UCLouvain

École polytechnique de Louvain

# Solving industrial scheduling problems with Constraint Programming and insertion sequence variables

Authors: **Julien Gelders, Emma Legrand , Damien Van Meerbeeck**
Supervisor: **Pierre Schaus**
Readers: **Augustin Delecluse, Charles Thomas, Roger Kameugne**
Academic year 2022–2023
Master [120] in Computer Science and Engineering

**Abstract**

Over the years, extensive research has been conducted on various optimization problems such as vehicle routing problems, scheduling problems, and more. In our case, we are interested in the Job-Shop problem, which has already been widely researched in the literature. Many techniques have been proposed to improve the results for large-scale problems where the exact solution is unknown. Some notable approaches include Tabu search [6], Genetic algorithms, Dynamic programming [11], among others.

One technique of particular interest to us is Constraint Programming, which is commonly used to model and solve optimizations problems. Recently, a new variable called the sequence variable has been introduced, primarily for modeling routing problems such as the Dial-a-Ride Problem (DARP) and the Patient Transportation Problem (PTP) [27].

The objective of this master's thesis is to investigate the potential of using the sequence variable to efficiently model scheduling problems and find optimal solutions. To this end, we will explore whether this variable can improve the modeling capabilities and performance of Constraint Programming in solving the Job-Shop problem.

# Acknowledgements

First and foremost, we would like to express our gratitude to Professor Pierre Schaus, who has been our supervisor throughout this year. He provided us with valuable guidance during the problem modeling process.

We would also like to extend a heartfelt thank you to Augustin Delecluse and Charles Thomas, who assisted us throughout this entire master thesis. They always made themselves available to help us whenever we encountered difficulties.

We would also like to express our gratitude to Roger Kameugne, who attended the launch of our thesis, agreed to review it, and was present at our defense.

Furthermore, we would like to thank our parents, relatives and friends who have been there to support and encourage us during challenging times. Special thanks go to Xavier Legrand, who provided us with a high-performance server, enabling us to collect all the necessary data.

# Contents

**Abstract** ii

**Acknowledgements** iii

**1 The Job-Shop problem** 1
- 1.1 Description . . . 1
- 1.2 Notations . . . 2
- 1.3 Time considerations . . . 3
  - 1.3.1 Task time-related attributes . . . 3
  - 1.3.2 Precedence between tasks . . . 5
- 1.4 Task, machine and job requirements . . . 5
- 1.5 Solution . . . 6

**2 Overview of Constraint Programming** 7
- 2.1 The Constraint Programming paradigm . . . 7
- 2.2 Model . . . 8
  - 2.2.1 Introducing variables, domains and constraints . . . 8
  - 2.2.2 CSP, COP and Solution set . . . 10
  - 2.2.3 Consistency and Filtering . . . 11
  - 2.2.4 Variable types examples . . . 16
  - 2.2.5 Scheduling constraints examples . . . 16
- 2.3 Search . . . 17
  - 2.3.1 Search concepts . . . 18
  - 2.3.2 Branching procedure – Building the search tree . . . 19
  - 2.3.3 Branch and prune . . . 21
  - 2.3.4 Optimization – Branch and bound . . . 23
  - 2.3.5 Large Neighborhood Search (LNS) . . . 25

**3 Overview of the sequence variable** 27
- 3.1 The sequence variable data-structure . . . 27
  - 3.1.1 A state for each node . . . 28

3.1.2 The state of the sequence . . . 29
3.2 Implementation and operations on the sequence variable . . . 30
3.2.1 Sequence State Implementation . . . 30
3.2.2 Sequence variable operations . . . 31
3.3 Related work . . . 32

**4 Job-Shop model with sequence variable 33**
4.1 Variables . . . 33
4.1.1 Machines . . . 33
4.1.2 Time windows . . . 34
4.1.3 Shared *begin* and *end* nodes . . . 34
4.2 Constraints . . . 35
4.2.1 Job and machine precedence . . . 35
4.2.2 Channeling constraint . . . 35

