Evaluating the Impact of Explanations on the Performance of an Edge-Finding Propagator

Radu Andrei Vasile[1]
Supervisor(s): Dr. Emir Demirović[1]**, Imko Marijnissen**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

Evaluating the Impact of Explanations on the Performance of an Edge-Finding Propagator

Radu Andrei Vasile
Delft University of Technology

Abstract

Explanations have been shown to significantly increase the performance of propagators, when applied to solvers that make use of Lazy Clause Generation. However, to date, there has been little work in exploring explanations for the disjunctive constraint and how they perform compared to simple propagation making use of Naive Explanations. I address this gap by considering an Edge-Finding propagation algorithm for the disjunctive and evaluating the performance of three variants of a solver on job-shop problems, each of them using different explanation techniques, which have been adapted from previous work. I also provide an algorithm serving as an extension to Edge-Finding, which can be used to generate explanations. Then, I compare all approaches in relation to a baseline consisting of a solver that has no support for the disjunctive and uses decomposition. Through experiments, I demonstrate that good explanations provide a significant improvement in terms of recorded metrics compared to the naive method, even if generating them requires additional computational overhead.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Mathematics of computing → Solvers; Theory of computation → Backtracking

Keywords and phrases Disjunctive, Explanations, Edge-Finding

1 Introduction

Constraint Programming (CP) is a paradigm for solving combinatorial optimisation problems by enforcing logical constraints on possible values a variable can take. One common practical representation of such problems arises in scheduling, for instance, assigning jobs to machines or creating timetables for classes or employees [10]. CP aims to solve these problems by enforcing constraints on variables and then using search algorithms to rule out infeasible schedules. This can be achieved through specialised algorithms, called propagators, that apply logical reasoning to identify infeasible solutions and reduce the search space.

One notable recent advancement in this field is the idea of Lazy Clause Generation, currently explored in the context of optimising propagation [15, 8]. Its goal is to extract the reasoning used by the propagator to rule out infeasible solutions, referred to as an explanation, and use it to generate clauses. These clauses can determine whether the assignments directly responsible for conflicts happened at earlier decision levels and ensure that those subtrees are not explored further.

In scheduling problems, one key challenge is to ensure that no two tasks are scheduled to be executed at the same time. This is modelled using the "DISJUNCTIVE" constraint, which is applied on a collection of tasks and enforces that any two tasks with a duration greater than zero should never be processed simultaneously by one machine [6]. One of the first algorithms explored to propagate this constraint is Edge-Finding, which remains a popular choice to date, thanks to its efficiency. The name originates from the initial use of graphs to represent the disjunctive, and the algorithm is based on the idea of identifying a task that must start after others have been completed. Due to its predominance, this paper also focuses on further exploring this algorithm, particularly exploring techniques that help improve its performance.

There has been tremendous progress in optimising the performance of an edge-finding propagator. The first efforts were made by Carlier and Pinson [5], giving an initial implementation with a time complexity of $O(n \log n)$, but using a complicated algorithm based on preemptive schedules and disjunctive graphs. Recently, Petr Vilím presents a simplified algorithm based on the idea of quickly computing the earliest completion time of subsets of tasks, achieving the same time complexity of $O(n \log n)$, however, with additional memory overhead due to the use of a custom data structure [20]. Moreover, in his recent work, Vilím also explores the idea of computing explanations for propagating disjunctive constraints, which appear to achieve significant speed-ups in combination with backjumping algorithms, specifically for larger instances [19].

However, while providing such massive improvements, previous work appears to focus only on optimising the propagation process and only scratches the surface on the impact of explanations on the performance of a search algorithm. Vilím explores the theoretical idea of justifications and provides instructions on how to generate them in a few disjunctive propagators, such as Edge-finding, Not-first/Not-last and Detectable Precedence, but lacks offering concrete algorithms to obtain them. Since then, it seems that the focus of exploring explanations has been shifted to other constraints [9, 15, 17], leaving the disjunctive behind. With the most recent developments in solvers, such as the idea of Lazy Clause Generation, having occurred after the latest advancements for the disjunctive constraint, the matter of how explanations would affect a propagator for the disjunctive constraint in a solver making use of LCG remains underexplored.

In this paper, I present an extension to the Edge-Finding algorithm introduced by Vilím with the purpose of extracting the variables for which explanations need to be generated. I also evaluate four strategies for handling the disjunctive constraint on a state-of-the-art CP solver that makes use of Lazy Clause Generation. Each variant reflects a different strategy for handling the disjunctive, ranging from no definition of the constraint to propagation combined with explanations. Specifically, I will compare the performance of the following:

(a) **Decomposed constraint**: The solver does not have any definition for the disjunctive and uses decomposition to define it.

(b) **Naive explanations**: The solver makes use of a basic Edge-Finding propagator for the disjunctive constraint. Explanations only consist of current domains of variables. Therefore, no generalisation is attempted.

(c) **Overload explanations**: The propagator generates explanations when a Resource Overload is detected but uses naive explanations for assignments.

(d) **Overload + Edge-finding explanations**: The propagator attempts to generalise explanations by extending variable domains every time an assignment is done or when a conflict is detected. The explanations are created using the introduced extension to Edge-Finding.

The formulas used to generate explanations are adapted from Vilím's work [19] to fit the problem definition used in this paper. The performance of each approach is assessed on instances taken from the OR-library [2]. These experiments will provide some insights on whether good explanations are worth exploring for an edge-finding algorithm, and how much can they improve the performance of a state-of-the-art solver, equipped with LCG.

From the experiments conducted, we confirm that the solver that makes use of meaningful explanations performs better than other versions, bringing improvements in recorded metrics of around 700% compared to Naive, on instances where both solvers terminate in time. Interestingly, the Overload Explanations variant only performed slightly better than the Naive one, producing very similar metrics but proving optimality faster. One unexpected result is the Decomposition variant that finds a solution closer to the optimal one than the Naive and Overload versions on most instances, however, struggling to prove optimality even on the easiest cases within the time limit.

In summary, my contribution is twofold. First, I introduce an adaptation of the algorithm described by Vilím to integrate explanation support. Second, I present an in-depth comparison of three different techniques to compute explanations in Edge-Finding and an evaluation of their performance. I also compare them to a baseline that makes use of decomposition.

