Evaluating the usefulness of Global Cardinality constraint propagators in Lazy Clause Generation

Comparing propagator implementations with explanatory clauses for the Global Cardinality constraint against decomposition in the Pumpkin Lazy Clause Generation solver

David Thomas Rockenzahn Gallegos[1]

Supervisor(s): Dr. E. Demirović, Dr. M.L. Flippo

[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: David Thomas Rockenzahn Gallegos
Final project course: CSE3000 Research Project
Thesis committee: Dr. E. Demirović, Dr. M.L. Flippo, Dr. B.P. Ahrens

Abstract

In Constraint Programming, combinatorial problems such as those arising in the fields of artificial intelligence, scheduling, or circuit verification are modeled using mathematical constraints. Algorithms for each type of constraint are implemented in a solver and are known as propagators. Some constraints can be implemented using combinations of smaller existing propagators (this is known as decomposition), which is often used in Lazy Clause Generation (LCG) solvers, where research has shown that decompositions can be competitive or sometimes superior to purpose-built propagators for certain kinds of constraints. However, there is little research about whether a purpose-built propagator is superior to decomposition for the `global_cardinality` constraint when used with an LCG solver. This paper benchmarks both approaches using an experimental Rust-based solver and uses existing algorithms to implement two `global_cardinality` propagators which are then adapted for use in an LCG solver. The benchmarks show that both implementations are competitive or outperform decomposition in a dataset of 90 instances of Sudoku puzzles, being 3.19 and 2.28 times faster for Régin GAC and Basic Filter algorithms respectively. On the Minizinc Challenge dataset, the speedup is and 1.34 and 1.0.

1 Introduction

Constraint Programming (CP) is a technique for solving combinatorial problems that often arise in the fields of artificial intelligence [11], scheduling [3], circuit verification [19], and Puzzle Solving. These problems are modeled using combinations of constraints; mathematical restrictions or rules that a solution of the problem must adhere to. Some constraints appear in many different problems, and have thus been given names. Because they are so common, a lot of effort has gone into developing algorithms to implement them. Such algorithms are called Propagators. Constraint Programming Solvers often include many different kinds of propagators, despite the fact that some of those constraints can be decomposed into simpler constraints and would thus seem "redundant" at first glance. This is because it has been shown that such redundant but highly-specialized propagators are often more efficient than their decomposition into smaller constraints [16] This paper focuses on the `global_cardinality` (`gcc`) constraint. It can be used to set bounds on how many times values can be assigned to groups of variables. For example, if one is interested in assigning students to groups with different capacities one could use a `global_cardinality` constraint to ensure that every group is assigned at least two students and that no group is assigned more students than it has capacity for. See *Figure 1* for an example. The `global_cardinality` constraint is also enough on its own to solve problems such as Sudoku. One can use nine `gcc` constraints so that each value one to nine appears one time on each column, another nine constraints for each row, and nine more for each square.

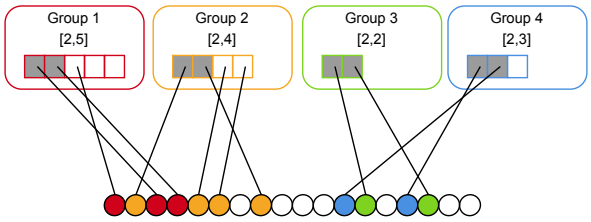

Figure 1: An example student-to-group problem corresponding to the constraint $\texttt{gcc}(X, V)$ with $X = \langle x_1, ..., x_{18} \rangle$ And $V = \langle (1, 2, 5), (2, 2, 4), (3, 2, 2), (4, 2, 3) \rangle$). Every tuple $(v, l, u) \in V$ has a value, a lower and upper bound constraining how many times that value may be assigned.

The `gcc` often arises in scheduling problems. For example, efficiently scheduling physicians during a pandemic where use of a `global_cardinality` constraint may improve the algorithm's performance [3]. Another example is the rotating workforce scheduling problem in which workers are assigned to shifts while ensuring enough shifts are covered each day. Among all the constraints and models used, the ones using a `global_cardinality` constraint achieved the best results overall [6].

For a time, examples like these have demonstrated the benefits of highly-specialized global constraint algorithms, but in 2007 a novel technique called Lazy Clause Generation that combines the use of propagator algorithms with a SAT solver can achieve state-of-the-art performance [8]. LCG and constraint decomposition has brought into question the need for global constraints like the `gcc` [17].

Later on it was discovered that using LCG with a decomposition into smaller constraints instead of using the `cumulative` constraint can achieve very competitive results [15], defying the conventional wisdom that specialized constraints are often more efficient. All this considered, there is little research about implementing `gcc` propagators for LCG and whether (or under which circumstances) a specialized `gcc` propagator is still needed.

The goal of this paper, therefore, is to investigate whether a `global_cardinality` constraint propagator implementation with explanatory clauses for use in Lazy Clause Generation solvers can compete in performance against its decomposition into smaller constraints. To answer the question two different propagators implementations will be benchmarked using the Minizinc Challenge problems [18] against the usual decompositions in the literature. The algorithms will be compared using their runtime, highest achieved objective value, and statistics such as Learning Block Distance (LBD), average Learned Clause Length and average Conflict Size.

The results show that a propagator implementation using Régin's [13] Generalized Arc Consistent algorithm is competitive against decomposition, with similar results using two kinds of explanation algorithms, while the Basic Filter algorithm does not provide a big improvement. This might chal-

lenge the new conventional wisdom in LCG that decomposition tends to beat global propagators. While the runtime and objective value measures favor Régin GAC, the LBD statistic is less conclusive.