**5 Branching procedures for the Job-Shop modelled with sequence variables 41**
5.1 Template branching . . . 42
5.1.1 Variable and value selection . . . 42
5.1.2 Branching procedure interface . . . 42
5.1.3 The template algorithm . . . 42
5.2 Branching heuristics . . . 44
5.2.1 First-fail branching . . . 44
5.2.2 Maximum duration branching . . . 45
5.2.3 Min-max-slack branching . . . 45
5.2.4 Regret MVI branching . . . 52
5.3 Ordering the branches . . . 53
5.4 Sequence selection . . . 54

**6 Large Neighborhood Search 57**
6.1 Initial search . . . 57
6.1.1 Adapting *SetTimes* . . . 58
6.1.2 Branching procedure . . . 58
6.2 Relaxation heuristics . . . 60
6.2.1 Relaxation Random . . . 60
6.2.2 Relaxation Job . . . 61
6.2.3 Relaxation Machine . . . 61
6.2.4 Relaxation Order on Job and Order on Machine . . . 62
6.2.5 Relaxation Critical Path . . . 62
6.2.6 Shaw Relaxation . . . 64
6.2.7 Relaxation Alternate . . . 66

**7 Results and analysis** **67**
7.1 Experimental setup and Job-Shop instances . . . . . 67
7.1.1 Setup . . . . . 67
7.1.2 Job-Shop Instances . . . . . 67
7.2 Assessment models and searches . . . . . 67
7.3 Constraints comparison . . . . . 69
7.4 DFS comparison . . . . . 71
7.4.1 Branching order comparison . . . . . 72
7.4.2 Sequence selector . . . . . 74
7.4.3 Final comparison . . . . . 77
7.5 LNS comparison . . . . . 81
7.5.1 First solution . . . . . 81
7.5.2 Relaxation comparison . . . . . 82

**8 Conclusion** **101**
8.1 Future works . . . . . 102
8.1.1 Channeling constraint . . . . . 102
8.1.2 Shaw relaxation criterion . . . . . 102
8.1.3 Relaxation alternate . . . . . 102
8.1.4 Single insertion branching procedure . . . . . 102

# Chapter 1

# The Job-Shop problem

## 1.1 Description

The **Job-Shop** problem [13] is a scheduling problem where a set of $n_j$ **jobs** have to be scheduled on a set of $n_m$ shared resources known as **machines**. **Tasks** (also named **activities** in the literature) are events or units of work with fixed processing time called **duration** that belong to a job and are processed on a specific machine. The objective of the Job-Shop problem is to minimize the total time required to process all the tasks, given a complete schedule. This process time is referred to as the **makespan**.

**Example** To illustrate, figure 1.1 shows a simplified example of a Job-Shop problem and a possible schedule. The goal in this toy example is to produce three cars of different types : classic, minivan and sport.

On the left side, we can see the jobs describing the steps of the construction for each car. These steps are the tasks of the problem. Each job needs to perform the same set of tasks that are *painting*, *battery charge* and *mechanical work*. Each job has its own order of tasks and for the sake of simplicity, all tasks have a duration of one time-unit in this small example.

On the right side is displayed a schedule of the tasks. Tasks (steps) of the same kind are performed on the same machine. As we will develop later, this schedule is a solution of the problem, since the order in which the tasks of the same job are performed is respected. The makespan, the time to perform the whole schedule, is depicted too.