The rest of the paper will be organised as follows. Section 2 presents a detailed definition of the problem at hand. Section 3 illustrates the current state-of-the-art and how this paper aims to improve the current body of knowledge. Section 4 introduces the notation and concepts that will be used throughout the rest of the paper. Section 5 presents the main contribution of the paper. Section 6 contains details about the way the experiment was conducted. It also describes and motivates the chosen datasets and benchmarks, and contains a discussion on the outcome of each experiment. Section 7 touches on the ethical aspects of this research and on the reproducibility of the methods. Finally, Section 8 presents the conclusions along with indications for possible future research.

2 Problem Description

2.1 Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is the result of modelling a combinatorial optimisation problem. It consists of a set of variables $X = \{x_1, ..., x_n\}$, each of them having a set of possible values, also known as the domain D, where $\mathrm{D}(x_i)$ denotes the domain of variable i, as well as a set of constraints C. The constraints are applied to a subset of variables $X' \subseteq X$ and enforce some properties that must be met at all times by the variables in X'. In our case, we restrict the domain of variables to integers, meaning that $\mathrm{D}(x_i) \in \mathbb{Z}$, $\forall\ i \in \overline{1, n}$. A CSP is called satisfiable if there exists at least one feasible assignment of values to each variable such that all constraints are fulfilled [3].

Constraint Programming is a popular approach that can be used to solve CSPs. It makes use of propagators to remove values from variable domains that lead to infeasible solutions. A propagator is a function that receives as input the domain of one variable, analyses the constraints of which it is part, along with previous assignments, and reasons about which values lead to unsatisfiability of the problem.

2.2 Disjunctive Constraint

To visualise the disjunctive, consider the following small example. We are given two tasks and one machine to process them: the first can start at any time in the interval $[0, 3]$, with a duration of 5 time units, while the second can start at $[2, 4]$, having a duration of 1. The objective is to find a schedule that completes all tasks as soon as possible. An optimal schedule for this example is shown in Figure 1.

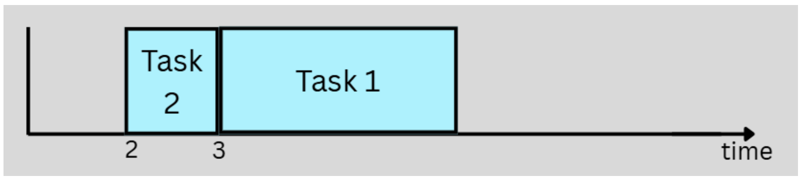

Figure 1 Optimal schedule for two task example.

Since this is a small example, we can apply simple reasoning on why task 2 must precede the first one: If the first task were to start at any other time point, it would be executed during all possible start times of task 2 (time points 2,3,4), meaning that it would be impossible to fit the second task into the schedule without violating the constraints. This implies that, in a feasible solution, task 2 must precede task 1, resulting in the above.

The formal definition of the disjunctive follows. We are given a collection of N tasks to execute on one machine, labelled as: $T = \{task_1, ..., task_n\}$. Each of the tasks has a set of possible start times, denoted as an interval $[est_i, lst_i]$, and a duration p_i, indicating the execution time of $task_i$ on the machine. The objective is to find a feasible schedule such that no two tasks are executed at the same time. This definition assumes that all tasks must be executed exactly once, there can be no preemptiveness (execution cannot be paused and resumed later), and the machine requires no setup time initially or between tasks. For simplicity, we also assume that all values of est_i and lst_i are non-negative integers, while all values for p_i are strictly positive integers, since tasks with duration 0 can be scheduled at any time without affecting the outcome.

Using this definition, we can use multiple disjunctive constraints to model job-shop scheduling problems, which are generalisations of practical problems that many people face nowadays [10]. Providing quick solutions to these problems is imperative for certain clients, such as factories manufacturing pieces that must pass through multiple machines, or even educational institutions scheduling classes for students [4]. They consist of scheduling N jobs over M machines, each taking a different time to process each task, so that the makespan (total time it takes for all jobs to finish executing) is minimised.

3 Related Work

3.1 Propagation Algorithms

The job-shop scheduling problem has many practical applications and has gained a lot of attention, ever since most of its variants, including the one I focus on in this paper, have been proven to be NP-hard [12]. This shows that a Constraint Programming approach based on exhaustive search is reasonable, since no algorithm that runs in polynomial time can be found. Therefore, there has been previous effort in defining constraint models for this problem, as well as developing an efficient propagator in order to help reduce the search space, and thus provide optimal solutions faster.

One of the first algorithms to be explored in propagating the disjunctive constraint was based on the idea of Edge-Finding, which remains a popular approach to this day. The first to explore such an algorithm were Carlier and Pinson [5], who explored computing preemptive schedules and identifying precedence relations based on the graph representation of the disjunctive, making it quite complicated to understand and implement in practice. Nevertheless, it achieves a worst-case time complexity of $\mathrm{O}(\mathrm{n} \log n)$, resulting in one of the quickest approaches to propagate a disjunctive constraint to date.

These ideas were later extended by Martin and Shmoys [14], providing other approaches to update the bounds of variables in Edge-Finding, however, with a worst-case time complexity of $\mathrm{O}(n^2)$. Despite their algorithms seeming slower, they were sometimes preferred in practice due to them using conceptually simple ideas, through renouncing the disjunctive graph formulation, and therefore being easier to implement.

The current state-of-the-art for Edge-Finding is presented by Petr Vilím, who shows an approach that is conceptually easy to understand and achieves an improved worst-case time complexity of $\mathrm{O}(\mathrm{n} \log n)$ [20]. He achieves these results by making use of a custom data structure, allowing for precomputation and fast retrieval of subsets of tasks by storing them in a balanced binary tree, for which a precedence relation can then be found. Vilím also provides efficient algorithms for two other disjunctive propagation algorithms: Not-first/Not-last and Detectable Precedences, which are newer, less popular approaches to propagating the disjunctive constraint [20]. This research makes use of this implementation for Edge-Finding, as it balances performance and conceptual complexity.

