# Improving propagation of the inverse constraint in lazy clause generation solvers

**To what extent can the use of Dulmage-Mendelsohn decomposition enhance the computational efficiency of propagating the inverse constraint in LCG solvers compared to decomposition methods?**

**Grigory Prikazchikov**

**Supervisor(s): Emir Demirović, Maarten Flippo**

EEMCS, Delft University of Technology, The Netherlands

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

## Abstract

Constraint programming is a paradigm used for solving complex combinatorial problems with applications in logistics, healthcare, telecommunications, and many other fields. Among the many constraints used in constraint programming is the *inverse constraint*, necessary for matching items one to one, such as the placement of containers on ships and task scheduling. This paper presents a novel propagator for the inverse constraint, leveraging the Dulmage-Mendelsohn decomposition, implemented in the Pumpkin solver, to improve the performance of the solver. Unlike decompositions of the constraint into other simpler constraints, our approach utilizes the full bipartite graph structure of the constraint, generating stronger and more reusable explanations for unsolvable states.

Experiments were conducted on benchmark problems taken from the MiniZinc challenge, including the Time-Dependent Traveling Salesman Problem (TDTSP) and Perfect One-Factorization (P1F). These experiments demonstrated the effectiveness of the propagator. Achieving up to a 35% reduction in the time taken by the solver for some instances of TDTSP, the method also highlights limitations in the symmetric application of the inverse constraint on a single array. These findings advance our understanding of the propagation of the inverse constraint and show the potential of graph based techniques to optimize LCG solvers. Future work aims to optimize the implementation and explore its performance in relation to a broader spectrum of techniques.

## 1 Introduction

Every day millions of containers are transported on vessels around the world, using complex algorithms to optimize container placement for safe and efficient transport [10]. Constraints programming is one of the paradigms used for solving these critical challenges [10], it does this by breaking them down into smaller constraints and solving them efficiently. This approach is also used in fields such as logistics, healthcare, and telecommunications [3; 6; 10].

The problem of placing containers on vessels is solved in constraint programming by splitting it into distinct constraints based on rules, regulations, and practical considerations [10]. For safety reasons, oxidizers cannot be placed near fuels and toxic chemicals cannot be stored near food [9]. Furthermore, to maximize efficiency, containers scheduled for earlier unloading should be placed on top of those requiring later unloading, while heavier containers should be placed at the bottom and lighter ones on top [10]. Among all these constraints, and perhaps most importantly, every container must be assigned to a single spot, and every spot must contain exactly one container. This constraint, which involves matching one item in one set to exactly one item in another, is commonly referred to as the inverse constraint and is the central focus of this paper.

In recent years, there has been a tremendous amount of research conducted by various universities, open source communities, and industry giants such as Google [7], in an attempt to produce more and more efficient constraint programming solvers. The two most prevalent competing types of solvers are the satisfiability (SAT) and lazy clause generation (LCG) solvers. SAT solvers work by transforming the constraints into a Boolean satisfiability problem and then using highly optimized algorithms to solve it [12]. LCG solvers, which are the focus of this paper, implement a layer above the SAT solver where global constraints are propagated first by dedicated propagators that reduce the domains of the variables using algorithms specific to the constraint, and if the problem is unsolvable, return a no-good explaining why [12].

Despite this, LCG solvers still struggle to solve handling the inverse constraint, which is essential to solve problems containing a one-to-one mapping, such as the container placement problem. Although some solvers provide dedicated propagators to solve the inverse constraint, they decompose it into other constraints [15; 14], potentially losing information that could be learned from the graph structure in situations where the problem is found to be unsolvable.

To address this gap, this paper presents the implementation of an inverse propagator for the Pumpkin Solver, developed in Rust by the ConSol lab at TU Delft [2]. The proposed propagator uses Dulmage-Mendelsohn canonical decomposition, to explain why specific states are unsolvable. The method is tested against the standard MiniZinc decomposition of the inverse constraint, on problems taken from the MiniZinc challenge. Our results demonstrate the effectiveness of this approach in improving solver performance.

This method takes into account complex structures present in the inverse constraint without decomposing it into other constraints. This allows for potentially more useful no-goods, which would lead to a more efficient solver.

The experiments show that our propagator outperformed the MiniZinc decomposition in the time-dependent traveling salesman problem (TDTSP), where its utilization of graph decomposition leads to stronger explanations and improved solver efficiency. However, it underperformed in the perfect one-factorization problem (P1F), where the symmetric application of the inverse constraint to an array and itself introduced unnecessary abstraction and complexity during the Dulmage-Mendelsohn decomposition, limiting its effectiveness.

This paper contributes a novel implementation of an inverse constraint propagator, utilizing the Dulmage-Mendelsohn decomposition technique to improve the strength and reusability of explanations of unsolvable states of the problem to improve solver efficiency. Unlike previous implementations, this propagator utilizes graph decomposition of the whole problem without decomposition, capturing more of its state. This is particularly impactful for problems like TDTSP, in some cases achieving as much as a 35 percent reduction in the time spent solving the problem, where the decomposition accurately models the state of the problem. However, for problems such as P1F, where the inverse is applied symmetrically to a single array, the MiniZinc decomposition or other decompositions of the constraint are more effective.

This paper is structured as follows. Section 2 introduces concepts used throughout this paper. Section 3 defines the problem that our research aims to solve. Section 4 reviews related work. Section 5 details the proposed algorithm for propagation of the inverse constraint. Section 6 presents the experimental results, and Section 7 discusses ethical considerations. Finally, Section 8 discusses the conclusions and recommendations for future work.