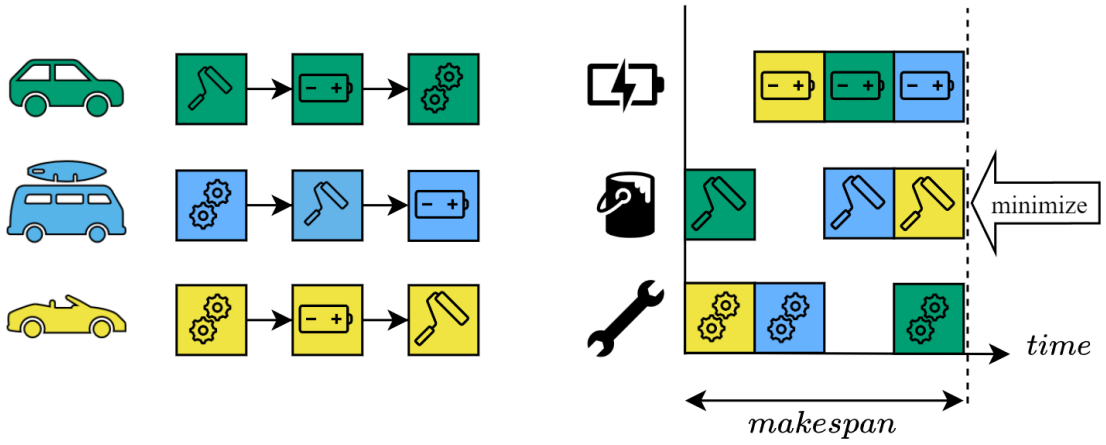



Figure 1.1: Toy example of a Job-Shop problem

## 1.2 Notations

We will now introduce some notations in order to describe the problem more formally. The set of $n_j$ jobs denoted $J$ is defined as the set of all individual job $J_j$ with $j$ the identifier of the job (eq. 1.1).

$$J \equiv \{J_j \mid j \in \{1, \ldots, n_j\}\} \tag{1.1}$$

Similarly, the set of $n_m$ machines denoted $M$ is defined as the set of all individual machine $M_m$ with $m$ the identifier of the machine (eq. 1.2).

$$M \equiv \{M_m \mid m \in \{1, \ldots, n_m\}\} \tag{1.2}$$

A Task can be represented by $\langle j, oj, m, om \rangle$ where $j$ is the identifier of the job for which the task is requested, $oj$ is the order of the task on the job, $m$ is the identifier of the machine on which the task must be processed and $om$ is the order of the task on the machine. A task can be uniquely identified by the pairs $(j, m)$, $(j, oj)$ or $(m, om)$. We therefore introduce three notations for individual tasks :

- $T_{j,m}$ that refers to the task of job $J_j$ on machine $M_m$.
- $T_{Jj,oj}$ that refers to task of job $J_j$ with job order $oj$.
- $T_{Mm,om}$ that refers to task on machine $M_m$ with machine order $om$.

The set of Task of a job $J_j$ and machine $M_m$ are denoted $T_{Jj}$ and, $T_{Mm}$ respectively (eq. 1.3).

$$T_{Jj} \equiv \{T_{Jj,oj} \mid oj \in \{1, \ldots, n_m\}\}, T_{Mm} \equiv \{T_{Mm,om} \mid om \in \{1, \ldots, n_j\}\} \tag{1.3}$$

With these notations, we can reformulate the representation of the example above. This reformulation is given in figure 1.2.

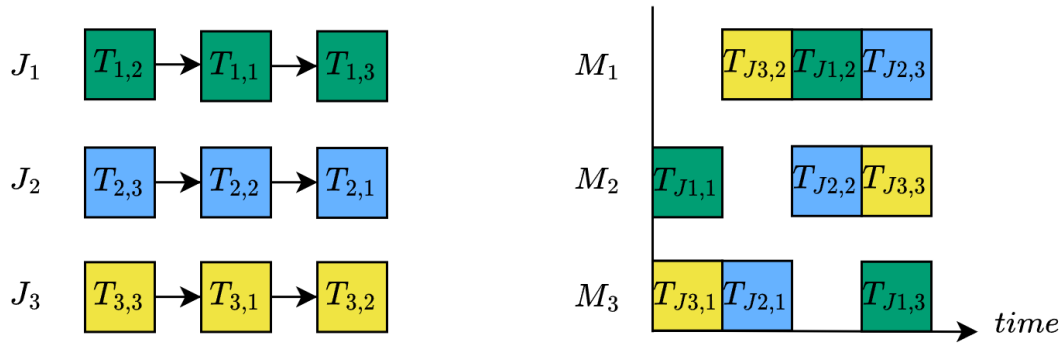



Figure 1.2: Toy example of a Job-Shop problem with notations

## 1.3 Time considerations

### 1.3.1 Task time-related attributes

To reason on the schedules, we will denote by **start** and the **end** of a task the time-step at which the task starts to be processed and the time-step at which it ends. Please note that a task can end at a time-step while another task starts from that same time-step.

The set of possible values for the start of a task defines it's **start-window**.

In order to express the time-steps spanned by an activity among all its possible schedules, we define the **execution-window** as the set of time-steps at which the task could take place.