3.2 Solver Improvements

More recent methods explored to boost the performance of solvers have focused not on improving the propagation techniques, but on advancing the search part of solvers. The first idea was to record explanations for domain modifications during propagation and use them to reason about the level on which an infeasible assignment may have happened. If this can be deduced, the solver would be able to backtrack multiple levels upon identifying a conflict and directly reassign the variable most likely to cause conflicts. This reasoning is achieved by lazily generating clauses from explanations and combining them whenever possible, as their efficiency is directly proportional to the quality of an explanation [15, 8]. This idea is incorporated in most of the top-performing solvers available today.

An initial intuition for explanations was also explored for an edge-finding propagator by Vilím, which presents an idea of computing justifications based on conflict windows [19]. However, the work was done before LCG was discovered, and thus it remains unclear what the effects of conflict windows are on the performance of a propagator integrated in a state-of-the-art solver. This represents the base motivation for this research: understanding whether these conflict windows can be used in combination with Lazy Clause Generation, and how effective these clauses are in pruning domains. Moreover, Vilím only mentions that $O(n^2)$ edge-finding algorithms can be modified to incorporate justifications, having published his own work on propagators after those findings [19]. Therefore, I also explore how his own $O(\text{n} \log n)$ algorithm can be modified to generate explanations.

4 Preliminaries

4.1 Notations

In our problem, we are given an interval of start times, defined through a lower and upper bound. However, we also need to introduce notation for extracting the earliest and latest completion times of tasks, since Edge-Finding makes use of those to reason about precedence. We can do so by adding the duration to the boundaries of the intervals as follows:

- $ect_i = est_i + p_i$
- $lct_i = lst_i + p_i$

This can also be extended to define the start and completion times for sets, which will be used in later formulas. Consider a subset of tasks $\Omega \subseteq T$, for which:

- $est_\Omega = \min\{est_i | i \in \Omega\}$ = Earliest start time of all tasks in Ω.
- $ect_\Omega = \max\{est_{\Omega'} + p_{\Omega'} | \Omega' \subseteq \Omega\}$ = Earliest completion time of all tasks in Ω.
- $lct_\Omega = \max\{lct_i | i \in \Omega\}$ = Latest completion time of all tasks in Ω.
- $p_\Omega = \sum_{i \in \Omega} p_i$ = Sum of all the durations of tasks in Ω.

We denote a precedence relation using the $\ll$ operator. Thus, the relation $i \ll j$ means that j must start after i has been completed. We can also define the precedence relation between a set and a task as follows: $\Omega \ll i$ means that i can only start after all tasks in Ω have been completed. More formally, we can interpret this precedence rule as follows: $i \ll j \rightarrow est_j \geq ect_i$. The same applies also for sets: $\Omega \ll i \rightarrow est_i \geq ect_\Omega$.

4.2 Propagation

4.2.1 Edge-finding

Edge-Finding is a propagation algorithm that is based on identifying pairs of: a subset of tasks $\Omega \subseteq T$ and a task i such that i must start after all tasks in Ω have been completed [5]. Formally, the rule can be defined as follows:

$$\forall\, \Omega \subset T, \forall\, i \in T, i \notin \Omega \implies est_{\Omega\cup\{i\}} + p_{\Omega\cup\{i\}} > lct_{\Omega} \rightarrow \Omega \ll i$$

This formula shows that, if task i was part of Ω, there would be no feasible way to schedule all tasks, and therefore i must start after all tasks in Ω are completed. If such pairs (Ω, i) are found, we can infer the following update rule:

$$est_i = \max\{est_i,\ ect_{\Omega}\}$$

which means that i must start after all tasks in Ω can be finished earliest.

4.2.2 Θ-Λ-Trees

This data structure was introduced by Petr Vilím [20] and plays an important role in keeping the complexity of the propagation algorithm at O(n $\log n$). It is represented as a balanced binary tree, with leaves representing tasks, sorted by earliest start time in non-strictly increasing order, and internal nodes keeping track of some precomputed values for intermediate sets, comprised of tasks that are stored in leaves for which the respective internal node is an ancestor. The idea behind this tree is to keep track of two sets Θ and Λ, each having the following purpose [20]:

- Θ: A subset of tasks for which we want to compute the earliest completion time: ect_{Θ}. These are also called "white tasks".
- Λ: A subset of tasks which are not in Θ, but for which we want to know how ect_{Θ} would change if we were to add one $task_i \in \Lambda$ to Θ. These are also known as grey tasks. Note that $\Theta \cap \Lambda = \emptyset$.

The goal is to quickly compute ect_{Θ}, which is an extension of Ω, with $ect_{\Omega} = ect_{\Theta}$, to be used in the Edge-Finding rule. For this, each node ν in the tree should keep track of the following values:

- ect_{ν} = ECT of white tasks for which ν is an ancestor.
- ΣP_{ν} = Sum of all durations of white tasks for which ν is an ancestor.
- $\overline{ect_{\nu}}$ = ECT of white tasks for which ν is an ancestor, including at most one grey task.
- $\overline{\Sigma P_{\nu}}$ = Sum of all durations of tasks in Θ for which ν is an ancestor, including at most one grey task.

This allows for quick application of the edge-finding rule to many sets, considering $\Omega \subseteq \Theta$ and $i \in \Lambda$ the task included in the calculation of $\overline{ect_{\nu}}$. Moreover, due to the balanced binary tree structure, the most time-consuming operations such as generating a tree from scratch, or pre-computing values in nodes have a worst-case time complexity of O(n $\log n$), while the operations of adding/removing elements from Θ have a time complexity of just O($\log n$) [20].