## 2 Preliminaries

This section introduces the main concepts used throughout this paper, such as the inverse constraint, MiniZinc decomposition, lazy clause generation, and Dulmage-Mendelsohn decomposition.

### 2.1 Inverse Constraint

The inverse constraint commonly appears in different computational problems in various fields. For example, in logistics, it is necessary when placing containers on a vessel to ensure that each container receives exactly one spot and each spot gets exactly one container [10]. It can also be used in computer task scheduling problems, where each task needs exactly one allocated resource or time slot and vice versa [12]. Another occurrence of the inverse constraint is in education, where during an exam week, exams from the same course must be in different time slots to allow students to attend all their exams and resists, and during the same time slot exams from different courses must be in different rooms [8].

The inverse constraint can be formally described as a constraint that requires the elements of two sets to be matched exactly one to one, this is also known as a perfect matching [4]. This can be expressed as two arrays X and Y where the value of one matches the index of the other and vice versa, so if $X[i] = j$ then $Y[j] = i$ [4]. The constraint can also be seen as a graph containing vertices $X_n$ and $Yn$ with a single edge connecting each of the vertices in $X$ to a single vertex in $Y$ [4]. For example, in the following arrays, $x_1$ is matched with $y_5$ because $X[1] = 5$ and $Y[5] = 1$. This also holds for $x_2, x_3, x_4, x_5$ and $y_4, y_3, y_2, y_1$.

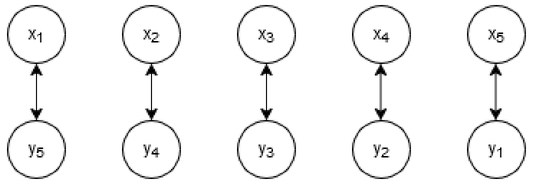

Figure 1: Graph satisfying the inverse constraint

This same relationship can be seen in figure 1. Here, $x_1$ is matched with $y_5$, $x_2$ with $y_4$, $x_3$ with $y_3$, $x_4$ with $y_2$, and $x_5$ with $y_1$.

$$\begin{aligned} X &= [5,4,3,2,1] \\ Y &= [5,4,3,2,1] \end{aligned} \tag{1}$$

### 2.2 MiniZinc Decomposition

MiniZinc provides an interface to communicate with various constraints programming solvers[16]. It does so by providing a language to describe problems and their constraints. MiniZinc can also compile problems into FlatZinc files, which contain exact instructions on what variables should be initialized at and what constraints are applied to them[16]. It does this by first checking which constraints the solver supports, and if the solver does not support a constraint, it decomposes it [16].

In the case of the inverse constraint, MiniZinc decomposes it as follows, for the two arrays $X$ and $Y$ it states that $x_i$ indicates where $i$ is in the array $Y$ and $y_j$ indicates where $j$ is in the array $X$. This constraint is then applied for each element in $X$ and $Y$ [16].

### 2.3 Lazy Clause Generation

Lazy clause generation (LCG) solvers work by combining Satisfiability (SAT) and Finite Domain (FD) solvers, to lazily generate clauses throughout the FD solving process that explain the problem in Boolean logic [12]. This allows the SAT solver to find the solution and take advantage of no-goods that have been generated. leading to a drastic reduction in search space and an increase in the performance of these solvers [12].

LCG solvers have separate propagators for different global constraints. These propagators work by first decreasing the size of the variables domains and giving a Boolean explanation for why this reduction was done[12]. For example, there is have an equality constraint between variables x and y with the following domains:

$$\begin{aligned} D(x) &= [1] \\ D(y) &= [1,2] \end{aligned} \tag{2}$$

It is clear that $y$ cannot be two since $x$ and $y$ have to be equal and $x$ cannot be two. This allows for the removal of two from the domain of $y$ with the explanation of $y \neq 2$ because $x \neq 2$. In the future, this explanation allows us to remove two from the domain of $y$ if $x$ is not equal to two. To be able to do this, the explanations provided must be self-sufficient. So in any situation where $x \neq 2$ then $y \neq 2$ must always hold [12].

The removal of all inconsistent values from the domains of all of the variables is referred to as achieving hyper-arc consistency. Or more formally: a constraint C is considered hyper-arc consistent if, for every variable $x_i$ in the constraint, every value in the domain of $x_i$ has a consistent combination of values from the domains of the other variables that satisfies the constraint[12]. Hyper-arc consistency is desirable, but not always achievable due to high computational costs for some constraints [12].

Additionally, propagators in an LCG solver verify if the current domains of variables allow for a feasible solution. If a solution can no longer exist, then the propagator returns a no-good. This no-good is a description of what in the problem leads to it being unsolvable [12]. For example, if we have an equality constraint on the variables $x$ and $y$ with the following domains:

$$\begin{aligned} x &\leq 2 \\ y &\geq 4 \end{aligned} \tag{3}$$

With these domains a solution can clearly never exist, as $x < y$ so they can never be equal. One of the no-goods that could be provided for this state is $x \leq 2 \cap y \geq 4$. No-goods are used to prune the search tree by stopping as soon as a known no-good is found to be true. So in this case, if x is less than two and y is greater than 4 then there is no need to search any further.

### 2.4 Dulmage-Mendelsohn Decomposition

A Dulmage-Mendelsohn decomposition is a decomposition of a bipartite graph into 3 sets, D: under-constrained, C: well-constrained, and A: over-constrained [4]. The under-constrained set is the set where there are more nodes in $X$ than in $Y$, or in Figure 2, $D1$ and $A2$. The over-constrained set is the opposite, where there are more nodes in $Y$ than $X$, or in Figure 2, $A1$ and $D2$ [5].