**Illustration** Assume a machine with two tasks $T_1$ and $T_2$, where $T_1$ must precede $T_2$. Additionally, let's assume that the activities cannot be processed after time-step 15. We illustrate this example on figure 1.3 where the definitions of start, end, start-window and execution-window of a task are exemplified.

To visualize the relation-ship between execution-window and start-window, both activities are displayed at their earliest schedule and at their latest schedule given the precedence constraint.

As the duration of tasks are fixed, the execution-window can be defined in terms of the start-window. For instance, the start-window of $T_1$ spans the range [0, 8] and the task has a duration of 3. The execution-window is consequently the [0,11] range.

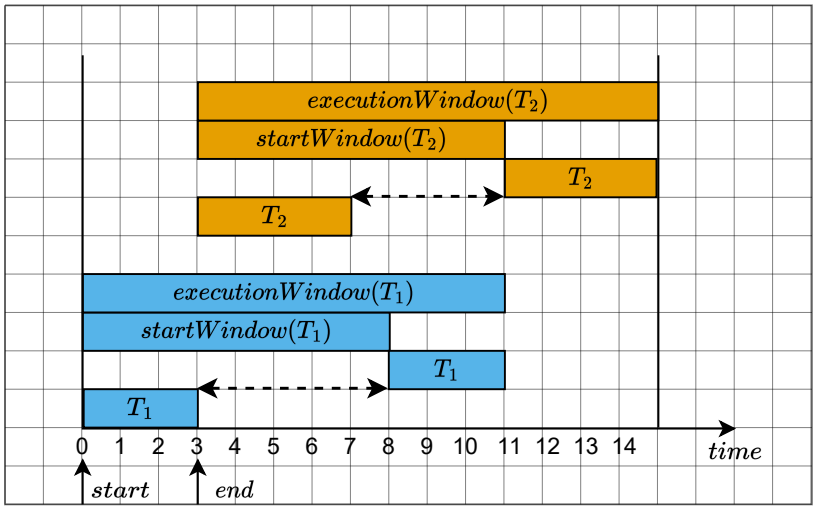



Figure 1.3: Start, end, start-window and execution-window illustration

Finally, we will introduce four more time-attributes that are usually employed together with the start and execution windows. They are depicted in figure 1.4.

- Earliest start time (est)
- Latest start time (lst)
- Earliest completion time (ect)
- Latest completion time (lct)

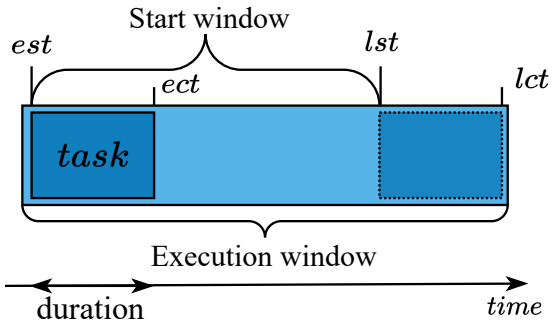



Figure 1.4: Est, lst, ect and lct illustration

### 1.3.2 Precedence between tasks

**Precedence** between tasks is an important concept in scheduling problems. In the Job-Shop problem, jobs being ordered sequences of tasks, there are precedence between the tasks of the same job.

In more detail, if a task $T_1$ has a precedence over another task $T_2$, the task $T_2$ does not start before $T_1$ has finished. All possible values spanned by the execution-window of $T_1$ are therefore smaller or equal than some value of the start-window of $T_2$. The impact of the precedence on the execution-window is also depicted on figure 1.3.

Precedences can either be defined by the problem definition such as those between the tasks of a same job, but they can also be generated during the search procedure.

## 1.4 Task, machine and job requirements

We are now able to summarize the requirements of the problem regarding the task, machines and jobs.