4.3 Explanations

4.3.1 General Concept

Explanations are a fundamental concept for LCG solvers. They are recorded once a domain has been modified, or when a conflict occurs, and represent the reason for the inferences

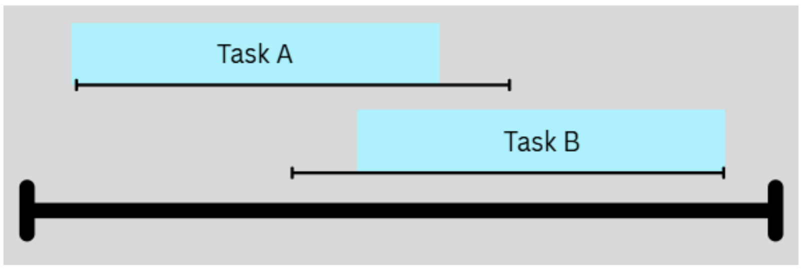

Figure 2 Conflict Window Illustration. Adapted from [19].

made by the propagation algorithm. These reasons can be combined to generate clauses on demand, which contribute to learning nogoods, a conjunction of predicates which can be used for as long as they remain valid. This process is called Lazy Clause Generation and is used in almost all modern CP solvers, providing noticeable improvements in the search process.

Explanations for scheduling problems are recorded in the form of conflict windows. They are defined per activity and represent an extension of its domain $[est_i, lct_i]$. The extended interval illustrates the largest possible period in which, if all tasks are executed, there is no solution to the scheduling problem. They can be identified by considering potential conflicts with other tasks and extending the interval until a feasible solution exists. Note that a task can still start at a time in its conflict window, the problem only arises if all tasks are assigned a starting time that is part of their conflict windows.

4.3.2 Overload Conflict Windows

Consider the example in Figure 2, consisting of two tasks, each having their execution window represented by the small bars below the rectangles. We see that there exists no feasible schedule such that the tasks do not overlap. Following the above definition, we can generalise the conflict windows for both tasks to the interval represented by the thicker bar, since a feasible schedule would still not be possible in this generalisation [19].

Following this idea, we can define explanations for Resource Overloads, used in the third and fourth variants. Let Ω be the set of tasks for which there is no feasible schedule and Δ be the maximum time which can be added to the schedule so that there would still be an overload. Formally, $\Delta = p_\Omega - (lct_\Omega - est_\Omega) - 1$. In this case, the conflict windows for all tasks in Ω can be widened per Vilím's definition, as follows [19]:

$$\left[est_\Omega - \lfloor\frac{\Delta}{2}\rfloor, lct_\Omega + \lceil\frac{\Delta}{2}\rceil\right]$$

Now, we must adapt this to our problem definition. First, we notice that the upper bound of this formula is for lct, however, we care for lst. It may also be the case that the overload occurs close to time point 0 ($est_\Omega \approx 0$), and the left bound of the window would go beyond that ($est_\Omega - \lfloor\frac{\Delta}{2}\rfloor < 0$). Following our non-negativity assumption from Section 2, we lose value from the explanation if we allow for this, since those values are impossible anyway. Taking these into account, we end up with the following conflict windows:

$$\forall\ i \in \Omega \rightarrow [est_\Omega - ol, lct_\Omega + \Delta - ol - p_i]$$

where $ol = \min\{est_\Omega, \lfloor\frac{\Delta}{2}\rfloor\}$

4.3.3 Edge-finding Conflict Windows

For the edge-finding algorithm, we can infer the necessary explanations by extending the bounds in the edge-finding rules. Logically, if a precedence $\Omega \ll i$ is found, this means that: if $i \in \Omega$ and i were to start at some time before ect_Ω, there would be an overload. Thus, intuitively, we should set the conflict window as follows:

$$\forall\ j \in \Omega \cup \{i\} \rightarrow [est_{\Omega\cup\{i\}}, est_{\Omega\cup\{i\}} + p_{\Omega\cup\{i\}} - 1 - p_j]$$

However, this rule does not take into account the definition of ect_Ω. It might not be the case that $est_{\Omega\cup\{i\}} + p_{\Omega\cup\{i\}} = ect_{\Omega\cup\{i\}}$. This means that we need to consider the way in which ect_Ω was computed and, consequently, set more specific windows for the activities in $\Omega' \subseteq \Omega$, which is the subset of tasks directly responsible for the result of ect_Ω. Following this reasoning, we obtain the following rules, as per Vilím [19], adapted to our problem definition:

$$\begin{aligned}\forall\ j \in \Omega' &\rightarrow \left[\max\{est'_i - p_{\Omega'}, est_{\Omega\cup\{i\}}\},\ est_{\Omega\cup\{i\}} + p_{\Omega\cup\{i\}} - 1 - p_j\right]\\ \forall\ j \in (\Omega \setminus \Omega') &\rightarrow \left[est_{\Omega\cup\{i\}},\ est_{\Omega\cup\{i\}} + p_{\Omega\cup\{i\}} - 1 - p_j\right]\\ i &\rightarrow \left[est_{\Omega\cup\{i\}}, \infty\right)\end{aligned}$$

where est'_i is the new lower bound for activity i.

5 Main Contributions: Computing Explanations from Θ-Λ-Trees

From this point forward, the main accomplishments of this research are presented. Specifically, a description of an algorithm used to generate explanations from an Edge-Finding implementation using Θ-Λ-Trees is defined. The next section also presents details regarding the experiments conducted. These results should provide an answer to the knowledge gaps presented in previous sections.

Modifying the algorithm based on Θ-Λ-Trees to support these explanations is not trivial. The algorithm propagates by finding $i \in \Lambda$ such that $\Theta \ll i$, however, assuming that $\Theta = \Omega$ is incorrect. Since the algorithm removes tasks from Θ in decreasing order by lct, it may be the case that Θ still contains some tasks with a very low est, which should not be included in Ω and should not have conflict windows set.

To visualise this, consider the following three-task example: the first task has an interval $[0, 25]$ and a duration of 5, the second task has $[24, 24]$ and $p = 3$, and the third task has start times $[24, 30]$ and $p = 4$. According to the Edge-Finding rule, when $\Omega = \{task_2\}$, we have: $24 + 7 > 27 \rightarrow est_3 = \max\{24, 27\} = 27$. When task 3 is updated, Edge-Finding also finds $ect_\Theta = 27$. However, $\Theta = \{task_1, task_2\}$, for which the rule does not hold.