This decomposition is found by first finding a maximum matching, which pairs the maximum number of nodes in $X$ with nodes in $Y$. This is illustrated in Figure 2 as the black and blue edges. subsequently, a search is performed, starting nodes not covered by the maximum matching ($x_1$ and $y_6$), using edges alternating between matched and unmatched ones. Every node with an even index on the path is placed in set $D$ and every node with an odd index is placed in set $A$. This can be seen in figure 2 where the paths starting in $x_1$ and $y_6$ are visible, alternating between unmatched (red) edges, and matched (blue) edges [5].

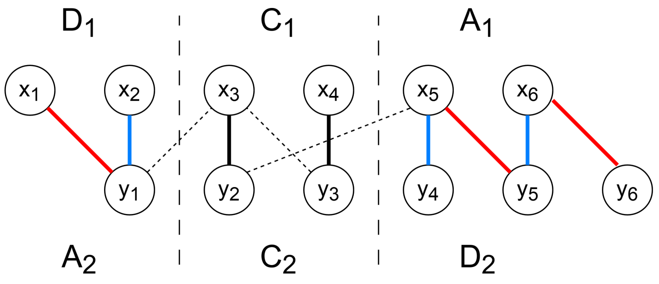

Figure 2: Odd and even edges in a Dulmage-Mendelsohn decomposition

## 3 Problem description

Although various algorithms for the propogation of the inverse constraint exist, none of them take into account the full graph structure of the problem. Parts of the problem, such as achieving hyper-arc consistency have been proven to be solvable in linear time [4; 15]. Other parts, such as the generation of no-goods, are more complex, and existing solutions decompose the constraint, potentially losing important information [12].

in light of this, this paper aims to provide an algorithm for the generations of explanations for insolvable states of the problem. The challenge lies in using the full structure of the bipartite graph that represents the inverse constraint to generate stronger and more reusable explanations. By avoiding decomposition into other constraints, this approach would capture the complex relationships in the graph, offering insights that would otherwise be ignored.

## 4 Related Work

In his PhD thesis Improving Scheduling by Learning, Andreas Schutt discusses the need for strong and reusable explanations in LCG solvers [12]. A strong explanation is defined as a more general explanation having fewer variables or larger domains for those variables. The reusability of an explanation is the probability that it will recur in future states. Stronger and more reusable explanations decrease the size of the domain of variables, or terminate the search early in the case of a no-good more effectively, leading to better solver performance [12].

In his paper and later in his PhD thesis, Radosaw Cymer describes the potential use of Dulmage Mendelsohn canonical decomposition, as well as other graph decomposition techniques for the propagation of various graph constraints in constraint programming solvers [4; 5]. Radosław proposes to decompose the inverse constraint into two all different constraints on both sets of variables, then pruning the variables domains by removing inconsistent values [4]. This is done by removing values from the domains of variables that would indicate a unidirectional edge. For example, if the domain of $x_1$ contains 2 but $y_2$ does not contain 1 then 2 can be removed from the domain of $x_1$ [5].

One of the metrics for measuring clause reusability is literal block distance (LBD), this metric was proposed by Gilles Audemard and Laurent Simon as a method to improve the efficiency of SAT solvers [1]. Literal block distance uses the decision levels of the literals in a clause to predict how reusable the clause is. LBD is calculated by counting the number of unique decision levels of literals present in a clause, so if the literals were assigned at the decision levels $\{1, 2, 1, 3, 2, 3\}$, then the LBD is equal to three, corresponding to the decision levels $\{1, 2, 3\}$ [1]. The SAT solver developed by Gilles Audemard and Laurent Simon aggressively removed clauses with high LBD, leading to a significant improvement in performance, especially on industrial benchmarks [1].

An implementation of the inverse propagator can be found in the Choco-Solver [14], which is an open source Java library for Constraint Programming. This propagator achieves hyper-arc consistency by eliminating illegal values from the variables domains in the two arrays in the same way as proposed by Radosław Cymer [4]. This propagator does not verify feasibility and instead waits for the final assignment of variables to see if it satisfies the constraint [14].

Another implementation of the inverse propagator can be found in the Chuffed solver [15], which is an LCG solver written in C++ by the Department of Computing and Information Systems University of Melbourne, Australia. This solver implements the same algorithm as described in the paper and thesis of Radosław Cymer [4; 5]. The propagation of the all different constraint is done using Tarjan's strongly connected component algorithm [15]. This allows the propagator to find inconsistencies and provide explanations for early termination based on leaks in the assignment of different values in each of the arrays [4].

## 5 Inverse constraint propagation

In our propagator hyper-arc consistency is achieved in linear time using the constraint that for the two sets $X$ and $Y$, if $j \in D(x_i)$ then $i \in D(y_j)$. This allows us to prune the domains of $X$ and $Y$ as follows, if $j$ is in the domain of $x_i$ and $i$ is not in the domain of $y_j$, then $j$ can be removed from the domain of $x_i$. Tthe inverse of this must then also be done, if $i$ is in the domain of $y_i$ and $j$ is not in the domain of $x_i$, then $i$ can be removed from the domain of $y_i$. An example of this propagation can be seen in Figure 3, where $1 \in D(x_2)$ but $2 \notin D(y_1)$, therefore, there can never be a match between $x_2$ and $y_1$ and one can be removed from $D(x_2)$. The same reasoning can be used to remove one from $y_2$.

$$\begin{aligned} D(x_1) &= \{1\} \\ D(x_2) &= \{1,2\} \\ D(y_1) &= \{1\} \\ D(y_2) &= \{1,2\} \end{aligned}$$

$$\begin{aligned} D(x_1) &= \{1\} \\ D(x_2) &= \{2\} \\ D(y_1) &= \{1\} \\ D(y_2) &= \{2\} \end{aligned}$$

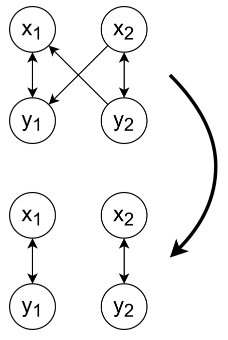

Figure 3: Example propagation of inverse constraint.