Section 2 of this paper serves as an introduction to LCG, an overview of the related work, and a formal description of the `global_cardinality` constraint. Lastly, an overview of Régin's GAC algorithm is discussed. Section 5 then focuses on how a propagator for the `gcc` can be adapted for use in a LCG solver, and on how the algorithms for the benchmark were implemented. In the experimental setup and results, section 6, the benchmarks are discussed. Finally, section 8 goes over the ethical considerations and reproducibility of the research before giving way to the conclusion.

2 Preliminaries

The next section focuses on a high level overview of the background of LCG. A more detailed mathematical review of the preliminaries follows. Then a summary of the related work is presented, and finally a mathematical definition of the problem is laid out.

2.1 Overview of LCG and explanation algorithms

Constraint Satisfaction solvers using techniques such as Finite Domain (FD) Propagation solve combinatorial problems by modeling restrictions in the solution as constraints. FD solvers alternate between:

- Search: Traversing the search space by checking for solutions among all possible assignments to some group of variables.
- Propagation: Ruling out parts of the search space using propagators implemented for the problem's constraints. Good propagators need make a trade-off between ruling out as many invalid assignments as possible, while also taking little time to run. [17]

Lazy Clause Generation (LCG) is a newer technique, but has many steps in common with FD propagation. LCG combines Propagation with a SAT Solver to achieve state-of-the-art solutions for some problems [2]. It has also uses search and propagation like FD solvers, and in addition it does:

- Explanations: Translating a propagator's search-space reduction into an implication of boolean clauses which makes interfacing with the SAT solver possible.
- SAT Solving: The explanation is fed to the SAT solver which uses it to continue the search, rule out further assignments and find solutions to the problem.

Explanations are therefore needed because a SAT solver operates on boolean variables, and propagators use variables that can be integers. Conceptually, LCG can be seen as a way to lazily transform the domain reductions of the propagator into a boolean representation.

Because both techniques use propagators, a propagator algorithm for the `gcc` would also work in Lazy Clause Generation as long as an explanation is created to encode the domain reduction the propagator does. There is a great deal of algorithms in the literature for propagators such as the `gcc`, so the main challenge for developing algorithms for LCG solvers is in the explanations. Developing good explanations can be as challenging to find as developing good propagators, and a good propagator might not lend itself to a way of obtaining good explanations, so the algorithm might have to be redesigned. [12]

2.2 Mathematical definitions

An overview of Constraint Programming and some important definitions. [14]

Definition 1 (Constraint Programming)**.** A programming paradigm to solve combinatorial problems. It involves finding valid assignments to sequences of variables $X = \langle x_1, x_2, ..., x_n \rangle$

Definition 2 (Domain)**.** mapping from a sequence of variables X to a sequence of sets. A set $D(x_i)$ contains all values that variable x_i is allowed to take. The notation $[a, b]$ represents the set $\{a, a+1, a+2, .., b\}$

Example. Let $X = \langle x_1, x_2 \rangle$. Possible domains for these variables are: $D(x_1) = \{0, 1\}$, $D(x_2) = [1, 4]$

Definition 3 (Constraint)**.** The sets of allowed values a sequence of variables can take. Formally: An n-ary relation over the domains $D(x_1), D(x_2), ..., D(x_n)$ of a sequence of variables $X = x_1, x_2, ..., x_n$. An n-ary relation is a subset of the Cartesian product of some sets. Therefore a constraint describes the solution space by a subset of the Cartesian product over the domains $C(X) \subseteq D(x_1) \times D(x_2) \times ... \times D(x_n)$. Constraints can also be implicitly represented using most math operators such as $=, \leq, \neq, +, -, \cdot, \wedge$.

Example. Using the domains from the previous example a constraint might be $C(X) = \{(0, 1), (1, 2)\}$. This would mean x_1 can only be 0 or 1 and x_2 can only be 1 or 2. An implicit way to define this is: $[x_1 < x_2] \wedge [x_2 \leq 2]$

Definition 4 (Constraint Satisfaction Problem (CSP))**.** Denoted by $(\mathcal{C}, \mathcal{X}, \mathcal{D})$, it is a sequence of variables $\mathcal{X} = x_1, x_2, ..., x_n$, their respective domains $\mathcal{D} = D(x_1), D(x_2), ..., D(x_n)$ and a set of constraints $\mathcal{C}$ in which each constraint $C(X) \in \mathcal{C}$ is defined over a subsequence of variables $X \in \mathcal{X}$.

Definition 5 (Assignment)**.** A mapping from a sequence of variables $\mathcal{X} = x_1, x_2, ..., x_n$ to one element from each domain. An assignment satisfies a constraint if the assignment is also an element of the constraint.

Example. An assignment for $\langle x_1, x_2 \rangle$ that satisfies the constraint of the previous example is $x_1 = 0, x_2 = 1$.

Definition 6 (Solution of a CSP $(\mathcal{C}, \mathcal{X}, \mathcal{D})$)**.** An assignment to the variables $\mathcal{X}$ that satisfies all constraints $\mathcal{C}$

Definition 7 (Propagator)**.** A propagator f is a monotonically decreasing function from domains to domains, such that $\forall x \in X : f(D_1)(x) \subseteq f(D_2)(x)$ for all domain sequences D_1 and D_2 with $\forall x \in X : D_1(x) \subseteq D_2(x)$, that also preserves the solutions for any CSP.

- A propagator creates a new domain, or equivalently it creates a new CSP problem $f(\mathcal{C}, \mathcal{X}, \mathcal{D}) = (\mathcal{C}, \mathcal{X}, f(\mathcal{D}))$.

- A propagator is usually implemented for one specific constraint (that is to say, the implementation depends on the constraint).

Propagators are used to prune the search space. Propagators better at pruning are more desirable. A notion of "strength" is used to classify how good the pruning (how small the resulting domain) is. The trivial propagator (that always leaves the domain unchanged) is the weakest possible propagator. Formal definitions of "strength" have been devised. For example, the notion of Generalized Arc Consistency.