- Each task must be performed on a specific machine.
- Tasks can not be interrupted once they are started (eq. 1.4). They have a known fixed duration.

$$\forall\ j \in \{1, \ldots, n_j\}, \forall\ m \in \{1, \ldots, n_m\} : \\ end(T_{j,m}) = start(T_{j,m})\ +\ duration(T_{j,m}) \quad (1.4)$$

- A machine can perform only one task at a time. It remains idle if no task is scheduled on it (eq. 1.5).

$$\forall\ M_m \in M, \forall\ om \in \{1, \ldots, n_j - 1\} : end(T_{Mm,om}) \leq start(T_{Mm,om+1}) \quad (1.5)$$

- Jobs should have one and only one task associated with each machine.
- Tasks of the same job can't happen concurrently (eq. 1.6).

$$\forall\ J_j \in J, \forall\ oj \in \{1, \ldots, n_m - 1\} : end(T_{Jj,oj}) \leq start(T_{Jj,oj+1}) \quad (1.6)$$

## 1.5 Solution

Finding solutions for the problem is quite trivial. Any ordering of tasks on machines can be a solution of the problem, as long as the start and end times of the schedule meets the requirements. However, all solutions are not equivalent, since the makespan they define can greatly vary. Finding the most optimal solution with minimal makespan is what makes this problem a challenging one. It has been proven in [8, 3] that this problem is in fact NP-hard.

**Critical path** Numerous scheduling problems describe what we call a **critical path** in their solution [14]. This critical path usually describes the set of elements in a schedule that dictates the minimal processing time to complete the schedule. For the Job-Shop, the critical path is the set of **critical tasks** which are tasks that have their start-window reduced to a single value, i.e., that are fully fixed in the schedule. Figure 1.5 gives a representation of the critical path for a Job-Shop solution example. We can see that, contrary to what the name suggests, the critical path is not a single sequence of tasks but a direct acyclic graph. Between the tasks noted by $\sigma$ and $\lambda$, two paths of critical tasks are defined. As seen in the requirements section 1.4, the start-window of a node is generally linked with four other tasks : predecessor and successor of both Job and Machine. It can happen that a task constrain both successors ($\sigma$) and/or predecessors ($\lambda$).

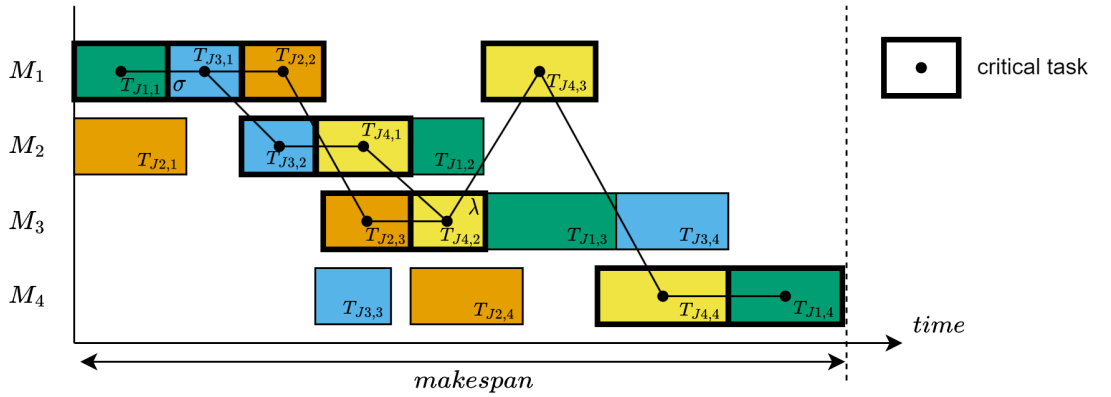



Figure 1.5: Representation of the critical path for an unoptimized solution of a Job-Shop instance with four machine and four jobs. The lines between the critical tasks show the predecessor-successor relations.

# Chapter 2

# Overview of Constraint Programming

## 2.1 The Constraint Programming paradigm

Constraint programming (CP) is a paradigm used to solve difficult, often NP-hard problems. It is employed in various problem categories, including routing problems and scheduling problems.

Problems can be optimization (minimization or maximization) problems, or not. For instance :

- *With optimization* – The Travelling Salesperson Problem (TSP) [12] problem is a famous problem where one has to find the shortest cycle in a graph passing through all nodes.
- *Without optimization* – Finding a feasible schedule for the exams of a university.

These two problem categories feel different and are actually modelled and resolved differently. The former will be modelled as a Constraint Optimization Problem (COP) whereas the second will be modelled as a Constraint Satisfaction Problem (CSP). Intuitively, when searching for an exam schedule, we are interested in some or all solutions, whereas the TSP problem implies finding the best cycle. CSPs and COPs definitions as well as the differences in their resolution process are detailed in this chapter.