After achieving hyper-arc consistency, the feasibility of a solution is determined by running a maximum matching algorithm on the bipartite graph of the two sets. If this matching contains all the nodes of the graph it is described as a perfect matching and a solution still exists. If not, an explanation of the no-good is calculated by performing a Dulmage-Mendelsohn decomposition of the graph and giving a description of the domains of the over-constrained variables ($D_1$ or $D_2$) in the graph. The soundness of this no-good can be seen in Figure 4, the variables in $D_1$ are only connected to $A_2$ and since $D_1$ has more variables than $A_2$, there can never be a perfect matching in the graph with the variables in $D_1$ having the domains they have. The same is true for the variables in $D_2$.

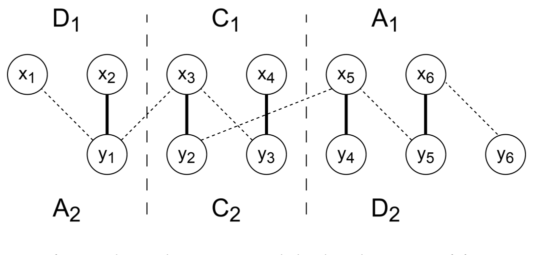

Figure 4: Dulmage Mendelsohn decomposition

In the example in Figure 4 the no-goods generated from $D1$ and $D2$ would be:

$$\begin{gathered} x_1 = 1 \cap x_2 = 1 \\ or \\ y_4 = 5 \cap y_5 \geq 5 \cap y_5 \leq 6 \cap y_6 = 6 \end{gathered} \tag{4}$$

As discussed in Andreas Schutt's PhD thesis, no-goods should aim to be strong and reusable [5]. This requires the domains of the variables to be as large as possible with as few variables as possible, so that the no-good described can apply to as many scenarios as possible. In this example, it is clear that the first no-good is better than the second, since it has two variables with a domain of size one, and the second has two variables with a domain of size one and a variable with a domain of size two. However, the decision on which no-good to use is not always as clear. To solve this, the following heuristics have been tested:

1. Use the side with fewer variables.
2. Use the side with the highest average domain size.
3. Use the side with the lowest LBD.

The first heuristic minimizes the number of variables, which generalizes and strengthens the explanation. The second favors larger domains, also generalizing the no-good since any domain smaller than the domains of the variables in the no-good would also satisfy it allowing the search to terminate. The third uses literal block distance, which has been shown to achieve more reusable clauses in SAT solvers [1].

## 6 Experimental Setup and Results

This section starts by explaining the setup of the experiments. Then describe the metrics used to measure their success or failure. Followed by a subsection for each of the two selected problems, the Time-Dependent Traveling Salesman problem (TDTSP) and the perfect one-factorization problem (P1F). These subsections start by explaining the problem, then the use of the inverse constraint in the problem, followed by the results, and lastly an explanation of them.

The algorithms and heuristics described in the previous section were implemented in Rust in the Pumpkin LCG solver. The solver was then compiled for each of the heuristics, and all instances of the problems were compiled using MiniZinc, with our inverse propagator and a decomposition of the inverse constraint, into FlatZinc, which provides specific instructions to the solver on how to solve the problem. The problems were then run on a Debian PC with an AMD Ryzen 5 5600G and 16GB of RAM, the full specs of which can be found in Appendix A.

The two metrics selected to determine the success of the experiments were the number of propagations and the time spent in the solver. The number of propagations measures the effectiveness of each propagation and, by extension, the strength and reusability of the explanations and no-goods. A lower number of propagations indicates that fewer steps needed to be taken to find the solution to the problem. The time spent in the solver shows how computationally efficient the propagations were. When very few propagations are required and the explanations and no-goods are incredibly strong and reusable, but their calculation is very expensive, the performance of the solver suffers.

### 6.1 Time-dependent traveling salesman problem

TDTSP is a variation of the classical traveling salesman problem (TSP). TSP is a graph optimization problem where a traveling salesman must visit every city (node) by traveling on roads (edges) between them that take a certain amount of time (weight) and return to the starting city while taking the shortest amount of time [13]. The time-dependent variation of this problem makes the problem more complex by changing the time it takes to travel a road, depending on the time of travel. This variation of the problem is closer to real world scenarios where travel times depend on various conditions such as traffic, road closures, or time-of-day effects [13].

Inverse constraint is applied in this problem to the previous and next arrays. This ensures that the backward and forward orders of the nodes are the same. The previous array states, for each city, which city is visited immediately before it. Or if $previous[j] = i$, it means that before visiting node $j$, the path came from node $i$. The next array is the inverse of this and shows, for each city, which city is visited immediately after it. This can be expressed as $next[j] = i$, which means that after visiting node $j$, the path leads directly to node $i$. Here, the inverse constraint ensures that there is a complete cycle in the result.

