Lazy Clause Generation for Bin Packing

Explaining Bin Packing Propagation with Boolean Variables

Melvin de Kloe

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

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

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

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

Abstract

In this paper we embed a solution for bin packing problems in a constraint programming environment. Existing solutions for bin packing problems are plentiful, but rigid. We have taken existing solutions of bin packing in constraint programming, and analysed the steps this algorithm takes. We then developed explanations for these steps in the form of boolean clauses. Using these explanations, a constraint programming paradigm by the name of lazy clause generation is able to generate new rules, that prevent the solver from making the same mistake twice. We have compared our implementation to two different solutions. Firstly we compared it to a decomposition, where the bin packing constraint was broken down into many smaller and simpler constraints. Secondly, we compared our implementation to a version with naive explanations. In one benchmark, comparing our version to the decomposition in a pure bin packing problem, our implementation vastly outperformed the decomposition. In other benchmarks, the results comparing our version to the other two were much more inconclusive. While showing marginal improvements in some cases, our version performed worse in other cases. The research performed in this paper definitely shows promise, and there is merit in exploring this approach in further research.

1 Introduction

The bin packing problem is a challenge that describes a wide variety of real world problems, from storing items in an actual bin, to packing items into containers for transportation. This set of problems has been studied since the 1970s [1], but also has recent applications in fields like cloud computing [2]. This paper focuses on the one-dimensional version of the bin packing problem. That is, putting items with a given size into bins with a given capacity, as shown in the example in Figure 1.

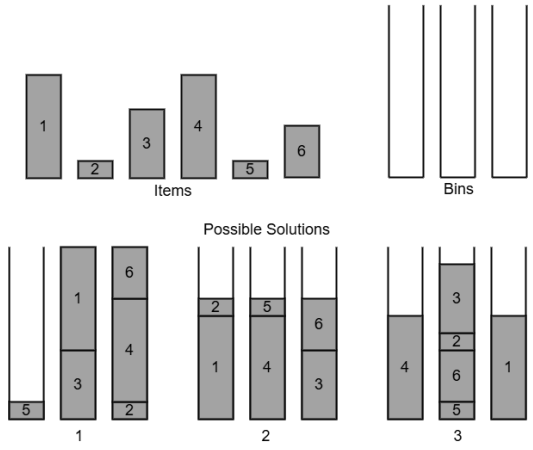

Figure 1: An instance of the bin packing problem with three possible solutions

While algorithms have long existed that solve this NP Complete problem and variations thereof (e.g. [3; 4; 5]), this paper focuses on creating a solution using Constraint Programming (CP), and more specifically, Lazy Clause Generation (LCG). The reason for this adaptation is that the bin packing problem is often part of a bigger problem, for example including allocation of other resources, or needing the items to be sorted in some way. This means that the previously mentioned algorithms can quickly become unviable, while CP allows for flexibility. CP uses a combination of domains and constraints to solve problems. CP can be used to simply solve the bin packing problem, but allows for flexibility with the ability to add other constraints to the problem, such as item ordering or resource management. Algorithms for bin packing in CP already exist, but not specifically for LCG, which shows promise over other branches of CP [6].

LCG is a CP paradigm that uses boolean clauses to explain the steps taken by the solver. When a conflict is encountered, these explanations are used to lazily construct a new clause called a nogood, which is used to avoid making the same mistake in the future, reducing the search space of the problem. The difficulty of implementing a new constraint in LCG comes from generating these explanations.

In this research paper, we developed a constraint for bin packing in LCG, using boolean statements to explain every decision the algorithms makes. By running the developed implementation on benchmarks simulating real life CP problems, we found that, in problems with only a bin packing constraint, our algorithm is superior to a decomposition of the problem into smaller constraints. However, more nuanced results are achieved in problems that also use other constraints, leading to mixed success in reducing the runtime, but showing great promise in effective clause generation. We also compared our solution to a naive implementation of the explanations. That is, describing the domains of every variable for each step taken as an explanation. While the data was mostly inconclusive about the effect of our explanations on the search steps taken, there was once again a clear improvement in the quality of the clauses generated by our method.

Firstly in Section 2, we cover information about CP, LCG and explanations to build understanding for the rest of the paper. Secondly, Section 3 goes more in depth into the bin packing constraint and its existing implementations. Then Section 4 lists existing work related to the problem we are solving. Section 5 shows our contribution to the field, listing the explanations we developed for each constraint. The results of our work are shown in Section 6, with data comparing our implementation to a decomposition and naive explanations on several benchmarks. Section 7 concludes by reiterating the most important aspects of the paper and proposing possible directions for future work. Finally Section 8 addresses the ethical aspects of the research done and the reproducibility of the methods used.

2 Preliminaries

This section gives a basic description of Constraint Programming and Lazy Clause Generation. In writing this Section we have taken strong inspirations from the Basic Principles chap-

ter in [7], and we refer you to that thesis for a more in-depth description of these subjects.

2.1 Constraint Programming

Constraint Programming (CP) [8] is a programming paradigm used for solving combinatorial search problems. CP takes approaches from artificial intelligence, computer science, databases, programming languages, and operations research, with applications in fields such as scheduling, planning, vehicle routing, configuration, networks, and bioinformatics, and more [8]. CP uses domains as possible values for variables, and constraints as rules that restrict those domains. Think of a Sudoku puzzle, where every empty square starts with the domain one through nine, and the constraint is: every row, column and box can only contain one of each number. Using these two facts, the solution can then be found by deduction, and trial and error.