To address this, we need to iterate over Θ and remove tasks, in increasing order by est, until we have $ect_{\Theta\cup\{i\}} + p_{\Theta\cup\{i\}} > lct_\Theta$. This new set represents the Ω we are looking for. To obtain Ω', we can apply the same logic, which is to remove tasks from Ω until the following holds: $ect_\Omega = est_{\Omega'} + p_{\Omega'}$. The algorithm for obtaining Ω and Ω' works as follows:

Listing 1 Algorithm for Computing Edge-Finding Explanations. Input: (Θ-Λ-Tree, i)

```
tasks := Θ
Δ := $\min\ \{est_i,\ est_{tasks}\}$ + $p_{tasks}$ + $p_i$
while Δ $\leq$ $lct_{tree}$ do begin
    tasks := tasks \ {j ; $est_j$ = $est_{tasks}$}
    Δ := $\min\ \{est_i,\ est_{tasks}\}$
end
Ω := tasks
while $ect_{tree}$ $\neq$ $ect_{tasks}$ + $p_{tasks}$ do begin
```

```
    tasks := tasks \ {j ; $est_j$ = $est_{tasks}$}
end
$\Omega'$ := tasks
$\Omega$ := $\Omega \setminus \Omega'$
```

The algorithm exploits the fact that Θ-Λ-Trees store tasks sorted by est, thus reducing the complexity from $O(2^n)$ to $O(n)$. This does increase the total runtime complexity of Edge-Finding to $O(n^2 \log n)$, however, it is necessary to record explanations. The first while loop is responsible for removing activities from "tasks" until the condition for Ω is satisfied, while the second loop continues to remove activities until we obtain Ω'. By definition, these sets must exist. Therefore, the situation where any while loop terminates by removing all tasks from the set cannot occur.

6 Experimental Setup and Results

This section presents a description of the experiments conducted in this paper, their results, and how they can be interpreted. The main goal is to evaluate the performance of the four variants of the Pumpkin solver and thus gain insight into whether explanations are worth exploring for the disjunctive constraint and how do the solver variations perform when compared against the baseline. The results confirm this hypothesis, namely that the solver variant using explanations performs significantly better than the other ones in terms of all recorded metrics, regardless of the computational overhead required to generate them.

6.1 Setup

All code related to the propagator that was used for this experiment is open source and added as an extension to the TU Delft Pumpkin Solver [7]. Various optimisations were tested in terms of using different data structures on every variant of the solver, and the best performing code in terms of runtime was chosen for each of them.

Based on previous research, most of the tests used were taken from the OR-library [2], which contains a wide collection of job-shop instances. I also use a subset of those tests, specifically the instances from Lawrence [11], as they contain many tests covering a wide range of difficulties. I also consider the ORB test instances [1] as they are small but challenging problems. Only these datasets are considered because they are the only ones where each variant can make continuous progress, as well as having the additional benefit of knowing the optimal makespan, which we can use to check the correctness of the variants. Another collection of used tests are generated pseudo-randomly using a specialised algorithm [18], then translated into MiniZinc[1] format. This approach is taken into account to see whether the results measured on known instances also hold on random inputs. This has previously been used with predefined seeds. However, the reported instances are very large and would not provide significant insight, due to solvers getting stuck at intermediate solutions for a very long amount of time. Therefore, I use the generator to create some easier instances, while ensuring to report the seeds used.

Metrics will be reported for tests that show the most interesting results related to the performance of each solver variant. Other instances that provide similar results can be found in the GitHub repository. Some instances will be compared individually, containing tests from all difficulty ranges, while others will be presented in the form of an average of recorded

[1] High-level constraint modelling language, which supports Pumpkin. https://www.minizinc.org/

Test	la05	la06	la19	la21	la26	la34	la37	la38	la11-15	orb	10j10m	15j8m	la31-35
Size	10x5	15x5	10x10	15x10	20x10	30x10	15x15	15x15	20x5	10x10	10x10	15x8	30x10
OM	593	926	842	1046	1218	1721	1397	1196	–	–	–	–	–

Table 1 Test instances used in result discussion. OM = known Optimal Makespan

metrics from multiple instances with the same size. The performance of each strategy is assessed on all chosen test instances, on an Intel I7-11370H processor, with a time limit of 20 minutes per run, reporting all intermediate solutions found during the allotted time. In all cases, the solver uses a free search strategy, based on VSIDS [13]. Many solver metrics are also measured during the execution, saved for each identified solution, as well as when the time limit has been reached or when optimality has been proven. Table 1 contains an overview of the tests chosen for the comparison, their size, and their known optimal makespan.

As metrics, I am primarily interested in the Number of Conflicts (NC). This provides a good estimate of how nogoods are used during the search process, as good explanations lead to better learnt clauses, which are used in pruning the search space, thus leading to fewer conflicts. Another important metric to consider is the average Literal Block Distance (LBD), which shows how many unique decision levels there are in a learnt clause. Ideally, predicates learnt on the same decision level are better, as they are more likely to imply each other and can produce better nogoods, which is why we want this value to be low. Additionally, I report the Average Backtrack Amount (ABA), which is relevant to how different explanation techniques influence the backtracking process. A higher number means that conflict-directed backjumping can be used more often, which is a direct consequence of good explanations. Finally, we also record the total runtime, as it represents how quickly an instance can be solved. Although this metric does not provide interesting results on its own, we can use it to study the relative performance in a direct comparison between two methods.

6.2 Tested Variants

The first variant of the solver used for testing is Pumpkin, using decomposed constraints. In this case, the solver has no definition for the disjunctive and has to represent it by introducing additional variables and imposing available constraints on them. There are some known advantages to this approach, the main one being that decomposition can be used to break the large problem into many smaller ones and can make use of relevant generated nogoods throughout the search process to compete with propagators. This has been proven to be a viable approach for the cumulative constraint [16] and therefore should serve as a viable baseline to compare against the different representations of the disjunctive.

The second variant of the solver would be a modified version of Pumpkin, having a definition for the disjunctive, as well as an Edge-Finding propagator, but using Naive explanations. This strategy explains all decisions with a conjunction of the domains of all variables, without attempting any generalisation. We cannot guarantee the generation of any useful clauses in this case, thus the solver relies more on the variable and value selection logic, rather than explanations.

The third variant we are evaluating builds upon the above by adding overload explanations. In this case, we are able to explain why a conflict occurs by generating conflict windows for the variables involved. This is the first approach in which the solver is able to generalise the bounds of variables, and thus should lead to better learnt clauses. However, conflicts are not a common outcome of propagation. Therefore, I do not expect this approach to bring any significant improvements.