Four instances of the TDTSP problem were selected, these instances were selected due to their relatively low computational times and hardware and time limitations. The four instances contained ten nodes and used scenarios 24 starting at $t = 10$, scenario 34 starting at $t = 0$, and scenario 42 starting at $t = 0$ and $t = 10$.

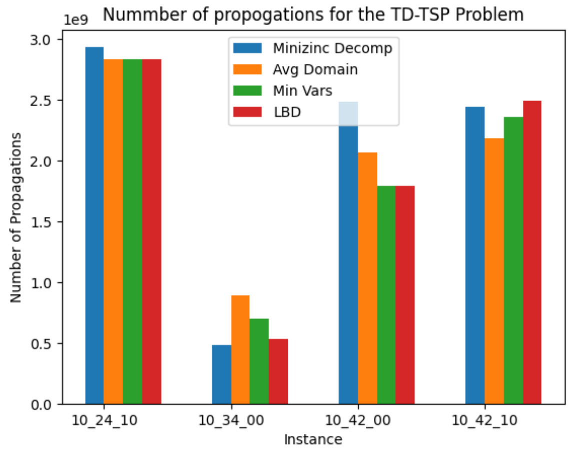

Figure 5: Number of propagations for TDTSP

The results in Figure 5 show an overall decrease in the number of propagations needed to solve the problem in our propagator over the MiniZinc decomposition. The most significant improvements are made in Scenario 42, start time $t = 0$. However, the number of propagations was higher for all heuristics in Scenario 34, start time $t = 0$. While this shows that our propagator improves upon the MiniZinc decomposition, there is some luck involved in finding the right solution efficiently. This luck or randomness occurs because some set of explanations or no-goods or a combinations of both might drastically reduce the search space and simplify the problem throughout the search. This exact combination cannot be known until a solution is found as these problems are NP-Hard.

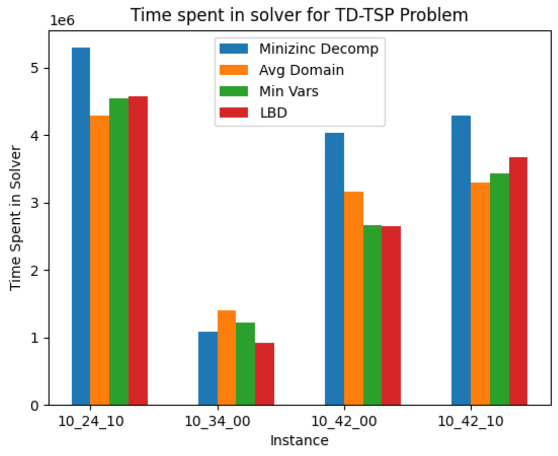

Figure 6: Time spent in solver for TDTSP

The results in Figure 6 mostly mirror the results in Figure 5, however, they show an even greater improvement in performance. This shows that for this problem not only were the explanations and no-goods generated by our propagator are stronger and more reusable, but their calculation is also less computationally expensive. This is particularly surprising because the implementation was not fully optimized and shows that a more optimized implementation of the algorithms could provide an even greater performance increase.

To assess which heuristic performed best, the results were also normalized across instances of the problem using the following formula:

$$\text{Normalized Sum}_{p,s} = \frac{\sum_{i \in \text{propagator } p} \text{Value}_{i,s}}{\sum_{j \in \text{All propagators}} \text{Value}_{j,s}}$$

This formula was used to take into account the varying difficulty of the problem instances and capture relative performance increases. Due to both metrics showing similar results, only the time spent in the solver is shown here in Figure 7 however, the graph for the number of propagations is available in Appendix B.2.

The normalized results in Figure 7 show a very small improvement in the LBD heuristic over the other two, which would indicate that there is little difference between them. However, the propagator using the LBD heuristic was the only one to outperform the decomposition in all instances of the problem, which leads to the conclusion that even if only marginally, the LBD heuristic was most effective.

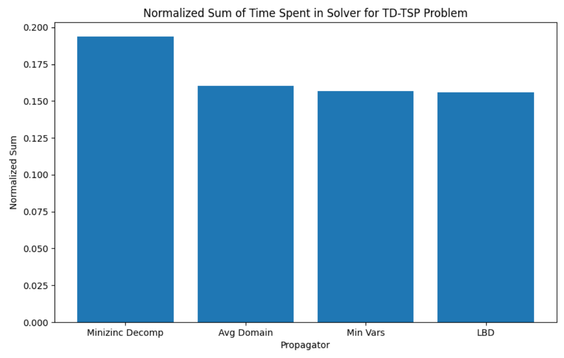

Figure 7: Normalized sum of time spent in solver for TDTSP

## 6.2 Perfect one-factorization

P1F is a combinatorial graph theory problem where the goal is to decompose the edges of a complete graph into disjoint one-factors [11]. A one-factor is a set of edges that pair up the nodes in the graph so that each node is matched with exactly one other node[11]. This can also be expressed as all nodes having a degree of one. A P1F is a special case of a one-factorization where any pair of one-factors forms a Hamiltonian cycle, which is a cycle in a graph that visits every node exactly once and returns to the starting node [11].

The inverse constraint is used in this problem to ensure that the sets that the edges get factorized into are actually one-factors. For a complete graph of size $n$ is done by creating an array $X$ of size $n$ that for each index contains the node with which that node is matched, so for example, the array $[1, 3, 2, 4]$ would show the pairs $(1, 3)$ and $(2, 4)$. The inverse constraint is then applied to the array $X$ to itself, ensuring that if node $i$ is matched with $j$, then $j$ is matched with $i$.

