Github - minizinc/minizinc-benchmarks: A suite of minizinc benchmarks. `https://github.com/MiniZinc/minizinc-benchmarks`, 2019. Accessed: 2024-07-04.

[29] Shuichi Miyazaki. Database queries as combinatorial optimization problems. *Proc. CODAS'96*, 1996.

[30] António Morgado, Carmine Dodaro, and Joao Marques-Silva. Core-guided maxsat with soft cardinality constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 564–573. Springer, 2014.

[31] Miguel Neves, Ruben Martins, Mikoláš Janota, Inês Lynce, and Vasco Manquinho. Exploiting resolution-based representations for maxsat solving. In *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18*, pages 272–286. Springer, 2015.

[32] Olga Ohrimenko, Peter James Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14:357–391, 2009. URL `https://api.semanticscholar.org/CorpusID:11356404`.

[33] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*, chapter 6, pages 169–208. Elsevier, 2006.

[34] SAT Association. Maxsat evaluation 2023. `https://maxsat-evaluations.github.io/2023/index.html`, 2023. Accessed: 2024-04-16.

[35] Willem-Jan Van Hoeve. The alldifferent constraint: A survey. *arXiv preprint cs/0105015*, 2001.

[36] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. Satire: A new incremental satisfiability engine. In *Proceedings of the 38th annual Design Automation Conference*, pages 542–545, 2001.

Appendix A

Results table and input file data box plots

Solver	# Solved	# Timed out	# Out of RAM	# Proven UN-SAT	# of LB steps	LB step size	LB step slope	Core size	Core size slope	Total time (s)	Solver time (s)	Special time (s)	# of WCE cores	Soft frac-tion	# of nodes
ss	347	400	24	11	113.8	10.8	-0.44	8.70	-0.61	39.1	38.1	-	-	-	105k
ss-w	350	394	24	14	79.7	11.4	-0.46	8.07	-0.47	32.1	31.0	0.00	29.4	-	98k
ss-wh	356	389	24	14	84.8	11.3	-0.45	8.10	-0.71	34.7	35.3	0.00	32.0	0.72	85k
ss-s	347	397	24	14	87.2	16.9	-17.6	8.24	-0.24	39.6	38.4	0.00	-	-	177k
ss-sh	349	396	23	14	101.3	17.1	-17.2	7.42	-0.16	43.9	42.7	0.00	-	0.78	159k
ss-p	338	407	24	13	86.4	16.3	-16.5	6.81	-0.26	35.7	33.9	0.53	-	-	141k
se	345	399	27	11	205.7	1.14	-0.14	6.25	-0.68	47.0	45.5	-	-	-	93k
se-w	348	394	26	14	170.6	1.53	-0.15	5.91	-0.55	41.3	40.0	0.00	54.6	-	113k
se-wh	355	387	26	14	183.7	1.86	-0.26	5.70	-0.74	41.4	40.2	0.00	58.5	0.72	87k
se-s	345	402	24	11	145.9	12.4	-17.1	6.17	-0.38	41.8	40.5	0.00	-	-	167k
se-sh	343	404	24	11	144.2	12.8	-17.0	5.79	-0.42	46.1	44.8	0.00	-	0.78	162k
se-p	335	411	24	12	157.0	12.2	-16.9	5.28	-0.16	35.5	33.6	0.53	-	-	114k
vs	357	385	29	11	113.8	10.5	-0.43	5.83	-0.70	40.3	39.3	-	-	-	100k
vs-w	360	384	24	14	103.1	13.3	-3.67	5.66	-0.78	35.1	34.3	0.00	5.06	-	129k
vs-wh	362	383	24	13	116.9	13.3	-3.65	5.50	-0.72	37.0	36.1	0.00	4.97	0.64	150k
vs-s	363	380	25	14	108.3	16.5	-16.8	5.49	-0.32	43.6	42.6	0.00	-	-	172k
vs-sh	362	383	23	14	108.7	16.4	-16.5	4.88	-0.55	44.9	44.1	0.00	-	0.76	155k
vs-p	346	398	25	13	92.2	16.5	-15.4	4.68	-0.33	35.8	34.1	0.53	-	-	165k
ve	366	378	27	11	210.5	1.19	-0.13	3.52	-0.88	48.0	46.9	-	-	-	122k
ve-w	360	381	27	14	200.2	2.54	-0.18	3.45	-0.73	39.1	38.2	0.00	4.14	-	213k
ve-wh	357	384	27	14	189.5	4.19	-0.40	3.40	-0.77	35.0	34.0	0.00	4.05	0.64	203k
ve-s	364	383	24	11	161.0	11.8	-16.1	3.51	-0.63	46.1	45.1	0.00	-	-	193k
ve-sh	363	384	24	11	162.4	12.3	-15.9	3.37	-0.61	46.2	45.2	0.00	-	0.76	177k
ve-p	350	396	24	12	156.8	11.9	-16.1	2.92	-0.40	39.3	37.7	0.53	-	-	129k
lin	359	385	23	15	-	-	-	-	-	41.8	-	-	-	-	196k

Table A.1: An overview of all collected data points for all core-guided solver variants, and some relevant data points for the linear search solver used as baseline. The core-guided solvers are abbreviated based on their reformulation (first letter; “s” for slice-based and “v” for variable-based) and weight handling (second letter; “s” for (weight) splitting and “e” for (coefficient) elimination) approaches, as well as their additional features (letters after the dash; “w” for WCE, “s” for stratification, “p” for partitioning and “h” for hardening).