Definition 8 (Generalized Arc Consistency or Domain Consistency (GAC))**.** A constraint $C(X)$ is domain consistent with respect to domain D iff:

$$\forall x_i \in X, \forall d_i \in D(x_i), \exists \theta \in D(X) : \theta \text{ satisfies} C(X)$$

Here θ represents an assignment. It can be seen intuitively how this definition is desirable: The domain has been reduced so much that any single choice of assignment of one variable has an assignment for the other variables that satisfies the constraint. This is the strongest form of consistency that can be achieved by a propagator, but finding an algorithm that achieves it can be NP-hard for some constraints.

While methods such as Finite Domain Propagation and Lazy Clause Generation both use propagators and work in similar ways, one of the main differences is that Lazy Clause Generation uses a SAT solver as an additional propagator to exploit some of the SAT solver's strengths. [8]. For a propagator to work with LCG it needs to implement an "explanation" [12].

Definition 9 (Explanation)**.** A propagator f that reduces the domains of a sequence of variables X to $f(X)$ can be explained by using a subset of original constraints $C' \subset C$ and choices made during search $d_1, d_2, ..., d_n$ such that:

$$C' \wedge d_1 \wedge d_2 \wedge .. \wedge d_n \Rightarrow f(X)$$

The explanation E is then $e = C' \wedge d_1 \wedge d_2 \wedge .. \wedge d_n$.
An explanation E_1 is more precise than E_2 iff $e_2 \rightarrow e_2$. More precise explanations are more useful because they allow the SAT solver to more efficiently prune domains and use less clauses to represent, taking less memory.

3 Related Work

Implementing a propagator that achieves GAC is trivial using brute force search, but a GAC algorithm is only useful in practice if it also has a polynomial runtime. Régin [13] mentions that the `gcc` can be decomposed into multiple local propagators using `min` / `max` constraints but shows that this is not efficient (for FD propagation). He then gives the first polynomial-time algorithm ($O(|x|^2 \cdot |v|)$) for achieving GAC using flow theory. A new algorithm for bounds consistency of the `gcc` with a runtime of $O(t + |x|)$ was found [9] where t is the time to sort the bounds of the domains of the variables. Despite this propagator having a weaker form of consistency the lower runtime can still outperform Régin's algorithm on some problems. The paper also introduces an improvement to Régin's algorithm to lower the worst-case performance in some applications. The GAC propagator was improved [4] and "the first efficient algorithm that achieves bound consistency for all variables, and not only the assignment variables" was discovered. An algorithm was proposed with an $O(|x|^{1.5}|v|)$ runtime complexity. [10] . Various constant-factor runtime optimizations on Régin and Quimper's algorithms [7] are found.
So far the aforementioned papers operate on the assumption that the `gcc` algorithms are superior over their decompositions (and this is true for Finite Domain propagation) but this began to change somewhat after the discovery of Lazy Clause Generation [8]: The reason is that decomposition with LCG achieves very competitive performance over specialized FD propagators. [15]
After Lazy Clause Generation and similar methods that require explaining the propagators, explanations can be developed for the `all_different`, `stretch` and `flow` constraints and the techniques are often applicable in general for other propagators, but that finding explanations is a challenging process because it requires a lot of domain-specific knowledge about the workings of the propagator [12]. Furthermore, the `gcc` is a specialization of the `flow` constraint and as such, an explanation for the `gcc` would be very similar. [12]

4 Problem definition

4.1 Formal definition of the global cardinality constraint

Recall the example problem from *Figure 1*. In order to solve it using a Constraint Satisfaction Toolkit it needs to be modeled as a Constraint Satisfaction Problem. In the example there are 18 students. Every student can be represented using a natural number i representing their position such that $i \in [1, 18]$. Which group each student i has been assigned to can be represented as the variable x_i. Any student x_i can be assigned to one of the four groups or to no group. Therefore the domain of x_i is: $\forall i \in [1, 18] : D(x_i) = \{0, 1, 2, 3, 4\}$. Furthermore, every group has a certain capacity. The capacity of group one is the set $\{2, 3, 4, 5\}$ or simply $[2, 5]$ because every group needs to be assigned at least two people and group one can fit at most 5. The capacities of all 4 groups are then: $c_1 = [2, 5], c_2 = [2, 4], c_3 = [2, 2], c_4 = [2, 3]$. If one is interested in finding all valid assignments of students to groups one can use a `global_cardinality` constraint to model and solve the problem.

Definition 10 (Global Cardinality Constraint)**.** Given a list of variables $X = x_1, x_2, ..x_n$ with domains $D_1, D_2, ..D_n$ and a list of tuples $V = (v_1, min_1, max_1), (v_2, min_2, max_2), ..., (v_m, min_m, max_m)$ of a value $v_i \in \mathbb{Z}$ and an upper and lower bound for that value min_i, max_i, the `global_cardinality`$(\langle X \rangle, \langle V \rangle)$ constraint enforces that each value v_i is assigned to the variables in X at least min_i times and at most max_i times.

For example, the `global_cardinality`(X, V) constraint with $X = \langle x_i, x_2, .., x_{18} \rangle$ where $\forall i \in [1, 18] : D(x_i) = [1, 4]$ and $V = \langle (1, 2, 5), (2, 2, 4), (3, 2, 2), (4, 2, 3) \rangle$ models the constraints of the problem in *Figure* **??**. A CSP solver will

then output the assignment in the Figure, which is the same as $X = \langle 1, 2, 1, 1, 2, 2, 0, 1, 0, 0, 0, 4, 3, 0, 4, 3, 0, 0 \rangle$ and all other valid assignments. Notice that some students are assigned to no group, because the gcc does not require that this is the case.

Note that in a real-world problem multiple constraints are usually combined in ways that make finding the solutions ontrivial. For example, one might also require that every student must be in a group, or ask the student for preferences and require that each student end up in one of their top 3 groups by preference. In this case the gcc and its propagator are just one of the building blocks to find a solution.