For our experiments, the instances that were selected were $n = 6, n = 8, n = 10, n = 12$. Due to the number of propagations and time spent in solver growing exponentially for each instance, the results were divided by the maximum result of the instance to graph them. These results can be seen in Figures 8 and 9.

Figure 8 clearly shows that the MiniZinc decomposition outperforms our propagator in the number of propagations, for all heuristics, with negligible, if any, difference between the heuristics.

Figure 9 mirrors the results of Figure 8 however, with an even greater decrease in performance. This can be explained by the symmetric application of the inverse constraint to an array and itself. As discussed in Section 2.2, MiniZinc decomposes the problem into constraints that state that, for arrays $X$ and $Y$, each $x_i$ in $X$ indicates what index in $Y$ is taken by $i$, the same constraint is applied to $Y$. In a problem such as TDTSP where the inverse constraint is applied on two different arrays of length $n$ this leads to $2n$ constraints in the decomposition. However, when the inverse constraint is applied symmetrically, as in P1F, this only leads to $n$ constraints, making the decomposition much more effective.

Our propagator creates a bipartite graph with the two sets of variables provided to it and performs a Dulmage-

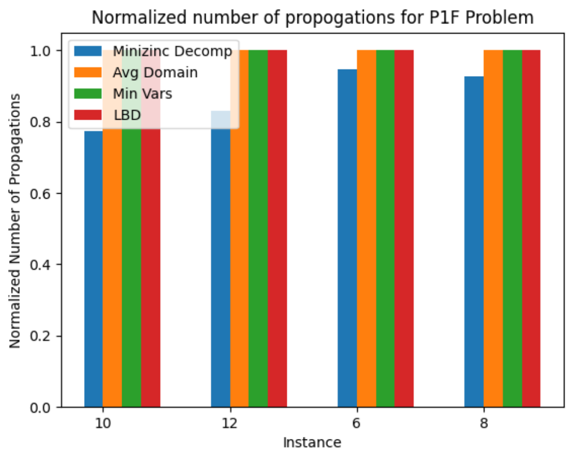

Figure 8: Normalized number of propagations for P1F problem

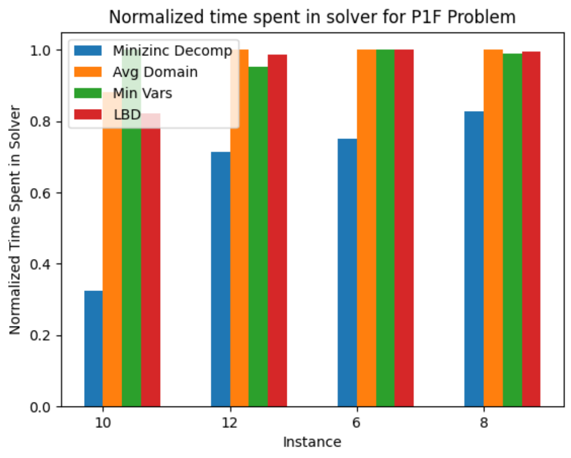

Figure 9: Normalized time spent in solver for P1F problem

Mendelsohn decomposition of the graph. However, in this case, since both arrays are the same, this is an unnecessary abstraction that adds complexity to the problem. This can be seen in Figure 10.

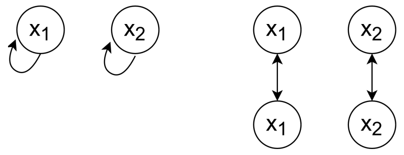

Figure 10: Problem in decomposition and our propagator

Figure 10 shows how our algorithm, by creating a bipartite graph, doubles the number of nodes for which it generates an explanation. The Dulmage-Mendelsohn decomposition is then performed on a graph that does not exist in the problem, greatly reducing the value of information gained from it. This shows why our proposed propagator is ineffective for problems such as P1F where the inverse constraint is applied symmetrically to a single array.

## 7 Responsible Research

In this section, we discuss ethical considerations, the reproducibility of results, and the FAIR data principles related to this paper.

### 7.1 Ethical considerations

The research done in this paper does not pose ethical considerations in and of itself. No personal data are used during any of the research or experiments and experiments are performed on fictitious and purely theoretical problems. However, there may be some ethical considerations in the future use of the algorithms provided. The inverse constraint can be used in resource allocation problems, so care should be taken when using these or similar algorithms to allocate vital resources to human beings.

### 7.2 Reproducibility

The code used in this paper is all open source and is available on GitHub. The problems used to evaluate the algorithm are also open source and are available on the MiniZinc website. The exact time that the algorithms take to run is dependent on the exact hardware used, the hardware used in our experiments is listed in the Appendix A. The number of decisions and other statistics would remain the same regardless of the hardware used.

### 7.3 FAIR principles

The data and methodology used in this article were performed according to the FAIR principles as follows:

- **Findable**: This paper along with the results in Appendix B.1 is available in the TU Delft repository, which provides search functionality to easily locate the data. The code used is located on GitHub as a fork of the Pumpkin repository, which allows it to be easily found.
- **Accessible**: The TU Delft repository is freely accessible to anyone, as is the GitHub repository.
- **Interoperable**: The data is provided in Appendix B.1 as a table and are available in the GitHub repository as a CSV, allowing them to be easily interoperable between platforms and analytical software. The code is written in Rust, allowing it to be compiled on any operating system and platform.
- **Reusable**: The data can be used to compare to other algorithms for propagating the inverse constraint, as long as they are implemented in the Pumpkin solver. The code can also be forked, changed, or used in other experiments as it is available under an Apache and MIT license.