Constraint Satisfaction Problem

CP problems are often stated as a Constraint Satisfaction Problem (CSP). A CSP is defined as $\mathcal{P} = (\mathcal{C}, \mathcal{X}, \mathcal{D})$ where $\mathcal{X}$ is the sequence of variables, $\mathcal{D}$ is their domains and $\mathcal{C}$ is the set of constraints that apply to the problem. To solve a CSP, every variable $x_i \in \mathcal{X}$ must be assigned a value, and every constraint $C(X) \in \mathcal{C}$ where $X \subseteq \mathcal{X}$ has to be satisfied by that assignment. The domain of a value $D(x_i)$ contains all the values that x_i could potentially take. For example the domain of a positive odd number would be $\mathcal{D}(\text{odd}) = \{1, 3, 5, ...\}$. The purpose of a constraint is to enforce rules on the variables, one such rule might for example be: x_1 has to be an odd number.

Some more notation that is used in the rest of this paper follows. We will denote the initial domain of x_i as $D_0(x_i)$, and the maximum and minimum element of $D(x_i)$ as $\overline{x_i}$ and $\underline{x_i}$ respectively. The assignment of a value v to a variable x is denoted as $x \leftarrow v$, this sets the domain $D(x)$ to $\{v\}$. Updating the maximum value of a domain $D(x)$ is shown as $\overline{x_i} \leftarrow v$. This removes all values larger than v from $D(x)$, if there are any. The same notation is used for updating the minimum: $\underline{x_i} \leftarrow v$. And finally, removing a value v from a domain is denoted as $x \nleftarrow v$

Solving

Solving a CSP is done by either finding a solution that satisfies all constraints, or by proving that the problem is infeasible. This is done by propagation and search steps. Propagation attempts to shrink the domain space, reducing the amount of possible assignments for each variable. The constraints are used to remove the values that can not lead to a solution from the domains. For example: take variables x_1 and x_2 with domains $D(x_1) = \{1, 2, 3\}$ and $D(x_2) = \{3, 4, 5\}$ and the constraint $x_1 + x_2 = 5$. In this case x_1 could never be 3 and x_2 could never be 5. The propagation step would then remove those values from the domains of x_1 and x_2. The propagation step continues until no such deductions can be made. If at any point during the propagation step a constraint leads to a domain becoming empty, we've encountered a conflict.

The search step happens after the propagation. This step assigns a value to a variable x_i from its domain, and attempts to solve the newly created CSP. If this assignment leads to a valid assignment of variables, a solution to the CSP has been found. In case of a conflict however, the search is *pruned*, i.e. the search stops at this point and backtracking is used to assign a different value to x_i instead, and once again attempts to solve the new CSP. This process continues until either a solution is found, or the search space has been exhausted, in which case the CSP is infeasible. In the previous example the domains after propagation would be $D(x_1) = \{1, 2\}$ and $D(x_2) = \{3, 4\}$. The search step might assign $x_1 = 1$, after which the propagation step happens again. This time the constraint can be used to determine $x_2 = 4$, after which the CSP is solved.

2.2 Lazy Clause Generation and Explanations

Lazy Clause Generation (LCG) is a combination of two constraint solving techniques: Boolean Satisfiability (SAT) and Finite Domain (FD). It was originally developed by O. Ohrimenko et al. in 2007 [9], and later improved upon to be able to compete with the best FD solutions of the time [6]. LCG combines the simplicity of an FD solver with the ability of SAT to reduce search space by recording conflicts as new boolean clauses to avoid running into the same mistake more than once. To achieve this LCG uses *explanations*: every time a propagation is made, or a conflict is encountered, it is recorded as a clause that explains what just happened using booleans.

LCG uses the boolean variables $[[x = a]], [[x = a + 1]], ..., [[x = b]]$ and $[[x \leq a]], [[x \leq a + 1]], ..., [[x \leq b - 1]]$ where $name$ is the name of the boolean variable $[[name]]$. Any domain $D(x) \subseteq D_0(x) = [a, b]$ can be described using these boolean variables. LCG uses these variables to create clauses in case of a conflict. Such a clause is called a nogood, and is generated from the constraint that caused the conflict. Nogoods are lazily added as a new pseudo-constraint to ensure this conflict doesn't happen again. We will sometimes write the following: $[[x \neq v]]$ instead of $\neg[[x = v]]$, $[[x > v]]$ instead of $\neg[[x \leq v]]$ and $[[x \geq v]]$ instead of $\neg[[x \leq v - 1]]$.

Take for example the domains:

- $D(a) = \{1, 2, 3\}$
- $D(b) = \{3, 4, 5\}$
- $D(c) = \{1, 2, 3, 4\}$

And the constraints:

- $a + b + c \leq 7$
- $a + b \geq 5$
- $b + c \geq 5$

The first search step arbitrarily takes $b = 3$. Propagation then uses the second constraint to reduce $D(a) = \{2, 3\}$ and $D(c) = \{2, 3, 4\}$. The second search step arbitrarily takes $a = 3$. This leads to a conflict with the first constraint: $D(c) = \{\}$. LCG uses the failing constraint to create a nogood that explains why this went wrong: $[[a = 3]] \wedge [[b = 3]] \wedge [[c \neq 1]] \rightarrow \bot$. However, since $[[c \neq 1]]$ is only true because $[[b = 3]]$ is true, we can replace that variable with $[[b = 3]]$, and in this case remove it because it is a duplicate. This leads to the reason for conflict: $[[a = 3]] \wedge [[b = 3]] \rightarrow \bot$. LCG now backtracks to before the assignment of a, and adds the lazily created clause $\neg[[a = 3]] \vee \neg[[b = 3]]$, to make sure this conflict does not happen again.