The final variant considered in this experiment comes by also explaining domain pruning. This is achieved through the Edge-Finding explanation formulas defined previously in Section 4.3.3. This approach allows us to investigate the full potential of explanations and their true impact on the propagator performance. Although generating these explanations adds some complexity overload to the algorithm, I expect this to be mitigated by the quality of the explanations and their usefulness in the search process. Therefore, I expect a large difference between this approach and the aforementioned.

6.3 Results and Discussion

The following subsections present tables containing results, each providing a direct comparison between two solver variants, which have been chosen to highlight the main differences each strategy brings. Specifically, the following strategies will be compared: Naive vs. Decomposition, Overload vs. Naive, and Edge-Finding vs. Naive. The goal is to compare the performance of advanced explanation strategies to the Naive variant, while also relating the results to the Decomposition approach, which provides the most insight when also compared to the Naive approach. The tables will contain values for the recorded metrics, as well as the column "End", which refers to the total time spent until the solver proves optimality.

6.3.1 Naive vs. Decomposition

Test	Decomposition						Naive					
	BM	NC	LBD	ABA	Time	End	BM	NC	LBD	ABA	Time	End
la05	593	165	4.1	6.3	2.5s	5.3s	593	38	11.5	3.1	0.84s	0.84s
la06	926	8.8K	9.9	1.7	13s	20m	926	172	18.2	4.2	5.37s	5.37s
la19	842	3.8K	8.4	2.6	18.6s	20.4s	842	16.7K	21.2	1.4	39.2s	107s
la21	1050	30K	14.6	2.4	107s	20m	1053	109K	32.7	1.2	878s	20m
la26	1218	100K	9.6	2.2	316s	20m	1258	10.5K	70	2.1	343s	20m
la34	3106	13K	6.7	20.4	1172s	20m	2109	10.8K	62.1	2.9	1200s	20m
la37	1397	12.5K	14.9	4.6	343s	604s	1397	15.8K	38	2.2	667s	667s
la38	1196	135K	10.3	2.0	825s	20m	1235	21.5K	39.8	1.9	685s	20m

Table 2 Metric comparison between Pumpkin using decomposition (left) and Naive explanations (right), on individual tests. BM = Best Makespan found in the time limit, NC = Number of Conflicts, LBD = Literal Block Distance, ABA = Average Backtrack Amount, End = Time needed to prove optimality

Tests	Decomposition						Naive					
	AD	NC	LBD	ABA	Time	End	AD	NC	LBD	ABA	Time	End
la11-15	0	5.3K	9.8	4.3	29.3s	20m	0	1.3K	25.9	3.1	10.9s	10.9s
orb01-10	0	12.5K	8.9	2.4	24s	45.6s	0	33K	17	1.3	84.1s	389s
10j10m01-10	0	474	4.4	9.2	11.6s	11.9s	0	6K	16.5	4.2	16.4s	19.4s
15j8m01-10	0	7.9K	11	5.4	36.7s	567s	0	5.3K	28.8	3.2	32.5s	32.5s
la31-35	539	9.3K	6.2	24.3	1172s	20m	87.6	8.9K	49.4	3.7	1172s	20m

Table 3 Metric comparison between Pumpkin using decomposition (left) and Naive explanations (right), on averaged tests. AD = Average distance from the best makespan found to the known Optimal.

Interestingly, the solver variant using decomposition shows surprising results, as can be seen in both Tables 2 and 3, outperforming the Naive on some difficult tests such as la21, la26, la37, and la38, in terms of runtime. However, it seems like it struggles to prove optimality, as can be seen in instances la06 and la26, where it finds the known optimal solution surprisingly quickly, but it gets stuck and runs out of the 20 minutes allotted time. It also performs well in terms of the metrics relevant to explanations, namely average LBD and ABA, where it demonstrates that it also makes use of good clauses to speed up its search process.

The performance is also surprising on averaged tests. Although the issue regarding proving optimality persists, the solver is still able to find good solutions quickly. The most surprising performance can be seen for the orb tests, in Table 3, where it is also able to prove optimality in a rather short amount of time. However, on harder tests, such as la31-35, it shows that it struggles with advancing solutions, as can be seen in the "AD" column which tells us that, on average, the solver using decomposition is significantly farther from the optimal solution than the Naive approach.

6.3.2 Overload vs. Naive

Test	Naive						Overload					
	BM	NC	LBD	ABA	Time	End	BM	NC	LBD	ABA	Time	End
la05	593	38	11.5	3.1	0.84s	0.84s	593	20	11.2	5.3	0.7s	0.7s
la06	926	172	18.2	4.2	5.37s	5.37s	926	230	19.1	6.1	6.94s	6.94s
la19	842	16.7K	21.2	1.4	39.2s	107s	842	5.5K	17.1	1.8	23.6s	108s
la21	1053	109K	32.7	1.2	878s	20m	1052	66.5K	30.7	1.3	398s	20m
la26	1258	10.5K	70	2.1	343s	20m	1218	50.7K	45.6	1.5	822s	823s
la34	2109	10.8K	62.1	2.9	1200s	20m	1721	12.8K	74.5	3.6	1197s	1197s
la37	1397	15.8K	38	2.2	667s	667s	1400	44.7K	27.3	1.7	1036s	20m
la38	1235	21.5K	39.8	1.9	685s	20m	1208	63.4K	35.8	1.6	1192s	20m

Table 4 Metric comparison between Pumpkin using Naive explanations (left) and Overload explanations (right), on individual tests.

Tests	Naive						Overload					
	AD	NC	LBD	ABA	Time	End	AD	NC	LBD	ABA	Time	End
la11-15	0	1.3K	25.9	3.1	10.9s	10.9s	0	667	23.7	4.6	12s	12s
orb01-10	0	33K	17	1.3	84.1s	389s	0	28.3K	17.9	1.5	56.2s	168s
10j10m01-10	0	6K	16.5	4.2	16.4s	19.4s	0	5K	15.3	3.9	17s	18.4s
15j8m01-10	0	5.3K	28.8	3.2	32.5s	32.5s	0	7.7K	25.2	3.2	52.8s	52.8s
la31-35	87.6	8.9K	49.4	3.7	1172s	20m	0	10.4K	48.8	4.9	871s	871s

Table 5 Metric comparison between Pumpkin using Naive explanations (left) and Overload explanations (right), on averaged tests.