## 8 Conclusions and Future Work

This paper explored the implementation of a novel propagator for the inverse constraint in lazy clause generation (LCG) solvers. This was done using Dulmage-Mendelsohn decomposition, a graphical analytical technique, to provide stronger and more reusable explanations for unsolvable states of the problem. With the aim of addressing issues with other propagators that decompose the constraint into simpler ones.

From the results of our experiments we found up to a 32% reduction in computational costs associated with solving problems such as the time-dependent traveling salesman problem (TDTSP). Unfortunately, our approach was less effective in cases where the inverse constraint is applied symmetrically to a single array, like in the perfect one-factorization (P1F) problem, where it was outperformed by the MiniZinc decomposition.

Our research shows how graph based techniques can improve the performance of inverse constraint propagation in LCG solvers. Our findings also present opportunities for further research, which include further optimization of the implementation of our propagator in the Pumpkin solver, the comparison of our propagator against other decompositions of the constraint, and the use of Dulmage-Mendelsohn decomposition for propagating other constraints. If our propagator were combined with another algorithm for dealing with the symmetric application of the inverse constraint, it would greatly improve its ability to solve complex problems.

In conclusion, this study advances the understanding of inverse constraint propagation, providing both practical insights and a foundation for further innovation in lazy clause generation solvers.

## References

[1] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Twenty-first international joint conference on artificial intelligence*. Citeseer, 2009.

[2] ConSol-Lab. Pumpkin. https://github.com/ConSol-Lab/Pumpkin, 2024. Accessed: 2024-11-16.

[3] IBM Corporation. Ibm ilog cplex optimization studio: Cp optimizer, 2023. Accessed: 2025-01-26.

[4] Radosław Cymer. Dulmage-mendelsohn canonical decomposition as a generic pruning technique. *Constraints*, 17(3):234–272, 2012.

[5] Radosław Cymer. *Applications of Matching Theory in Constraint Programming*. PhD thesis, University of Hannover, 2013.

[6] Vu Duong, Stefan Petrovic, and Ana Viana. Home healthcare routing and scheduling: A stochastic optimization approach. In *Proceedings of the International Conference on Practice and Theory of Automated Timetabling (PATAT)*, pages 115–130, 2014.

[7] Google. Or-tools: Google's operations research tools. https://github.com/google/or-tools, 2025. Accessed: 2025-01-19.

[8] Burairah Hussin, Abd. Samad Hasan Basari, Abdul Samad Bin Shibghatullah, Siti Azirah Asmai, and Norwahida Syazwani Othman. Exam timetabling using graph colouring approach. *2011 IEEE Conference on Open Systems*, pages 133–138, 2011.

[9] International Maritime Organization. Code of Safe Practice for Cargo Stowage and Securing (CSS Code), 2023. Accessed: 2025-01-19.

[10] Byung Kwon Lee and Joyce M. W. Low. A constraint programming approach to capacity planning in container vessels. *Maritime Economics & Logistics*, 24(2):415–438, June 2022.

[11] Alexander Rosa. Perfect 1-factorizations. *Mathematica Slovaca*, 69(3):479–496, 2019.

[12] Andreas Schutt. *Improving Scheduling by Learning*. PhD thesis, The University of Melbourne, Melbourne, Australia, 2011.

[13] Mohit Tawarmalani and Nikolaos V. Sahinidis. *Time-dependent traveling salesman problem*, pages 2619–2625. Springer US, Boston, MA, 2001.

[14] Choco Team. Choco solver: An open-source java library for constraint satisfaction problems. https://github.com/chocoteam/choco-solver, 2025. Accessed: 2025-01-19.

[15] Chuffed Team. Chuffed: A lazy clause generation solver. https://github.com/chuffed/chuffed, 2025. Accessed: 2025-01-19.

[16] MiniZinc Development Team. Minizinc: A free and open-source constraint modeling language, 2025. Accessed: 2025-01-25.

## A Experimental desktop specifications

Listing 1: lshw Output

```
description: Desktop Computer
product: System Product Name (SKU)
vendor: ASUS
version: System Version
serial: System Serial Number
width: 64 bits
capabilities: smbios-3.3.0 dmi-3.3.0 smp vsyscall32
*-core
   description: Motherboard
   product: PRIME B550M-K
   vendor: ASUSTeK COMPUTER INC.
   physical id: 0
   version: Rev X.0x
   serial: 210280933501759
   slot: Default string
 *-memory
      description: System Memory
      physical id: 2b
      slot: System board or motherboard
      size: 16GiB
    *-bank:0
         description: DIMM DDR4 Synchronous Unbuffered (Unregistered) 3200 MHz (0.3 ns)
         product: CT16G4DFRA32A.M16FR
         vendor: CRUCIAL
         physical id: 1
         serial: E572949B
         slot: DIMM_A2
         size: 16GiB
         width: 64 bits
         clock: 3200MHz (0.3ns)
 *-cache:0
      description: L1 cache
      physical id: 2d
      slot: L1 - Cache
      size: 384KiB
      capacity: 384KiB
      clock: 1GHz (1.0ns)
      capabilities: pipeline-burst internal write-back unified
      configuration: level=1
 *-cache:1
      description: L2 cache
      physical id: 2e
      slot: L2 - Cache
      size: 3MiB
      capacity: 3MiB
      clock: 1GHz (1.0ns)
      capabilities: pipeline-burst internal write-back unified
      configuration: level=2
 *-cache:2
      description: L3 cache
      physical id: 2f
      slot: L3 - Cache
      size: 16MiB
      capacity: 16MiB
      clock: 1GHz (1.0ns)
      capabilities: pipeline-burst internal write-back unified
      configuration: level=3
 *-cpu
      description: CPU
      product: AMD Ryzen 5 5600G with Radeon Graphics
      vendor: Advanced Micro Devices [AMD]
```

```
        physical id: 30
        bus info: cpu@0
        version: 25.80.0
        serial: Unknown
        slot: AM4
        size: 3552MHz
        capacity: 4463MHz
        width: 64 bits
        clock: 100MHz
        configuration: cores=6 enabledcores=6 microcode=173015051 threads=12
```