3 Bin Packing

The one-dimensional version of the bin packing problem that this paper focuses on is described as follows: Given n amount of indivisible items with non-negative integer sizes s_i, is it possible to pack them into a set of m bins, each with capacity C? A possible instance of such a problem including three possible solutions is shown in Figure 2.

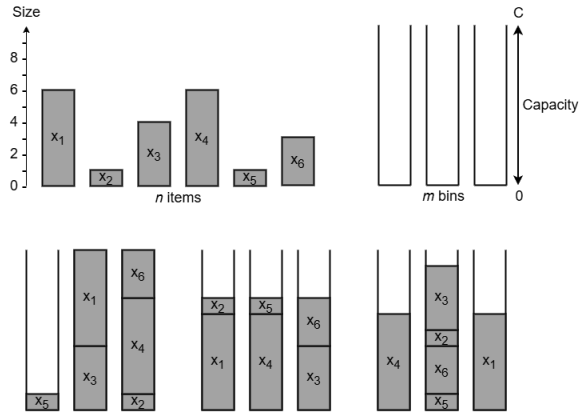

Figure 2: An instance of the bin packing problem, where $n = 6$ items need to be packed in $m = 4$ bins of capacity $C = 10$ with three possible solutions

3.1 Notation

The bin packing constraint takes three vector parameters containing the bin *loads* l, the *bin assignment* for each item b and the *sizes* of each item s. The load vector l contains m constrained variables l_i, one for each bin, usually[1] ranging from 0 to the maximum capacity C: $l_i \in [0, C]$. The *bin assignment* vector b contains n constrained variables b_i, one for each item, indicating in which bin the item is stored by taking their index as a value: $b_i \in [1, m]$. The *size* vector s contains n integers s_i, one for each item, for which $s_i \in \mathbb{N}$. For simplicity in propagations that depend on ordering, we assume for the rest of this paper that the sizes are in non-increasing order, i.e. $s_i \geq s_{i+1}$.

We use some notation that is not included in the constraint itself, but are still important to the propagations. Every item has its own index, denoted by the set $I = \{1...n\}$. The bins have a similar set for indices $B = \{1...m\}$. The total sum of all items is represented as $S = \sum_{i=1}^{n} s_i$. P. Shaw introduces some more notation to describe partial assignments [10] which we will use in describing propagations. The set of items that are already packed in bin j are denoted as the *required set* $R_j = \{i \mid i \in I \wedge D(b_i) = \{j\}\}$, i.e. the set of items that have only j in their bin assignment domain. The set of items that are or could potentially be packed in bin j is defined as the *possible set* $P_j = \{i \mid i \in I \wedge j \in D(b_i)\}$, i.e. the set of items that contain j in their bin assignment domain. The *candidate set* is then defined as the set of items that could still be packed in bin j, but have not yet been assigned to it: $C_j = P_j - R_j$.

[1]These values might differ, for example in problems where bins have individual capacities instead of identical ones.

The set of *unpacked* items contains all items that have not been assigned to any bin yet: $U = \{i \mid i \in I \wedge |D(b_i)| > 1\}$. The total size of the items packed in bin j is referred to as $p_j = \sum_{i \in R_j} s_i$.

3.2 Typical Constraints

To maintain consistency in the bin packing constraint, some constraints have to be set up. The following constraints are described by P. Shaw as the "basic" constraints of the typical bin packing model [10].

Pack All. Every item $i \in I$ has to be packed in a bin $j \in B$.

$$\underline{b_i} \leftarrow 1 \qquad\qquad \overline{b_i} \leftarrow m$$

Load Maintenance. Naive minimum and maximum load for each bin is maintained by using the required set and potential set to calculate them. For each bin $j \in B$:

$$\underline{l_j} \leftarrow \sum_{i \in R_j} s_i \qquad\qquad \overline{l_j} \leftarrow \sum_{i \in P_j} s_i$$

Load and Size Coherence. These propagations allow for more refinement of the upper and lower bounds. It uses the total sum of all item sizes, and the maxima or minima of the other bins to calculate the minimum or maximum of the current bin. for every bin $j \in B$:

$$\underline{l_j} \leftarrow S - \sum_{k \in B \setminus j} \overline{l_k} \qquad\qquad \overline{l_j} \leftarrow S - \sum_{k \in B \setminus j} \underline{l_k}$$

For example: if the sum of sizes is $S = 10$, and the sum of upper bounds of all other bins is 6, then the current bin has to have a lower bound of 4 to be able to pack all items.