**Model and Search** Constraint programming follows a declarative programming approach as it separates problem modelling, how the problem is expressed, and problem-solving, how the search towards solutions is performed. This separation

allows maintaining generality in the choice of solving techniques. If one defines an efficient algorithm for a problem, it is likely to be reusable in similar problems.

In the next two sections, we will lay out what is behind those two parts and how they are connected.

## 2.2 Model

In order to understand the modelling approach of CP, we will first introduce in section 2.2.1 the basic building blocks of a CP model : variables, domains and constraints. Follows section 2.2.2 that gives a more formal definition of the problem (CSP/COP model) and the solution set, based on the definitions of the former section. Section 2.2.3 will afterward define the notions of inconsistency in a CSP, as well as the filtering algorithms of constraints who are closely related to it. Finally, section 2.2.4 will elaborate on the different types of variable, whereas section 2.2.5 will introduce some typical constraints used in scheduling problems.

### 2.2.1 Introducing variables, domains and constraints

#### Intuition

To understand how problems are modelled, let's take the example of the *Sudoku* problem. As a reminder, in the *Sudoku* problem the goal is to assign a value to each cell such that the values in a row/column/sub-square are all different. The value of a cell lies between 1 and 9 included.

This small example allows us to already introduce the three main entities that will be used to model the problem :

- The cells are modelled by *variables*.
- The values a variable can take, initially the 1 to 9 range in the *Sudoku* problem, define the *domain* of that variable.
- The row/column/sub-square requirements define the *constraints* of the problem. One constraint is added to the model or **posted**, for each row/column/sub-square. Each constraint enforces that all the variables it covers take a different value when assigned.

#### Definitions

More formally, we can give the following definitions :

**Definition 2.1** (Variable)**.** A variable $v$ is a placeholder for a value. The set of values that a variable $v$ can take defines its domain, noted $D_v$. If the domain of a variable is reduced to a singleton, the variable is said to be **fixed** or assigned to that value. The action of reducing the domain of a variable to the empty set is called a **domain wipe-out**.

**Definition 2.2** (Domain)**.** A domain $D_v$ associated with a variable $v$ defines the set of data or values $d$ such that the assignment $v = d$ is allowed. We assume the domains are totally ordered, $\min(D_v)$ and $\max(D_v)$ representing the maximum and the minimum value in $D_v$. Furthermore, we will denote by $D_V$ the set of domains associated with a set of variables denoted $V$.

Formally, on eq. 2.1.

$$D_V \equiv \{D_v \mid v \in V\} \tag{2.1}$$

**Definition 2.3** (Assignment)**.** An assignment of a variable $v$ to value $d$, represented as a couple $\langle d, v\rangle$, is the fact of reducing $D_v$ to the singleton $d$. $\langle d, v\rangle$ is a valid assignment only in case $d$ belongs to $D_v$.

We denote by $A_V$ a tuple of valid assignments made of a single assignment for each variable contained in a set of variables $V$.

Formally, on eq. 2.2.

$$\begin{aligned} A_V \equiv \langle \langle d_1, v_1\rangle, \ldots, \langle d_{|V|}, v_{|V|}\rangle\rangle \mid \\ (\forall\ \langle d, v\rangle \in A_V : v \in V\ \ \&\ \ d \in D_v)\ \ \&\ \ (\nexists\ \langle d_i, v_i\rangle, \langle d_j, v_j\rangle \in A_V : v_i = v_j) \end{aligned} \tag{2.2}$$

When talking about a tuple of valid assignments $A_V$, it is implicit that each variable $v_i$ of $V$ is present once and only once in the tuple by the presence of a couple $\langle d_i, v_i\rangle$. By extension, for a constraint $c$, $A_c \equiv A_{V_c}$.

**Definition 2.4** (Constraint)**.** A constraint $c$ is a couple $\langle P_c, V_c\rangle$ where $V_c$ is the set of variables associated with $c$ and $P_c$ is a predicate over $V_c$. The constraint is said to be satisfied for a set of valid assignments $A_c$ if $A_c$ is such that the predicate $P_c$ is true when $A_c$ holds.