Using Overload explanations slightly outperforms Naive in terms of all recorded metrics, which is an expected result, as conflicts are not a common result of propagation, resulting in using Naive explanations in most cases. An interesting result is la37, where the Naive solver managed to prove optimality, while the one making use of explanations did not even reach the optimal solution, as can be seen in Table 4. A similar situation can be seen in Table 5, in the third row, where the Naive outperforms the Overload version in terms of runtime on the randomly generated tests. However, this seems like an isolated situation, where it may be the case that the former took a good path in the search tree. In general, on larger tests, it seems that, even with just overload explanations, the performance improvement can still be noticeable, as it appears on tests la31-la35 in Table 5, where we manage to decrease the average difference between the best solution found and the known optimal from 87.6 straight to 0.

On the other hand, both the Overload and Naive versions appear to perform poorly in terms of average LBD and ABA, when compared to the baseline solver. The figures for LBD are higher, indicating that the predicates in the learnt clauses were inferred at different decision levels, suggesting lower-quality nogoods. At the same time, we see that the numbers for ABA are low in all cases, suggesting that the solver cannot make good use of conflict-driven backjumping, thus potentially getting stuck in subtrees that only lead to infeasible solutions. It may be the case that low values for ABA can be good, for instance, in the situation where a solver takes a good path in the search tree, it may not be necessary to backtrack many steps. However, being in a "good" subtree would also imply that the solver would converge more quickly to optimal solutions, which is not always the case, given that the baseline mostly outperforms these variants, as we learnt from Tables 2 and 3.

6.3.3 Edge-Finding vs. Naive

Test	Naive						Edge-Finding					
	BM	NC	LBD	ABA	Time	End	BM	NC	LBD	ABA	Time	End
la05	593	38	11.5	3.1	0.84s	0.84s	593	63	3.6	14	0.86s	0.86s
la06	926	172	18.2	4.2	5.37s	5.37s	926	200	3.8	27.1	2.7s	2.7s
la19	842	16.7K	21.2	1.4	39.2s	107s	842	1.6K	9.8	7.8	7.1s	10s
la21	1053	109K	32.7	1.2	878s	20m	1046	1.1K	7.2	31.9	24.4s	212s
la26	1258	10.5K	70	2.1	343s	20m	1218	2.9K	10.9	59.3	182s	182s
la34	2109	10.8K	62.1	2.9	1200s	20m	1721	4.3K	5.2	121.7	774s	774s
la37	1397	15.8K	38	2.2	667s	667s	1397	2.9K	8.6	40	98.5s	98.5s
la38	1235	21.5K	39.8	1.9	685s	20m	1196	44K	16.3	6.8	457s	20m

Table 6 Metric comparison between Pumpkin using Naive explanations (left) and Edge-Finding explanations (right), on individual tests.

Tests	Naive						Edge-Finding					
	AD	NC	LBD	ABA	Time	End	AD	NC	LBD	ABA	Time	End
la11-15	0	1.3K	25.9	3.1	10.9s	10.9s	0	398	6.2	36.8	7.95s	7.95s
orb01-10	0	33K	17	1.3	84.1s	389s	0	6.2K	11.9	6.4	15.8s	23s
10j10m01-10	0	6K	16.5	4.2	16.4s	19.4s	0	567	4.4	24	7.4s	7.5s
15j8m01-10	0	5.3K	28.8	3.2	32.5s	32.5s	0	740	6.1	35.7	15.8s	15.8s
la31-35	87.6	8.9K	49.4	3.7	1172s	20m	0	4.3K	5.2	119.7	715s	715s

Table 7 Metric comparison between Pumpkin using Naive explanations (left) and Edge-finding explanations (right), on averaged tests.

Tables 6 and 7 show that the initial hypothesis is correct, that is, good explanations do provide a significant improvement in most cases, when compared to naive explanations, or even decomposition, when looking back at Tables 2 and 3. There is a sharp decrease in the average LBD, as well as a noticeable increase in the Average Backtrack Amount metrics, indicating that the learnt clauses are more general and can be used more often. This also leads to a sharp decrease in the Number of Conflicts metric, as we can use these clauses to prevent future assignments that lead to infeasible schedules. We also show that proving optimality also becomes an easier task when the solver makes use of good explanations on instances orb01-orb10, in Table 7, where previous versions indicate large differences between the average time it takes to find the optimal solution and the average time to demonstrate it. The randomly generated instances also presented in Table 7 confirm these results, showing significant improvements in terms of NC, LBD and total runtime.

7 Responsible Research

In this research, I work with an algorithm that attempts to find an optimal solution to instances of the job-shop scheduling problem. The algorithm used is deterministic and will produce the same output when executed on the same input every time. All code used in this project is open source and publicly available[2], and all datasets and the source where I obtained them have been clearly stated in the paper. All results are clearly presented, and a detailed discussion is carried out. The only instance where ethical considerations may arise is related to using randomly generated test cases. To address this, I ensure that my work is reproducible by clearly stating the seeds used in generating the test cases, as well as having the generator code publicly accessible.

For the purpose of writing this paper, I have not used generative AI for content or ideas. The only instance arises when using a spell checker powered by artificial intelligence, with the scope of identifying potential spelling mistakes. Throughout the research, I have also used code autocompletion tools based on AI, such as GitHub Copilot, to help me with syntax-specific issues and generate unit tests to verify the correctness of the solutions.

8 Conclusions and Future Work

In summary, I have presented a method for generating explanations from Θ-Λ-Trees. I also used it in a benchmark between multiple variants of computing explanations for propagating a disjunctive constraint using an Edge-Finding algorithm. I also compared these methods with a baseline solver that uses decomposition to represent the disjunctive. The experiments are carried out in such a way that they can be reproducible under any conditions.