Single Item Eliminations and Commitment. The upper and lower bounds on bin loads can be used to determine if an item can or cannot be packed into a bin. If adding an item to a bin would exceed the upper bound of that bin, it is eliminated:

$$\text{if} \quad \exists i (i \in C_j \wedge p_j + s_i > \overline{l_j}) \quad \text{then} \quad b_i \nleftarrow j$$

A similar case occurs with the lower bound, if packing all candidates C_j except for i into the bin does not exceed the lower bound, then i has to be added to the bin.

$$\text{if} \quad \exists i (i \in C_j \wedge \text{SUM}(P_j) - s_i < \underline{l_j}) \quad \text{then} \quad b_i \leftarrow j$$

3.3 Neighbouring Subsets

In addition to these basic constraints, P. Shaw introduces another concept that allows for more propagations: Neighbouring subsets [10]. The goal of using neighbouring subsets is to verify that not a single subset of candidates C_j falls within the load bounds $\underline{l_j}$ and $\overline{l_j}$ of bin j. If this is indeed the case, the search can be pruned since there is no valid solution that fills bin j correctly. To prove this fact we can use neighbouring subsets of the candidate set C_j. A pair of subsets are considered neighbouring subsets if there is no other subset of items whose sum of sizes is strictly between the other pair. By using neighbouring subsets, it is possible to prove infeasibility by finding two neighbouring subsets where one of them falls

below the lower bound, and the other above the upper bound of the bin load l_j. Figure 3 shows an example of this. It can be proven that the sets $\{x_1, x_4\}$ and $\{x_2, x_3\}$ are neighbours. Since the first set falls below the lower bound of the bin, and the second set falls above the upper bound, with the knowledge that these two sets are neighbouring, it is clearly visible that this problem is infeasible.

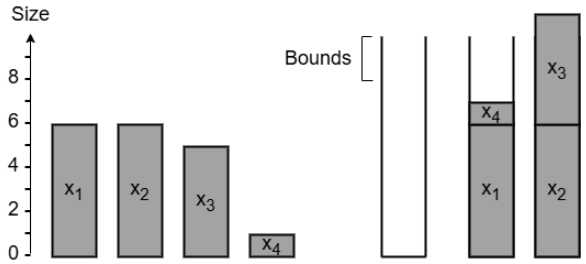

Figure 3: Example of an infeasible bin packing problem using neighbouring subsets $\{x_1, x_4\}$ and $\{x_2, x_3\}$.

To use neighbouring subsets for propagation, we first need to find neighbouring subsets. P. Shaw describes a way to generate neighbouring subsets of a particular structure using Figure 4 [10]. We start with a set of candidate items X. As stated before, we assume that the items are sorted in non-increasing item size. Three subsets of X are marked in the figure: A, B and C. Subset A consists of the k largest elements in X, while subset C consists of the k' smallest elements. Subset B consists of the $k+1$ smallest elements not already included in C. Overlap between A and B is allowed, but C is disjunct from the others. By then combining A and C into a *low-set* L_k, and taking B as the *high-set* H_k, the following holds: if $\sum_{i \in L_k} s_i \leq \sum_{i \in H_k} s_i$ then L_k and H_k are neighbouring. For the proof of this statement, see the original paper [10].

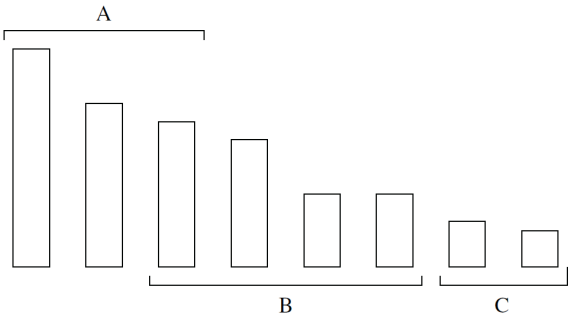

Figure 4: Structure of neighbouring subsets, taken from [10]

The procedure for finding neighbouring subsets is then used in an algorithm to detect if there is a pair of neighbouring subsets where L_k falls under the lower bound α, and H_k falls above the upper bound β. This procedure is shown in Algorithm 1 as presented by P. Shaw [10]. The NoSum algorithm returns true if it detects that the bin with lower bound α and upper bound β cannot be filled with the candidate items in set X. Using this algorithm, it is possible to construct more constraints.

Algorithm 1 NoSum(X, α, β) by P. Shaw [10]

if $\alpha \leq 0 \vee \beta \geq Sum(X)$ **then**
  return false
end if
$\sum_A, \sum_B, \sum_C := 0$
$k, k' := 0$
while $\sum_C + s_{|X|-k'} < \alpha$ **do**
  $\sum_C := \sum_C + s_{|X|-k'}$
  $k' := k' + 1$
end while
$\sum_B := s_{|X|-k'}$
while $\sum_A < \alpha \wedge \sum_B \leq \beta$ **do**
  $k := k + 1$
  $\sum_A := \sum_A + s_k$
  if $\sum_A < \alpha$ **then**
    $k' := k' - 1$
    $\sum_B := \sum_B + s_{|X|-k'}$
    $\sum_C := \sum_C - s_{|X|-k'}$
    while $\sum_A + \sum_C \geq \alpha$ **do**
      $k' := k' - 1$
      $\sum_C := \sum_C - s_{|X|-k'}$
      $\sum_B := \sum_B + s_{|X|-k'} - s_{|X|-k'-k-1}$
    end while
  end if
end while
return $\sum_A < \alpha$

NoSum Pruning. The simplest propagation we can make with NoSum is checking if the bin is packable. To do this, input the candidate set C_j and remove the size of the already packed items from the upper and lower bounds. Then for every bin j:

$$\text{if} \quad \text{NoSum}(C_j, \underline{l_j} - p_j, \overline{l_j} - p_j) \quad \text{then } \textbf{prune}$$

NoSum Load Bound Tightening. It is also possible to tighten the bounds using NoSum. By altering the algorithm to also return $\alpha' = \sum_A + \sum_C$ and $\beta' = \sum_B$, we can update the current bounds by using the following propagations. (Note the change in input variables.)