The satisfaction of a constraint $c$ can be defined formally as in eq. 2.3, where $P_c(A_c)$ represents the evaluation of the predicates $P_c$ of $c$ considering the assignments of $A_c$ are performed.

$$c(A_c) \Longleftrightarrow P_c(A_c) \tag{2.3}$$

### Illustration

In order to give a practical example of the concepts of variable, domain and assignment, we display in table 2.1 an example of a line of a *Sudoku* problem (no other line is considered).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 2 | | 1 | 5 | 6 | 3 | | 8 |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ |

Table 2.1: Sudoku example for variable, domain and assignment illustration

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4,7,9 | 2 | 4,7,9 | 1 | 5 | 6 | 3 | 4,7,9 | 8 |
| $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ | $v_8$ | $v_9$ |

Table 2.2: Domain representation for the Sudoku variables

Regarding the variables, we can simply define 9 variables from $v_1$ to $v_9$, one for each cell. On table 2.2 are displayed the domains of the above-defined variables.

The domain of a variable is either the value to which the variable is fixed in the *Sudoku* or the possible values that are still allowed for a non-fixed variable. As cell $x_2$ is fixed to 2, the domain of the associated variable $v_2$ only contains 2. The cell $x_3$ is not filled, the domain of variable $v_3$ therefore contains the remaining possible values : 4, 7 and 9.

A valid assignment of the problem would be $\langle\ \langle 4, v_1\rangle, \langle 2, v_2\rangle, \langle 7, v_3\rangle, \langle 1, v_4\rangle, \langle 5, v_5\rangle, \langle 6, v_6\rangle, \langle 3, v_7\rangle, \langle 9, v_8\rangle, \langle 8, v_9\rangle\ \rangle$. Indeed, the given assignment is valid as the constraint (that requires that all values of the line are different) is satisfied for this example of assignment.

### 2.2.2 CSP, COP and Solution set

The *Sudoku* problem itself and the set of solutions it admits can also be described in a more formal way using the three components described above. This representation defines the problem as a Constraint Satisfaction Problem, abbreviated CSP.

**Definition 2.5** (Constraint satisfaction problem (CSP))**.** A CSP is a triplet $\langle V, D_V, C\rangle$ where $V$ is a set of variables, $D_V$ is the set of domains associated with the variables in $V$, and $C$ is a set of constraints over a subset of $V$.

Looking at the TSP problem introduced in the introduction of this chapter, we need to extend the definition of CSP. Optimization problems have their fundamental difference with CSPs in the fact that they possess an objective function that for each solution defines how good this solution is.

**Definition 2.6** (Constraint Optimization Problem (COP))**.** A COP is a quadruplet $\langle V, D_V, C, O\rangle$ where $V$ is a set of variables, $D_V$ is the set of domains associated with the variables in $V$, $C$ is a set of constraints over a subset of $V$ and $O$ is the objective function of the COP.

The objective function is a method that for each solution $s \in \mathcal{S}$ associates a value that represents the quality of the solution. We take here the convention

of a minimization problem, where lower objective values denote better solutions. Conversely, higher objective functions denote better solutions for maximization problems.

As the COP model is a direct extension of the CSP model, almost all concepts applicable to CSPs are applicable to COPs too. All definitions in the next sections that are defined for a CSP can therefore be applied to COPs. Section 2.3.4 will detail how the search can be made differently for COPs, while the CSP search can still be applied on a COP instance.

In order to support upcoming definitions, we define hereunder the set of solutions accepted by a CSP.

**Definition 2.7** (Solution set)**.** The set of solutions of a CSP $\langle V, D_V, C\rangle$ is written $\mathcal{S}(\langle V, D_V, C\rangle)$ and is defined by the set of **candidate solutions** (i.e. a tuple made of assignments for all the variables of the CSP) that satisfy all the constraints.

Formally, on eq. 2.4.

$$\mathcal{S}(\langle V, D_V, C\rangle) \equiv \{A_V \mid \forall\ c \in C :\ P_c(A_V)\} \tag{2.4}$$

### 2.2.3 Consistency and Filtering

#### Inconsistency

An important notion that will be used next is Inconsistency.