4.2 An arc consistent algorithm for the gcc

This section will discuss Régin's [13] GAC propagator for the gcc that was mentioned earlier in Section 2, because the implementation and benchmarks in Sections 5 and 6 make use of this algorithm. It is a high level overview. The proofs of correctness and other details can be found in the paper itself.

The propagator for an instance of the gcc like the one in Figure 1 works as follows:

- A graph is constructed like in Figure 2.
  - The source node connects to the value nodes, using the value's upper and lower counts as upper and lower bounds for the flow, respectively.
  - A value node connects a value node to a variable node if the variable has that value in its domain. $[0, 1]$ is used as the lower and upper capacities.
  - All variable nodes are connected to the sink, with lower and upper capacities of $[0, 1]$.
  - The sink connects to the source with capacities $[0, \infty]$
- The augmenting path method is used to find a feasible flow in the graph (a flow that satisfies all lower bound capacities).
- Using a feasible flow, a maximum flow is found using a slightly modified Ford-Fulkerson algorithm. This is a flow that is maximum (but less than the sum of the upper bound capacities) and also satisfies the lower bound capacities. The FF algorithm also returns a residual graph that is used to prune the domains of the variables.
- Tarjan's algorithm is used to find Strongly Connected Components (SCCs) in the residual graph of the maximum flow.
- If an edge connecting a variable node x and a value node v are not in the same SCC, then that edge is not used in any maximum flow in the graph, thus the value v is not part of the domain of x in any possible solution. This means that v can be removed from the domain of x.

5 Main contributions: The implementation of the propagators for the gcc

The research question is to investigate the feasibility of a gcc implementation in LCG when compared against decompositions. As has been mentioned in the previous section, Lazy

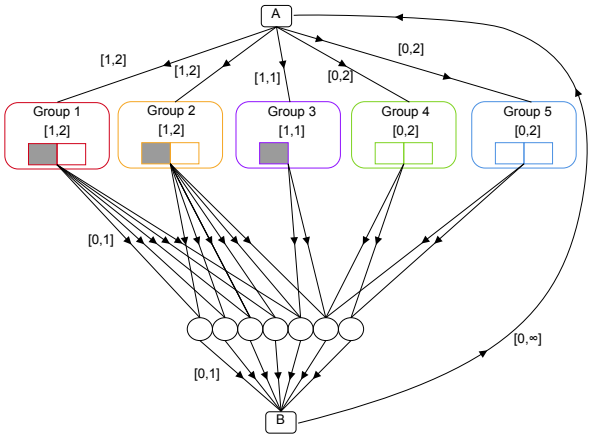

Figure 2: An example of the graph constructed for Régin s GAC algorithm for gcc$(\langle x_1, ..., x_{18} \rangle, \langle (1, 1, 2), (2, 1, 2), (3, 1, 1), (4, 0, 2), (5, 0, 2) \rangle)$

Clause Generation requires that propagators be modified to explain their decision to remove a value from the domain of one of the variables. The main challenge lies in how to adapt existing algorithms for the gcc propagator for use in LCG and in developing explanations for them. Therefore, this section focuses on the following three points:

- Implement and adapt the existing algorithms for the gcc to make them work in an LCG solver.
- Develop explanation algorithms for the chosen propagator algorithms.
- Investigate which steps can be performed to improve the propagator's performance to make them as competitive as possible against already optimized decompositions.

In this section, two different propagation algorithms are treated: An algorithm following the Arc consistent GCC propagator proposed by Régin [13], and a simpler but faster filtering algorithm of no particular consistency.

5.1 The naive explanation algorithm

The following explanation can be used for any propagator. Note that $H(x)$ returns the holes in the domain of x.

$$E_{\text{naive}}(X) = \bigwedge_{\forall x \in X} \left([x \geq \text{lb}(x)] \wedge [x \leq \text{ub}(x)] \bigwedge_{h \in H(x)} [x \neq h] \right) \tag{1}$$

The explanation for failure is then $E_{\text{naive}}(X) \Rightarrow \bot$, and the explanation for a removal of value v from the domain of variable x is $E_{\text{naive}}(X) \Rightarrow [x \neq v]$

This explanation is a conjunction using the upper and lower bounds of variables $x \in X$ as well as the negation of any holes in its domain (values that have been removed between the upper and lower bound). This explanation describes the domain of all variables X at the current position in the search space. As such, it is the most general explanation possible because it works to explain any propagation step or failure. However it ends up using many literals and because it is only

logically true in the current position on the search space, it will not allow the SAT solver to use "nogoods" to short circuit search at any other point.

Since the SAT solver already uses clauses to constrain the domain of the variables to their initial domain, an equivalent explanation is:

$$\left(\bigwedge_{\forall x \in X}\left(\bigwedge_{\forall v \in D_0(x)}[x \neq h] \quad \textbf{if} \quad h \notin D(x)\right)\right) \Rightarrow [x \neq v] \tag{2}$$

Where $D_0(x)$ is the initial domain, and $D(x)$ the current domain of x. This explanation is the conjunction of all the values that have been removed during the search (they are in the initial domain, but not in the current domain). This is logically the same as describing the domain of every variable and when automatically minimizing clauses it offers no benefit in terms of the clause length, but can be more concise and easier to reason about.

5.2 The simple filtering propagator

This kind of propagator uses simple rules to prune domains and detect inconsistencies. As such, the general ideas are not a novel method, and similar algorithms are implemented in some existing solvers because these filtering algorithms can be simple, propagate quickly, and are easy to reason about but their pruning strength tends to be low.