$$\text{if} \quad \text{NoSum}(C_j, \underline{l_j} - p_j, \underline{l_j} - p_j) \quad \text{then} \quad \underline{l_j} \leftarrow p_j + \beta'$$

$$\text{if} \quad \text{NoSum}(C_j, \overline{l_j} - p_j, \overline{l_j} - p_j) \quad \text{then} \quad \overline{l_j} \leftarrow p_j + \alpha'$$

NoSum Item Elimination and Commitment. Finally, by trying the NoSum algorithm in cases where an element from the candidate set C_j would be packed in the bin or removed from the set, we can use the pruning rule to check for feasibility in these cases. We check for example the case where, if we were to add i to the bin, that the problem would still be feasible. The same tests happen for the case where an item is removed from C_j.

$$\text{if} \quad \text{NoSum}(C_j \backslash i, \underline{l_j} - p_j - s_i, \overline{l_j} - p_j - s_i) \quad \text{then} \quad b_i \nleftarrow j$$

$$\text{if} \quad \text{NoSum}(C_j \backslash i, \underline{l_j} - p_j, \overline{l_j} - p_j) \quad \text{then} \quad b_i \leftarrow j$$

4 Related Work

All of the constraints described in Section 3 are taken from P. Shaw [10]. There is one more constraint in that paper that we were not able to include due to time restrictions. This constraint calculates, given the current partial solution, a lower bound on the amount of bins necessary to solve the packing. If this lower bound is larger than the amount of available bins, we have run into a conflict. While P. Shaw showed only minor improvements in most cases by including this, some cases benefitted greatly from including the lower bound constraint [10]. Besides P. Shaw's implementation of a lower bound constraint, several other lower bound calculations exist [11; 12], and a comparison between these in future research could provide valuable information.

This paper by P. Schaus [13] adds two constraints to the bin packing model and tests their constraints. A third constraint is also described but it requires a modified global cardinality constraint (gcc), which is out of the scope of this paper, and is also not tested to show an improvement over Shaw's model. Due to these factors we will not cover this constraint, but it might be interesting to explore in future work.

The first constraint relies on relaxing the bin packing problem such that items could be split amongst several bins. This relaxation is then used to test for inconsistencies using a maximum flow algorithm. While leading to stronger pruning, this method had too much overhead in time which caused poor performance.

The second constraint improves upon the calculation of the lower bound on the bin count described by P. Shaw. Specifically, it improves the adaptation used in partial solutions. While in most cases this lead to a negligible improvement in runtime, some cases experienced a large benefit from this improved algorithm, one such case going from a timeout of 300 seconds with Shaw's algorithm, to a successful run in 26 seconds. The time restriction of this paper did not allow us to include this improved algorithm for LCG solving, but we highly recommend future work be carried out to do so.

5 Explanations for Bin Packing

In this Section we describe the explanations we developed for every propagator described in Section 3. First, we derive a general explanation for the candidate set C_j, since the candidate set is used in many of the propagators. Then, using this explanation, we provide explanations for the typical constraints and the NoSum constraints.

5.1 Explaining the Candidate Set

The candidate set is used by several propagators, and is therefore part of the explanations of those propagations. For that reason we developed a generalized explanation that describes the elements of the candidate set. The difficulty in describing this set using the boolean variables used by LCG, comes from the lack of a *contains* variable. We can only use the variables $[[b_i = j]]$, $[[b_i \leq j]]$ and their negations $[[b_i \neq j]]$ and $[[b_i > j]]$, where the latter two are simply $\neg[[b_i = j]]$ and $\neg[[b_i \leq j]]$. Using only these variables, the candidate set can only be described by listing both the items whose domains does not contain bin j, as well as the items that are already assigned to bin j. That is, we are describing set C_j with its complement C_j^C relative to I: where $C_j^C = \{i \mid i \in I \wedge (j \notin D(b_i) \vee D(b_i) = \{j\})\}$. Candidate set C_j is then equal to $I \setminus C_j^C$. Every item i that is already packed into bin j then results in the variable $[[b_i = j]]$, and every item whose domain does not contain j results in $[[b_i \neq j]]$. The candidate set can then be described by a conjunction of all of these variables, and we will refer to this conjunction as $[[C_j]]$ in the future.

As an example of this explanation, Figure 5 shows an instance of a bin packing problem with a partial assignments of items. To describe the candidate set of the leftmost bin, we need to collect all of the items in its complement. Firstly, the set of items already packed in the bin is in this case $\{x_1\}$. Secondly, the items that do not have j in their domain is in this case $\{x_2, x_3, x_4\}$, since x_2 and x_3 are already packed in a different bin, and x_4 is too large to fit into the bin. The explanation describing this candidate set is then: $[[x_1 = 1]] \wedge [[x_2 \neq 1]] \wedge [[x_3 \neq 1]] \wedge [[x_4 \neq 1]]$.

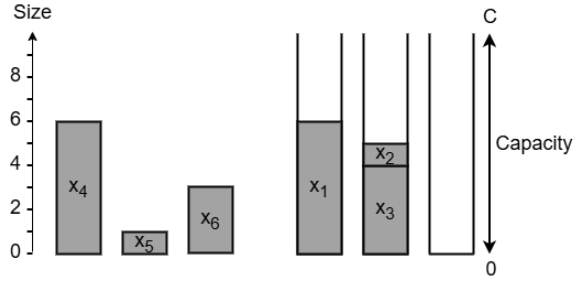

Figure 5: An instance of a bin packing problem with a partial assignment of items.