## B Experimental results

### B.1 Raw results

Table 1: Experimental results

| type | problem | instance | NumDecisions | NumConflicts | NumRestarts | NumPropagations | TimeSpentInSolver |
|---|---|---|---|---|---|---|---|
| avgdominv | p1f | 10 | 175206 | 30848 | 6 | 21475349 | 39236 |
| avgdominv | p1f | 12 | 11715188 | 10255366 | 35 | 8416269559 | 15634823 |
| avgdominv | p1f | 6 | 356 | 91 | 0 | 9109 | 4 |
| avgdominv | p1f | 8 | 14966 | 749 | 0 | 249781 | 209 |
| avgdominv | tdtsp | 10_24_10 | 7362925 | 1143847 | 49 | 2832527362 | 4288757 |
| avgdominv | tdtsp | 10_34_00 | 3337660 | 581574 | 25 | 894740329 | 1405079 |
| avgdominv | tdtsp | 10_42_00 | 6217129 | 932087 | 59 | 2067303830 | 3170728 |
| avgdominv | tdtsp | 10_42_10 | 6186379 | 1041249 | 22 | 2186286848 | 3299496 |
| ctrl | p1f | 10 | 196242 | 24244 | 4 | 16589544 | 14375 |
| ctrl | p1f | 12 | 11612622 | 9882832 | 102 | 6979120007 | 11173866 |
| ctrl | p1f | 6 | 308 | 79 | 0 | 8628 | 3 |
| ctrl | p1f | 8 | 14270 | 670 | 0 | 231437 | 173 |
| ctrl | tdtsp | 10_24_10 | 7709638 | 1156351 | 45 | 2932894917 | 5297278 |
| ctrl | tdtsp | 10_34_00 | 2777004 | 480196 | 41 | 487365941 | 1079534 |
| ctrl | tdtsp | 10_42_00 | 6866050 | 1022243 | 59 | 2485503909 | 4037964 |
| ctrl | tdtsp | 10_42_10 | 6625821 | 1051905 | 36 | 2442589236 | 4291880 |
| lbdinv | p1f | 10 | 175206 | 30848 | 6 | 21475349 | 36599 |
| lbdinv | p1f | 12 | 11715188 | 10255366 | 35 | 8416269559 | 15443345 |
| lbdinv | p1f | 6 | 356 | 91 | 0 | 9109 | 4 |
| lbdinv | p1f | 8 | 14966 | 749 | 0 | 249781 | 208 |
| lbdinv | tdtsp | 10_24_10 | 7654440 | 1198115 | 53 | 2836838133 | 4570620 |
| lbdinv | tdtsp | 10_34_00 | 2846730 | 627950 | 15 | 533195000 | 917656 |
| lbdinv | tdtsp | 10_42_00 | 5703024 | 868249 | 49 | 1789854975 | 2650816 |
| lbdinv | tdtsp | 10_42_10 | 6503984 | 1070720 | 54 | 2492666601 | 3680679 |
| minleninv | p1f | 10 | 175206 | 30848 | 6 | 21475349 | 44482 |
| minleninv | p1f | 12 | 11715188 | 10255366 | 35 | 8416269559 | 14868888 |
| minleninv | p1f | 6 | 356 | 91 | 0 | 9109 | 4 |
| minleninv | p1f | 8 | 14966 | 749 | 0 | 249781 | 207 |
| minleninv | tdtsp | 10_24_10 | 7654440 | 1198115 | 53 | 2836838133 | 4547447 |
| minleninv | tdtsp | 10_34_00 | 2985776 | 552182 | 18 | 698816547 | 1228525 |
| minleninv | tdtsp | 10_42_00 | 5591312 | 862345 | 48 | 1788949450 | 2670443 |
| minleninv | tdtsp | 10_42_10 | 6444058 | 1050404 | 43 | 2358202916 | 3437833 |

## B.2 Normalized number of propogations TDTSP

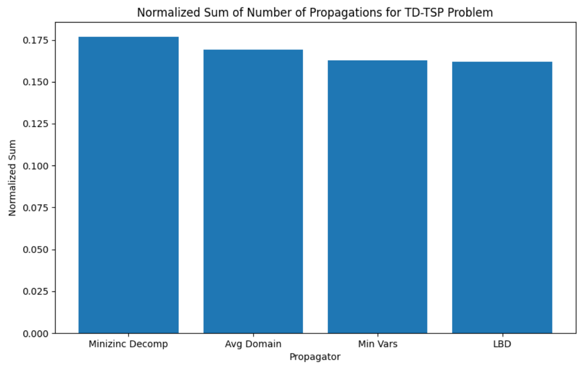

Figure 11: Normalized sum of number of propagations for TDTSP