A propagator for the simple filtering algorithm

Let gcc(X, V) where $V = \langle v, \text{lower(v)}, \text{upper(v)}\rangle$ are the values, with upper count and lower count that variables x can take. Let max_count(v) be defined as the number of variables of the gcc that have been fixed to value v during the search (v is the only value currently in their domain). Let min_count(v) be the number of variables containing v in their domain, and let $E(X)$ be the (left side of) any valid explanation algorithm, for example the general explanation from equation (2). Then the basic filter algorithm for the gcc(X, V) is:

Algorithm 1 Basic Filter

```
1: for v ∈ V do
2:     if min_count(v) > lower(v) ∨ max_count(v) < upper(v) then
3:         return E(X) ⇒ ⊥                          ▷ Inconsistency
4:     end if
5:     for x ∈ X|v ∈ D(x) do
6:         if min_count(v) + 1 > upper(v) then
7:             E(X) ⇒ [x ≠ v]                       ▷ x can't have value v
8:         end if
9:         if max_count(v) − 1 > lower(v) then
10:            E(X) ⇒ [x = v]                       ▷ x must have value v
11:        end if
12:    end for
13: end for
```

Line 2 essentially checks whether the gcc is satisfied. If not, it explains why the failure happened. Lines 6 and 9 use two simple rules to prune values from the domain of the variables: If assigning value v to variable x causes the $min\_count(v)$ to be greater than the upper bound this would make the problem inconsistent. Therefore we must remove v from $D(x)$. If not assigning variable x to this value v would make the $max\_count(v)$ lower than the lower bound, the problem becomes inconsistent. Therefore $D(x) = v$.

Explaining the simple filtering algorithm

As is mentioned in the previous subsection, the general explanation can be used to explain failures or prunings done by the Basic Filter algorithm. The performance of the algorithm can be improved by using a more specific explanation.

An alternative explanation algorithm for Basic Filter

An explanation that can be used for the Basic Filter algorithm 1 is:

$$E_{\text{filt1}}(X, v) = \left(\bigwedge_{x \in X}[x = v] \quad \textbf{if} \quad x = v\right) \tag{3}$$

$$E_{\text{filt2}}(X, v) = \left(\bigwedge_{x \in X}[x \neq v] \quad \textbf{if} \quad v \notin D(x)\right) \tag{4}$$

The problem is inconsistent when min_count $>$ lower(v), in which case the explanation for failure is $E_{\text{filt1}}(X, v) \Rightarrow \bot$. Intuitively, the inconsistency happened because of all variables x that are assigned value v. The problem can never be consistent as long as all those variables are assigned to v, and might become consistent again if v is removed from at least one of the variables such that the count is less than or equal to lower(v). Similarly, for the inconsistency max_count $<$ upper(v), the explanation for failure is $E_{\text{filt2}}(X, v) \Rightarrow \bot$.

The same explanations can be used to justify domain pruning (because the underlying condition is the same for detecting inconsistencies and removals in this algorithm). If x can't have value v then the explanation is $E_{\text{filt1}}(X, v) \Rightarrow [x \neq v]$, and if x must have value v it is $E_{\text{filt2}}(X, v) \Rightarrow [x = v]$.

5.3 The arc consistent propagator

The arc consistent propagator uses the algorithm explained in Section 2, with a small modification and an algorithm added to explain removals and failures during the search to make it work with Pumpkin, the LCG solver.

A propagator for the arc consistent algorithm

The arc consistent propagator uses the Régin GAC propagator algorithm from Section 2, with a modification to how a feasible flow is found: Régin [13] computes a feasible flow using augmenting paths starting from an infeasible flow and then uses Ford-Fulkerson (with a modification to take into account lower bounds) to find a maximum flow. However, is also possible to use Ford-Fulkerson for finding both the feasible flow as well as the maximum flow. For this the weight of the edges connecting the sink to the values defined in the gcc are set to the lower bounds before running an unmodified Ford-Fulkerson. The resulting maximum flow will be a feasible flow if the total flow is the same as the sum of the lower bounds (if all lower bounds are satisfied, the flow is feasible). Now that there is a feasible flow, the edges connecting the sink to the values are set back to the upper bounds, and a

modified Ford-Fulkerson is used as described in the original paper.

Developing a new explanation for Régin GAC

The idea behind this explanation hinges on the fact that Régin's algorithm uses strongly connected components (SCC) to remove values from the domains: If a variable x and one of its values v are in different strongly connected components of the residual graph after running the algorithm, then that value v is removed from the domain of x. Intuitively, if the variable and the value nodes were in the same SCC and now are not, the cause must have been one removal from the domain of some variable (since that is the only change that can happen during search). The minimal explanation then consists of all the removed variable-value edges that cause a variable x and its potential value v to be in two SCCs. The explanation of Rochart [12] for `all_different` and of Katsirelos [5] to develop so called g-nogoods can be adapted to create an explanation for the `gcc` for domain pruning.

$$E_{\text{Régin}}(x, v, X, V, G_0, G) = \bigwedge_{(v',x')\in G_0} [x' \neq v'] \textbf{ if } (v', x') \notin G \wedge \mathrm{J}(x', v', R(G), x, v) \tag{5}$$

Note that $(v', x') \in G_0$ refers all value-to-variable edges in the original graph at the start of search. $J(v', x', R(G), x, v)$ returns True iff there is a path in the current residual graph $R(G)$ starting at x and ending at v when adding the edge (v', x') to $R(G)$.

Intuitively, if there is a value-variable edge (v', x') that is in the original graph, but not in the current one, then it must have been removed during search. There exists a path from v to x in $R(G)$ because it contains the edge (v, x) (since the edge is being selected for removal by the Régin algorithm). If adding back the removed edge (v', x') creates a path from x to v in $R(G)$, it must be that x and v are in a cycle, since there is a path from v to x and from x to v. This means x and v are in the same SCC when (v', x') is added, and in a different SCC when it is absent. Therefore $[x' \neq v']$ must be part of the explanation.