5.2 Explaining the Propagators

Here we show the explanations we developed for each propagator listed in Section 3, and give some intuition to our choice of these clauses.

Load Maintenance. For the lower bound on the load maintenance, only the items in the required set R_j are mapped to boolean variables for every bin $j \in B$:

$$\bigwedge_{i \in R_j} [[b_i = j]]$$

For the upper bound on the load maintenance, every item in the possible set P_j needs to be mapped. However, this is every item in either R_j or C_j. Since the explanation for C_j already includes all items in R_j, we do not need to map the set R_j again separately. The explanation for the upper bound on the load maintenance is then simply for every bin $j \in B$:

$$[[C_j]]$$

Load and Size Coherence. Both propagations here use the total size of all items, which remains constant throughout the problem, and therefore does not need an explanation. The lower bound propagation uses the upper bounds of all

other bins, where the upper bound propagation uses the lower bounds of all other bins. The explanations for the lower bound and upper bound coherence for every bin $j \in B$ are then respectively:

$$\bigwedge_{k \in B \setminus j} [[l_k \leq \overline{l_k}]] \qquad \bigwedge_{k \in B \setminus j} [[l_k \geq \underline{l_k}]]$$

Single Item Eliminations and Commitment. For single item elimination, although every item in the candidate set C_j is evaluated, we do not need to include $[[C_j]]$ in the explanation, since it is simply the set we iterate over, and not a part of the reasoning for the elimination. The only variables that need to be used in the explanation are the items in the required set R_j and the upper bound on the load of bin j $\overline{l_j}$. This explanation can be improved by increasing the value of $\overline{l_j}$ in the explanation if possible. For example: $p_j = 2;\ s_i = 4;\ \overline{l_j} = 4$. We can see that $p_j + s_i > \overline{l_j}$, leading to the explanation $[[l_j \leq 4]] \wedge \bigwedge_{i \in R_j} [[b_i = j]]$. However, we can increase the strength of the explanation by increasing the value of $\overline{l_j}$ to $p_j + s_i - 1 = 5$, since the item would have still been eliminated in the case where $\overline{l_j} = 5$. The explanation for single item eliminations for every item $i \in I$ eliminated from bin $j \in B$ then becomes:

$$[[l_j \leq p_j + s_i - 1]] \wedge \bigwedge_{i \in R_j} [[b_i = j]]$$

A similar case occurs with the lower bound, where we can decrease the value of the lower bound to $\text{SUM}(P_j) - s_i + 1$. Since this propagation uses the possible set P_j, as before we explain that set as $[[C_j]]$. Combining these two facts we get the following explanation for every item $i \in I$ committed to bin $j \in B$:

$$[[l_j \geq \text{SUM}(P_j) - s_i + 1]] \wedge [[C_j]]$$

NoSum Pruning. Since the NoSum algorithm only takes the lower bounds, upper bounds and candidate set of the bin as inputs, and doesn't allow for strengthening these values, the explanations for the following propagations are quite simple:

$$[[C_j]] \wedge [[l_j \geq \underline{l_j}]] \wedge [[l_j \leq \overline{l_j}]]$$

NoSum Load Bound Tightening. For lower bound load tightening:

$$[[C_j]] \wedge [[l_j \geq \underline{l_j}]]$$

For upper bound load tightening:

$$[[C_j]] \wedge [[l_j \leq \overline{l_j}]]$$

NoSum Item Elimination and Commitment. For both item elimination and item commitment, since the inputs are essentially the same:

$$[[C_j]] \wedge [[l_j \geq \underline{l_j}]] \wedge [[l_j \leq \overline{l_j}]]$$

6 Experimental Results

The goal of this section is to compare the constraints from Section 3 and their explanations from Section 5 to both a decomposition of the bin packing global constraint, and an implementation of our code with naive explanations for propagation. A decomposition of a global constraint breaks the constraint down into smaller, simpler constraints. The methods described in this paper are implemented in the Pumpkin[2] solver, developed at TU Delft, and the developed code can be found on GitHub[3]. The solver, and therefore the implementation, are written in Rust. Pumpkin can be used as a back-end solver for the MiniZinc[4] constraint modelling language. The decomposition experiments in this Section specifically regard the default decomposition done by MiniZinc.

MiniZinc also provides a yearly competition called The MiniZinc Challenge[5] containing a variety of benchmarks to be used on CP solvers. We used the 2019 steelmillslab and the 2022 team-assignment benchmarks. The steelmillslab benchmark only uses bin packing constraints, and will be used to analyse the performance of the constraint on its own. The team-assignment benchmark uses bin packing constraints as well as the all_different[6] constraint. This will allow us to analyse the implemented constraint in a problem where it works in conjunction with other constraints. On top of that, we have also made a version of the constraint that uses naive explanations. These naive explanations simply describe the domains of each variable at the time of the propagations, and the experiments will allow us to measure the effectiveness of our explanations.

We have selected the following subset of statistics we deemed most interesting:

- **Objective.** In steelmillslab, a minimization problem, a lower value indicates a better performance. In team-assignment, a maximization problem, a higher value indicates a better performance. In case the solver did not time out, the objective value will be listed as "completed", this is the best possible outcome.
- **Decisions.** The amount of decisions is a counter that keeps track of the amount of value assignments made in the search step of the solver.
- **Time.** A time limit of five minutes, or 300 seconds was set for every execution. If the solver timed out, we display the statistics of the last solution it found, if any.
- **Avg. Clause Length.** This indicates the average length of the clauses learned during execution. For example: $[[a \geq 1]] \vee [[b \neq 2]]$ has a clause length of two.
- **Avg. LBD.** This keeps track of the average literal-block distance (LBD) of the nogoods learned during execution.