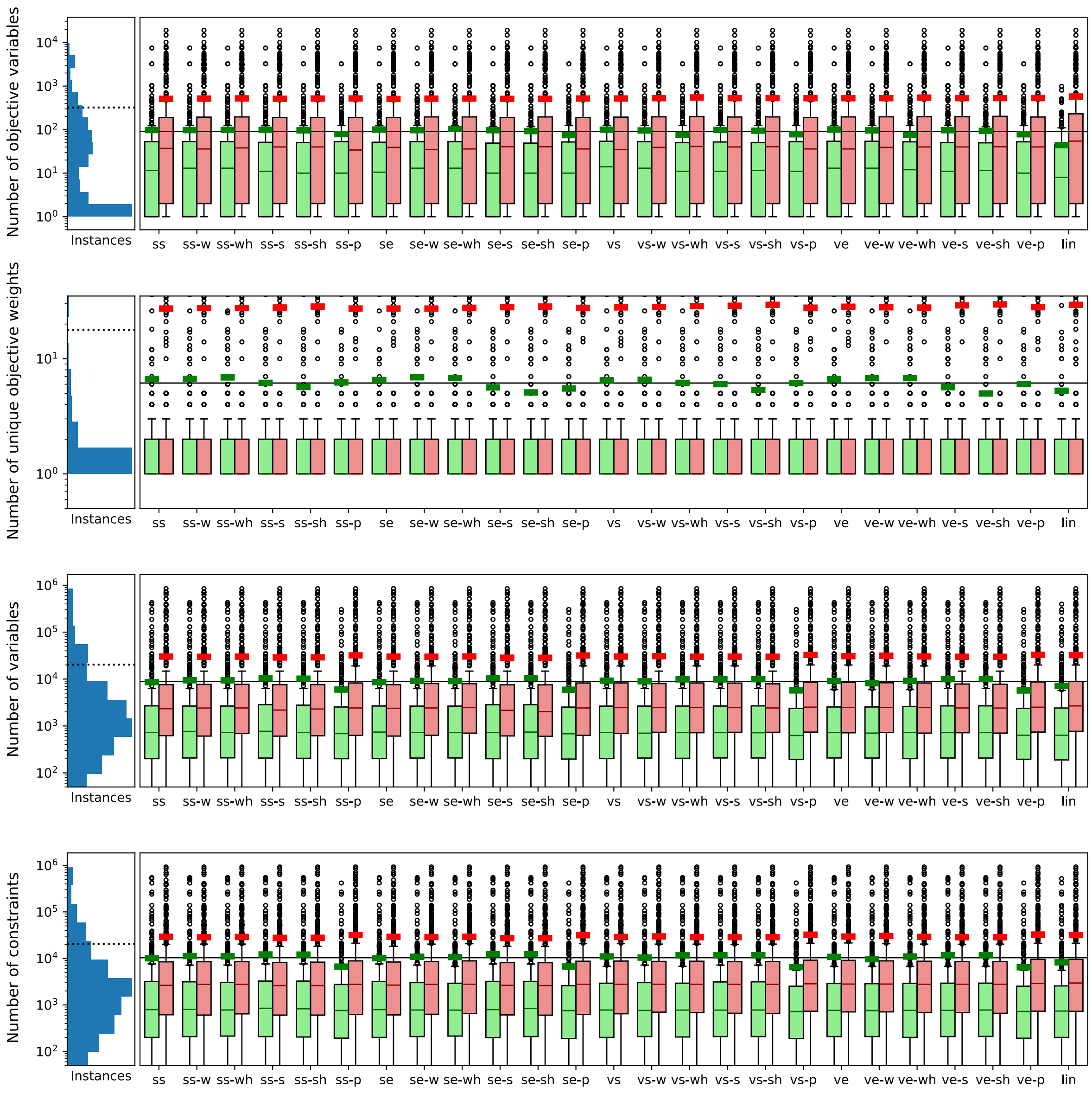


Figure A.1: Box plots showing the approximate distribution of solved (green) and unsolved (red) instances per solver, across four input file metrics. To the left of each plot is a histogram showing the distribution of the instances. The arithmetic mean of each box is represented by a thick bar, usually above the box itself. Note that the Y-axis is logarithmic, meaning the histogram has exponentially increasing bin sizes. A dotted line in the histogram shows the arithmetic mean value across all input files, and a thin solid line shows the arithmetic mean value across the solved input files. To highlight the most relevant data, three plots have been cropped; the plot of the unique objective weights lacks its top part, while the plots showing the number of variables and constraints lack their bottom parts. Note that a significant proportion of input files has only a single objective variable, and an even larger proportion has only a single unique weight.