6 Experimental Setup and Results

The following experiments are performed to benchmark the performance of a `gcc` propagator in LCG against decompositions:

- Compare the performance of the two propagator algorithms on a problem where it is easy to generate instances of different sizes.
- Compare the performance of the two propagator algorithms with their explanation algorithms against the decomposition on multiple problems that resemble those encountered in the real world.
- Compare the performance improvements of the new explanation algorithms and optimizations.

While the `gcc` propagators can be bench marked on simple example problems or puzzles such as Sudoku, this alone is not likely to provide a realistic estimate of the performance one would get in a real world problem. To address this, the benchmarks also make use of the Minizinc Challenge [18] problems. These problems are designed to compare the performance of different Constraint Programming solvers against each other in a competition. Since a big part of the performance of a CSP solver are its propagators, the problems are also well suited to compare propagator implementations against each other. Problems from the challenge make use of a balanced number of the common global constraints, and some problems have a larger or smaller number of them. Additionally, the whole corpus of problems is public and specified in the Minizinc constraint programming language, a common standard that many solvers accept, which is a benefit in terms of reproducibility. The experiments use all problems that include the `global_cardinality_low_up` constraint.

All algorithms have been implemented and benchmarked in the source available Rust based Pumpkin CSP solver[1]. A separate Python test bench has been created to run the problems on Pumpkin, collect the results and generate plots. The problems from the Minizinc Challenge which are written in the Minizinc language are run via Pumpkin's compatibility layer for Minizinc. The decomposition of the `gcc` used for benchmarking is performed automatically by the Minizinc language and decomposed into smaller constraints that Pumpkin supports. These smaller constraints are assumed to be highly optimized.

All experiments were run on a 5.5 GHz AMD Ryzen 5 7600X with 6 physical and 12 logical cores. Experiments were run in parallel but up to 5 at a time, in random order, to reduce the effect of OS thread scheduling with all instances in a random order.

6.1 Comparing the gcc against decompositions in Sudoku

The `gcc` is the only type of constraint needed to solve Sudoku puzzles. This makes it good to benchmark the performance of these propagators without the interference of having to use other constraints to model the problem. Furthermore, instances of the puzzle are easy to generate and problems can be made arbitrarily large using any number (n, n) of grids instead of the usual $(9, 9)$.

A dataset [2] of 90 Sudoku puzzles was used to compare the runtime of the Régin GAC algorithm and the Basic Filter algorithm against the Minizinc decomposition (see Figure 3). Both Régin GAC and Basic filter use the naive explanation algorithms. The missing bars are of problems where the corresponding algorithm did not terminate within the time limit of 10 minutes. The results show that the geometric mean of the speedup ratio across all problems is $(3.42, 1.52)$ for Régin GAC and Basic Filter with decomposition as baseline. In other words, Régin GAC is, on average, 3.42 times faster

[1] Pumpkin solver: https://github.com/consol-lab/pumpkin

[2] Sudoku problems: https://web.archive.org/web/20200121075758/http://hakank.org/minizinc/sudoku/problems2/index.html

than decomposition.

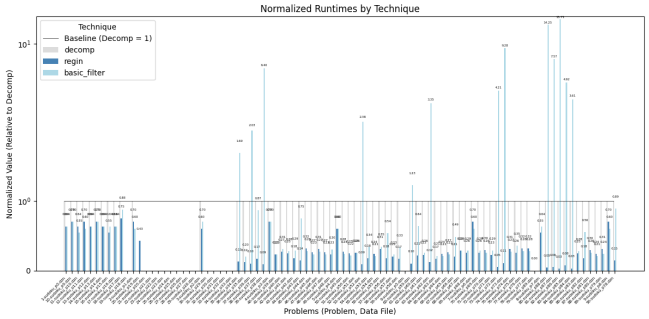

Figure 3: Runtime benchmark of the Régin GAC and basic filter propagators against decomposition. Run-times are normalized so that decomposition acts as the baseline of one.

6.2 Comparing the gcc against decompositions on the Minizinc Challenge problems

Figure 4 shows the normalized runtime of Régin's GAC propagator and the Basic Filter implementations, both using the naive explanation algorithms. The runtime of the decomposition is normalized by setting it to one. Some problems have no bar in the plot because no solution was found within the time limit (20 minutes, which is the standard time used to grade the Minizinc challenge winners). Some of these problems are optimization problems, where they are not expected to be solvable within the time limit, and instead an objective measure is used to indicate the better algorithm. The objective is plotted in Figure 4 as well, when applicable.

The geometric mean of the speedup ratios of the runtime are $(1.34, 0.91)$ for Régin GAC and Basic Filter respectively.

Régin GAC tends to be the best algorithm on "evmopt" (a compilation problem), physician scheduling and especially on rotating workforce scheduling while decomposition is marginally superior on community detection and vaccine.

The drawbacks of using Runtime is that it is a metric subject to changes in the OS thread scheduler, it does not provide information when the algorithm does not terminate, and it does not indicate if performance gains come from the low runtime of the propagator, it's pruning strength, or the strength of the explanations it produces. To address some of these weaknesses, a number of other statistics have been gathered.

Learned Block Distance (LBD) [1] is a standard heuristic to evaluate the quality of learned clauses in SAT solvers. The explanations provided by the `gcc` propagator are used by the SAT solver to learn clauses and improve the search. Therefore LBD provides a good way to measure the usefulness of a propagator's explanations. The lower the LBD, the more useful the explanation. Learned clause length measures the length of boolean clauses that the SAT solver develops from the propagator's explanations and conflict size measures the size of conflicts the SAT solver constructs from the propagator's filtering and conflict explanations. Generally, the smaller these two the better, as the SAT solver can then keep track of more of them. Figure 5 shows Régin GAC overall has a lower LBD, Learned Clause Length and Conflict Size. Basic Filter has a higher LBD: The geometric mean among all problems of the computed statistics is: LBD $= (0.90, 1.05)$, LCL $= (0.91, 0.95)$, CS $= (0.92, 0.95)$ for Régin GAC and Basic Filter in physician scheduling, Régin GAC and Basic filter achieved a significantly higher LBD which allowed them to find a solution with some objective value, where decomposition did not find any solution at all. Régin GAC tended to have a lower LBD in vaccine and rotating workforce scheduling. Surprisingly, it tended to outperform decomposition in the latter problem, despite having a lower LBD.