[2] https://github.com/consol-lab/pumpkin

[3] https://github.com/MelvinDK/PumpkinBP

[4] https://www.minizinc.org/

[5] https://www.minizinc.org/challenge/

[6] https://docs.minizinc.dev/en/stable/lib-globals-alldifferent.html#all-different

Table 1: Statistics of the steelmillslab and team-assignment MiniZinc challenges. The statistics are given for both our implementation and the default MiniZinc decomposition for every benchmark data file.

steelmillslab	Version	Objective	Decisions	Time (s)	AvgClauseLen	AvgLbd
bench_2_19	our	70	17299	18,9	2,08	1,06
	decomp	N/A	11952	300	6585,15	88,00
bench_17_7	our	19	46597	208,6	325,98	77,68
	decomp	N/A	11483	300	6665,42	88,04
bench_19_6	our	14	33223	125,9	358,97	72,89
	decomp	N/A	11270	300	6676,00	88,03
bench_20_8	our	completed	37178	167,8	301,67	75,88
	decomp	N/A	11465	300	6659,41	88,04
bench_20_15	our	31	38975	169,7	356,19	77,78
	decomp	N/A	11337	300	6654,80	88,04
team-assignment						
data1_4_6	our	completed	100140	39,2	8,66	5,54
	decomp	completed	163662	58,1	55,29	7,00
data1_6_6	our	completed	338109	172,1	15,14	7,30
	decomp	completed	191271	80,9	63,97	8,51
data2_6_15	our	9654	29020	24,2	109,41	51,95
	decomp	8646	81987	115,2	554,78	51,81
data3_4_31	our	6357	67607	103,9	195,15	68,41
	decomp	8363	35556	69,7	729,16	79,94
data3_5_31	our	6271	64365	154,2	188,40	77,64
	decomp	8300	91983	267,2	754,90	108,72

6.1 steelmillslab

The top half of Table 1 shows the results of running our implementation and the decomposition on the steelmillslab minimization challenge on five benchmarks. Most notable is the observation that the decomposition was not able to find a single solution for any benchmark. Our implementation was able to find solutions for every benchmark, and even find and confirm the best solution in benchmark 20_8 in just under three minutes. In benchmark 2_19, our implementation was able to find a solution in 20 seconds, but it could not find an improvement or prove it was the optimal solution in the remainder of the time.

The average clause length is much smaller in our implementation, showing that our explanations are performing much better than the decomposition in this set of benchmarks. The large clause length in the decomposition could also explain why it was able to make less decisions in more time than our implementation, since the solver was spending more time evaluating the large learned clauses. This difference in clause length is then likely also why the decomposition was not able to find any solutions in the allotted amount of time. Finally, the average LBD was also slightly smaller in our implementation for all benchmarks except 2_19, where the difference was much more significant. Overall, in this problem with only a bin packing constraint, our version works a lot better than the decomposition.

6.2 team-assignment

The results from the team-assignment benchmark as seen in the bottom half of Table 1 are much more mixed than those from steelmillslab. Since this is a maximization problem, a higher objective value means better overall performance. Since our version completed data1_4_6 in less time than the decomposition, and finished data2_6_15 with a larger objective value, the overall performance based on those two metrics was better in those two benchmarks. However, in the other three, decomposition beat our implementation. Similarly, the amount of decisions is smaller for the version that performs better in each benchmark, directly correlating higher objective value and less time. Although our version had less decisions in data3_5_31, it also achieved a lower objective score, and according to the rest of the data, it is likely that given enough time, our version would reach the objective 8300 with a larger amount of decisions than the decomposition.

Where our version is consistently better however, is the average clause length. This number is significantly smaller in every benchmark. The average LBD is also smaller in every benchmark but data2_6_15. These statistics show that the clauses generated by the explanations in our implementation are marginally to significantly better, depending on the benchmark used.

In the steelmillslab benchmark, where there is only a bin packing constraint, our implementation outperformed the decomposition in essentially every statistic. For the team-assignment benchmark, which also contains an all_different constraint, it is unclear why the decomposition made less decisions than our implementation in some of the benchmarks. The average clause length and LBD however are a good indicator that our explanations work well, and show that the work done in this paper is worth improving upon in the future.

6.3 Naive Explanations

On top of our explanations, we have made a version that uses naive explanations to test the efficacy of our explanations.

Table 2: Statistics of the steelmillslab MiniZinc challenge. The statistics are given for both our implementation and a version of our implementation with naive explanations.

steelmillslab	Version	Objective	Decisions	Time (s)	AvgClauseLen	AvgLbd
2_19	our	70	17299	18,9	2,08	1,06
	naive	70	17235	152,8	7,25	6,25
17_7	our	36	16157	20,2	16,84	8,41
	naive	36	16996	170,7	49,56	48,56
19_6	our	26	17811	20,6	7,94	4,36
	naive	26	12613	250,4	51,57	50,57
20_8	our	17	11281	16,6	15,43	7,42
	naive	17	11873	147,5	56,79	55,79
20_15	our	44	14630	20,3	13,39	5,34
	naive	44	14853	165,6	44,95	43,95

Specifically, for every propagation we provide the lower and upper bounds of every bin load, the lower and upper bounds of every bin allocation, and every bin allocation in that range that is not in the domain of the variable. Because this requires iterating over the domain of every variable each time a propagation is made, this process intrinsically requires more time to run. Since this means that the naive implementation will not achieve the same objective value as our implementation in the allocated five minutes, we display the statistics of our version for the highest objective value that the naive implementation was able to achieve. See Table 2.