The results prove the initial hypothesis, showing that good explanations can achieve significant speed-ups in the performance of a state-of-the-art solver that makes use of Lazy Clause Generation. We also show a less expected result: Decomposition sometimes performs better than a propagator using Naive explanations, and even than the version that provides explanations when an Overload occurs. However, the best approach is the variant that provides meaningful explanations from propagation, which significantly outperforms the rest on all test cases.

This work is limited to exploring four variants of solvers to propagate the disjunctive. Future work could expand this idea by exploring other explanation strategies and seeing whether they perform better than the current state-of-the-art. Another approach that would be interesting is to explore new explanation strategies for an edge-finding algorithm, or even a different one, and see whether these results can be improved even further. A different path to follow in future research would be to further investigate the results achieved by decomposition. This has been previously studied for the cumulative constraint [16], and it would be interesting to learn whether similar results can be obtained for the disjunctive.

[2] Accessible at: https://github.com/Bradul/Pumpkin

References

1 D Appelgate and W Cook. A computational study of jobshop scheduling. *ORSA Journal on Computing*, 3, 1991.

2 J. E. Beasley. Or-library. `https://people.brunel.ac.uk/~mastjjb/jeb/info.html`, 1990. Accessed: 27/5/2025.

3 Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, 1999. URL: `https://www.sciencedirect.com/science/article/pii/S0377221798003646`, `doi:10.1016/S0377-2217(98)00364-6`.

4 Peter Brucker. The job-shop problem: Old and new challenges. In *Proc. of the 3rd Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA)*, pages 15–22, 2007.

5 J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78(2):146 – 161, 1994. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-0028518441&doi=10.1016%2f0377-2217%2894%2990379-4&partnerID=40&md5=8ec74f0a1a261bcfde44451ad5cda684`, `doi:10.1016/0377-2217(94)90379-4`.

6 Sophie Demassey. Global constraint catalog. `https://sofdem.github.io/gccat/gccat/Cdisjunctive.html`, n.d. Accessed: 30/04/2025.

7 Emir Demirović, Maarten Flippo, Imko Marijnissen, Konstantin Sidorov, and Jeff Smits. Pumpkin: A lazy clause generation constraint solver in rust, 2024. URL: `https://github.com/ConSol-Lab/Pumpkin`.

8 Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, pages 352–366, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

9 Ian P. Gent, Ian Miguel, and Neil C. A. Moore. Lazy explanations for constraint propagators. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5937 LNCS:217 – 233, 2010. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-77749295468&doi=10.1007%2f978-3-642-11503-5_19&partnerID=40&md5=27c053de36c038091d30fa59adabe734`, `doi:10.1007/978-3-642-11503-5_19`.

10 Google Developers. Constraint optimization | or-tools. `https://developers.google.com/optimization/cp`, 2024. Accessed: 15/06/2025.

11 S Lawrance. Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques. *GSIA, Carnegie Mellon University*, 1984.

12 J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1(C):343 – 362, 1977. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-77951139709&doi=10.1016%2fS0167-5060%2808%2970743-X&partnerID=40&md5=69f3733133f41a5a12081a68a08c6385`, `doi:10.1016/S0167-5060(08)70743-X`.

13 Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers. In *Hardware and Software: Verification and Testing: 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings 11*, pages 225–241. Springer, 2015.

14 Paul Martin and David B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1084:389 – 403, 1996. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-84947927805&doi=10.1007%2f3-540-61310-2_29&partnerID=40&md5=965db11e8592fe527781bd9d8882e7b6`, `doi:10.1007/3-540-61310-2_29`.

15 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357 – 391, 2009. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-67651111759&doi=10.1007%2fs10601-008-9064-x&partnerID=40&md5=5bb7b986b2072983cff3555754d38dbc`, `doi:10.1007/s10601-008-9064-x`.

16 Andreas Schutt, Thibaut Feydy, Peter J Stuckey, and Mark G Wallace. Why cumulative decomposition is not as bad as it sounds. In *International conference on principles and practice of constraint programming*, pages 746–761. Springer, 2009.

17 Andreas Schutt, Thibaut Feydy, Peter J Stuckey, and Mark G Wallace. Solving rcpsp/max by lazy clause generation. *Journal of scheduling*, 16:273–289, 2013.

18 Eric Taillard. Benchmarks for basic scheduling problems. *european journal of operational research*, 64(2):278–285, 1993.

19 Petr Vilím. Computing explanations for the unary resource constraint. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2005)*, volume 3524, pages 396 – 409, 2005. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-26444453950&doi=10.1007%2f11493853_29&partnerID=40&md5=cf493e7730ec0a0e274a1f3ed0f0ff48`, `doi:10.1007/11493853_29`.

20 Petr Vilím. *Global constraints in scheduling.* Ph.d. thesis, Univerzita Karlova, Matematicko-fyzikální fakulta, 2007. Available at: https://dspace.cuni.cz/bitstream/handle/20.500.11956/12252/140038772.pdf.

A Seeds of randomly generated tests

This section presents a table containing the seeds used for obtaining the randomly generated tests, which have been used in Section 6 for reporting results.

Filename	Time Seed	Machine Seed
instance_10j10m01.dzn	595312809	1226873846
instance_10j10m02.dzn	1994010092	674268132
instance_10j10m03.dzn	684069210	17711631
instance_10j10m04.dzn	433393242	1856683078
instance_10j10m05.dzn	1737541870	2123655630
instance_10j10m06.dzn	1053901157	2093288715
instance_10j10m07.dzn	197486053	2072736806
instance_10j10m08.dzn	541533994	836141230
instance_10j10m09.dzn	591728687	1250988364
instance_10j10m10.dzn	64076732	175007812
instance_15j8m01.dzn	166140137	21143938
instance_15j8m02.dzn	472453410	1957606298
instance_15j8m03.dzn	325953502	1085494782
instance_15j8m04.dzn	417269198	833500955
instance_15j8m05.dzn	1001380920	1585883286
instance_15j8m06.dzn	607390163	1104152727
instance_15j8m07.dzn	1561051784	2003679251
instance_15j8m08.dzn	422129019	1697884185
instance_15j8m09.dzn	1213865971	957331870
instance_15j8m10.dzn	1039225514	117613006

Table 8 Seeds used for randomly generated instances