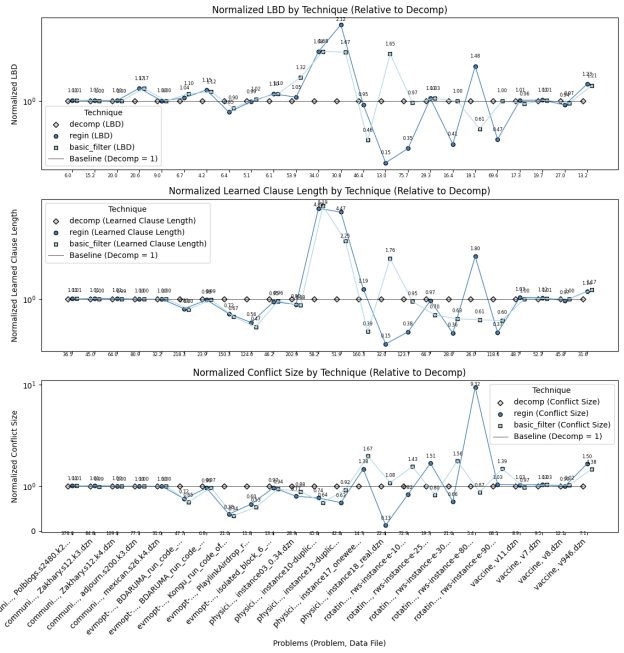

Figure 4: Benchmark of the runtime and objective value of the the two propagators with decomposition as baseline

6.3 Measuring the effect of improvements on propagation or explanation algorithms

Figure 6 shows a run of Régin GAC and Basic filter compared to decomposition, but this time using the specialized explanation algorithms. The geometric mean speedup rates are $(3.19, 2.28)$ compared to $(3.42, 1.52)$ when using the general explanations. From this it follows that Basic Filter greatly improves in performance with the new explanations. There is now only a single problem in which it is slower or equal to decomposition. On the other hand the Régin explanation and optimization saw a decrease in runtime performance, but it is still far superior than decomposition, on every single instance.

On the Minizinc challenge, the Régin GAC and Basic Filter with the specialized explanations achieve a geometric mean speedup rate of $(1.36, 1.00)$ with a success rate of $(37.5, 37.5)$ compared to 33 percent for decomposition. As with Sudoku, the new explanation algorithm benefited Basic Filter, with an increase in speedup from 0.91 to 1.00 making

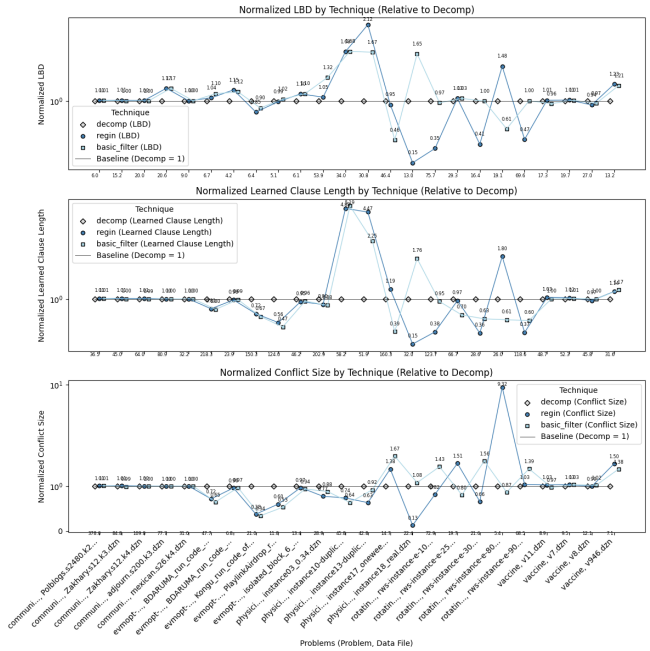

Figure 5: Benchmark of LBD, learned clause length and average conflict size

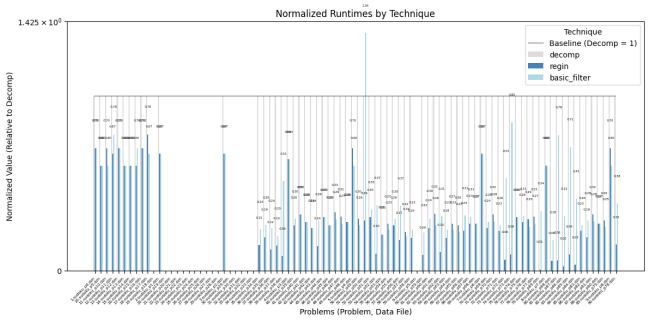

Figure 6: Sudoku normalized runtime benchmark using the spezialized explanation algorithms

it now on par with decomposition. Régin GAC saw a statistically negligible improvement of 1.36 compared to 1.34 from before.

The statistics are LBD $= (0.85, 1.0)$, LCL $= (0.91, 1.03)$, CS $= (0.93, 0.84)$ compared to LBD $= (0.90, 1.05)$, LCL $= (0.91, 0.95)$, CS $= (0.92, 0.95)$ from before. Both Régin and Basic Filter saw a decrease in relative LBD of 5%.