While in three out of five cases, our implementation leads to a smaller amount of decisions than the naive version, for two benchmarks this is not the case. The first benchmark has a small amount of extra decisions, but the third one has a significant 5.200 added decision points. While unexpected, we believe this is likely due to the different explanations causing a different value assignment in the search step, which lead to the exploration of a large infeasible search space. For example, think of performing a depth-first search and going the wrong way at the root, after which you will only find the solution after having fully explored the wrong path.

The average clause length and literal block distance are both consistently significantly smaller in our version than in the naive implementation. This shows that our explanations are much smaller and more effective at creating nogoods than the naive implementation. While the quality of the explanations is clearly better, any positive effects they might have on the search are not completely evident.

7 Conclusion

In this paper we developed intuitive and effective explanations for existing bin packing constraints. These constraints and explanations were implemented in Pumpkin, and compared against a decomposition of the bin packing constraint. Using MiniZinc benchmarks, we learned that there is an overall performance increase in the steelmillslab problem, a pure bin packing problem. In the team-assignment benchmark however, where another global constraint is involved, the results are mixed. While the amount of decisions and the time spent on finding a solution vary from benchmark to benchmark, sometimes leading to worse performance for our version, there is an improvement in average clause length and LBD. These statistics indicate the usefulness of the clauses generated during execution. Both clause length and LBD were smaller for our implementation in almost all benchmarks, indicating better explanations than those of the decomposition.

When comparing our solution to a version with naive explanations for the propagations, we notice a mixed result in the amount of decisions made. While in a small majority of the cases, the number decreases, in one benchmark the naive implementation starkly outperforms our version. Similarly to with the decomposition, our version vastly outperforms the naive implementation in regards to average clause length and LBD, showing the generation of shorter explanations and more effective nogoods.

While we used P. Shaw's model [10], other papers on bin packing constraints exist such as the one by P. Schaus [13]. There is merit in trying to implement their model in the future, and trying to create explanations for it. On top of that, several options for calculating lower bounds on the bin count exist, and creating propagators based on those as described by P. Shaw [10] are worth adding to the implemented constraint. Finally, many propagations use the *candidate set* in their explanation, which is the largest part of those explanations. There is potential for improving these explanations by introducing so called *contains* variables, showing whether a bins candidate set contains a specific item. Their usefulness is unknown, and is worth analysing in future work.

8 Responsible Research

This paper attempts only to further the state of the art of solving problems that can at least partially be described by the bin packing problem. The responsibility of using our methods for ethical applications lies entirely with the user.

The data we have presented shows only the subset of statistics available we deemed most interesting. Other statistics such as number of restarts, average backtrack amount and number of conflicts were available to us, but we chose not to present these statistics to keep the data concise.

The code implemented for this paper is entirely open source, and recreating the results or measuring the other statistics can be done by running the code found on our GitHub repository[7].

[7] https://github.com/MelvinDK/PumpkinBP

The README file gives full instructions for the execution of the code.

We chose to use the benchmarks steelmillslab and team-assignment for this paper for several reasons. Firstly, all problems in these benchmark families had other solvers that either solved or found a solution for them. Secondly, we thought it would be interesting to pick one benchmark with only a bin packing constraint, and one benchmark with other constraints as well to show the difference between the two. While it would have been beneficial to run more benchmarks, we decided on only using these two so it would allow us to clearly present the data obtained without exceeding the page limit. For other MiniZinc benchmarks not included in our repository, visit the MiniZinc Challenges page[8]. Any problem with either bin_packing or bin_packing_load are applicable for our implementation.

References

[1] M. R. Garey and D. S. Johnson. *Approximation Algorithms for Bin Packing Problems: A Survey*, pages 147–172. Springer Vienna, Vienna, 1981.

[2] K. S and M. Nair. Bin packing algorithms for virtual machine placement in cloud computing: A review. *International Journal of Electrical and Computer Engineering (IJECE)*, 9:512, 02 2019.

[3] K. L. Krause, V. Y. Shen, and H. D. Schwetman. Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems. *J. ACM*, 22(4):522–550, October 1975.

[4] E. G. Coffman, J. Y.-T. Leung, and D. W. Ting. Bin packing: Maximizing the number of pieces packed. *Acta Informatica*, 9(3):263–271, Sep 1978.

[5] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Dynamic bin packing. *SIAM Journal on Computing*, 12(2):227–258, 1983.

[6] T. Feydy and P. J. Stuckey. Lazy clause generation reengineered. In I. P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, pages 352–366, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[7] A. Schutt. *Improving scheduling by learning*. University of Melbourne, Department of Computer Science and Software Engineering, 2011.

[8] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[9] O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation = lazy clause generation. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 544–558, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[10] P. Shaw. A constraint for bin packing. In M. Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 648–662, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[11] S. Martello and P. Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Mathematics*, 28(1):59–70, 1990.

[12] H. Cambazard and B. O'Sullivan. Propagating the bin packing constraint using linear programming. In *International Conference on Principles and Practice of Constraint Programming*, pages 129–136. Springer, 2010.

[13] P. Schaus. Solving balancing and bin-packing problems with constraint programming. *These de doctorat, Université catholique de Louvain*, 2009.

[8] https://www.minizinc.org/challenge/