Overall the new explanation has benefited Basic Filter greatly. On the other hand, it has not helped Régin GAC much when it comes to runtime. A possible reason could be that increase in explanatory strength (seen in the reduction of the LBD) is offset by a the high runtime cost of computing the explanation. Alternatively, it could be the artificial 20 minute limit on the runtime of the problems: The new explanations reduced the runtime in the worst cases from before, but did barely not terminate on some problems. If the runtime is around this limit, on some runs a problems might or might not arbitrarily terminate on time, skewing the statistics. This arbitrary time restriction is, however, a part of the Minizinc challenge but the performance improvement might be more reliably estimated by using a criterion that combines the objective value with the runtime.

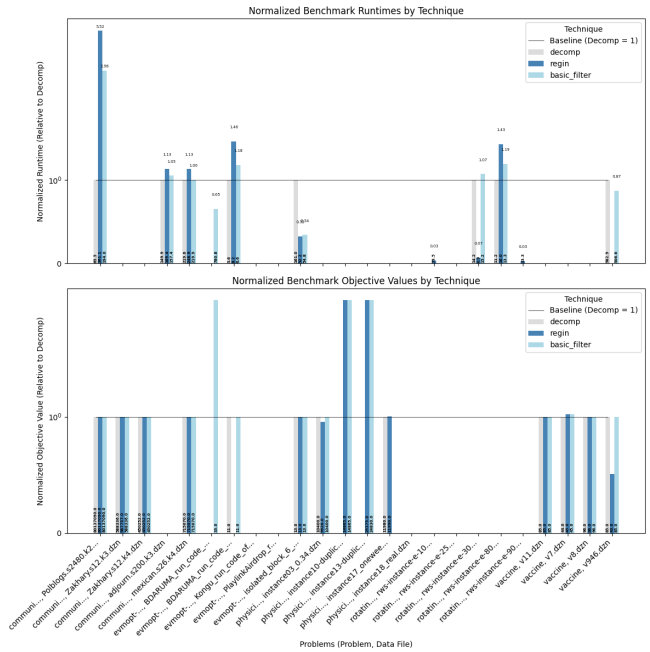

Figure 7: Minizinc challenge runtimes of Régin GAC and Basic Filter using the specialized explanations (normalized to decomposition = 1)

7 Conclusions and Future Work

This paper poses the question of whether an implementation of a propagator for the `global_cardinality` constraint in a LCG solver can be competitive against the status quo: the decomposition method. To answer the question, two propagators were implemented in the Rust-based Pumpkin LCG solver. The two algorithms (Régin GAC and basic filter) originally for a Finite Domain solver, were adapted to be used in a LCG solver by adding an algorithm algorithm to explain domain pruning or inconsistencies.

The benchmarks show that both implementations outperform decomposition in a dataset of 90 instances of Sudoku puzzles $(3.19, 2.28)$ times faster for Régin GAC and Basic Filter respectively and $(1.34, 1.0)$ times faster on the Minizinc Challenge dataset. This might challenge the new conventional wisdom in LCG that decomposition tends to beat global propagators. While the runtime and objective value measures favor Régin GAC, the LBD statistic is less conclusive.

A possible improvement to better compare the algorithms against decomposition would be to improve the explanation algorithm as mentioned in Section 5. The current implementations have a significant runtime cost which might negatively affect the performance of the SAT solver's learning and memory requirements.

8 Responsible Research

All the source code used is openly accessible at: https://github.com/spaghetti-stack/Pumpkin/tree/develop. This includes the code for the LCG solver and the algorithms implemented, as well as the code and configuration files to generate the benchmarks and create the plots. The files of the problems used for the benchmark are not in the repository, but the URLs where they are taken from are written in the experimental setup section. The Minizinc Challenge problems are properly cited the way they recommend. Meanwhile the Sudoku problems do not have a proper reference in their website, so a URL using the Wayback Machine is provided to reduce the chance of the problems becoming unavailable if the server goes down. This should make all the experiments in the paper reproducible.

References

[1] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *IJCAI International Joint Conference on Artificial Intelligence*, 2009.

[2] Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5732 LNCS, 2009.

[3] Tobias Geibinger, Lucas Kletzander, Matthias Krainz, Florian Mischek, Nysret Musliu, and Felix Winter. Physician scheduling during a pandemic. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12735 LNCS, 2021.

[4] Irit Katriel and Sven Thiel. Complete bound consistency for the global cardinality constraint. *Constraints*, 10, 2005.

[5] George Katsirelos. Nogood processing in csps, 2008.

[6] Nysret Musliu, Andreas Schutt, and Peter J. Stuckey. Solver independent rotating workforce scheduling. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10848 LNCS, 2018.

[7] Peter Nightingale. The extended global cardinality constraint: An empirical survey. *Artificial Intelligence*, 175, 2011.

[8] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4741 LNCS, 2007.

[9] Claude Guy Quimper, Peter Van Beek, Alejandro López-Ortiz, Alexander Golynski, and Sayyed Bashir Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2833, 2003.

[10] Claude Guy Quimper, Alejandro López-Ortiz, Peter Van Beek, and Alexander Golynski. Improved algorithms for the global cardinality constraint. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3258, 2004.

[11] Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for data mining and machine learning. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence, AAAI 2010*, 2010.

[12] Guillaume Rochart, Narendra Jussien, and François Laburthe. Challenging explanations for global constraints, 2003.

[13] Jean Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the National Conference on Artificial Intelligence*, volume 1, 1996.

[14] Andreas Schutt. Improving scheduling by learning, 2011.

[15] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Why cumulative decomposition is not as bad as it sounds. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5732 LNCS, 2009.

[16] Kostas Stergiou and Toby Walsh. The difference all-difference makes, 2002.

[17] Peter J. Stuckey. Lazy clause generation: Combining the power of sat and cp (and mip?) solving. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6140 LNCS, 2010.

[18] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The minizinc challenge 2008-2013, 2014.

[19] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1, 1996.