**Definition 2.8** (Inconsistent CSP)**.** An inconsistency in a CSP means that either a constraint is violated or not satisfiable, or a variable's domain has been wiped-out. In all cases, the current CSP can not lead to any solution ($\mathcal{S}(CSP) = \emptyset$) as reducing the domains, which is how the search works as we will see, can not lead a constraint to be re-satisfied again once it has been violated.

To be clear, a consistent state does not mean, per see, that all values of the domains of the state will be chosen to be part of a solution during the search at some point. Impossible, *unnoticed* values may still remain. If the CSP is said consistent, it only means the constraints did not detect any inconsistency.

#### Constraint filtering

So far, the constraints have been seen as predicates on a candidate assignment ($A_c$ for a constraint $c$). Constraints can be used as such to assess the validity of a candidate solution ($A_V$ for a CSP $= \langle V, D_V, C\rangle$) leading to a solution in case the constraints are indeed satisfied.

In fact, constraints are further useful when all variables are not yet fixed. They are associated with a filtering algorithm that is used to *filter out* invalid values (i.e., values that will not be part of any solution) from the domains of its associated variables and detect inconsistent states to some extent.

**Definition 2.9** (filtering algorithm)**.** A filtering algorithm or **constraint propagator** of a constraint $c$ noted $F_c$ is a function that removes from the domains of its variables $D_{V_c}$ values that are known to not lead to any valid solution. A filtering algorithm should therefore never reduce the set of solutions. This property is described formally in eq. 2.5.

$$\forall\, c \in C :\ \mathcal{S}(\langle V, D_V, C\rangle) = \mathcal{S}(\langle V, F_c(D_V), C\rangle) \tag{2.5}$$

Besides the removal of invalid values, the filtering algorithm may detect the current state is invalid regarding its constraint.

We say a constraint is **propagated** or scheduled for propagation when a constraint is asked to apply its filtering on its related variable domains. Constraint propagation is a central component in the search procedure, as it allows detecting inconsistent states without waiting for all variables to be fixed. We will see when constraints are propagated in more extent in section 2.3.3.

Filtering algorithms vary in their filtering power and their cost (computational complexity). The ideal filtering would filter much at small cost. To remain formal, we introduce in the next section two definitions used to characterize the filtering power of a constraint.

### Domain and bound consistency

The filtering power of a propagator $F_c$ of a constraint $c$ is expressed by the consistency (roughly speaking, the degree of filtering) it can obtain for $D_{V_c}$. A constraint propagator with a certain degree of consistency is a constraint propagator that will enforce a that degree of consistency on the domains of the variables associated with its constraint.

To be able to define more easily the properties of domain and bound consistency, we will beforehand define the notions of *support* inspired from [17].

**Domain support** In a nutshell, a value associated with a variable has a domain support for a constraint if this value for this variable is part of a tuple of valid assignments that satisfy the constraint.

**Definition 2.10** (Domain support)**.** Given a constraint $c$, a variable $v \in V_c$ and a value $d \in D_v$, $d$ is said to have a domain support in $c$ if there exists a tuple of *valid* assignments $A_c$ which contains $\langle d, v\rangle$ and satisfies $c$.

Domain support is defined formally in eq. 2.6 using the definition 2.3 of $A_c$. Recall the following property : $\forall\ \langle d_i, v_i\rangle\ \in A_c : v_i \in V_c\ \ \&\ \ d_i \in D_{v_i}$.

Given a constraint $c$, a variable $v \in V_c$ and a value $d \in D_v$,

$$d \text{ has a domain support in } c \iff \exists A_c \mid (P_c(A_c)\ \ \&\ \ (\langle d, v\rangle\ \in A_c)) \quad (2.6)$$

**Interval support** The definitions of interval support is a weaker version of domain support. It does not require the assignments to be valid, but rather that the assigned values must lie between the upper and lower bounds of the domain of the variable they are related to.

**Definition 2.11** (Interval support)**.** Given a constraint $c$, a variable $v \in V_c$ and a value $d \in D_v$, $d$ is said to have an interval support in $c$ if there exists a tuple of assignments $A_c^*$ which contains $\langle d, v\rangle$ and that satisfies $c$ such that for all $\langle d_i, v_i\rangle\ \in A_c^*$, $d_i$ lies in the $[min(D_{v_i}),\ max(D_{v_i})]$ interval.

Formally, on eq. 2.7.

$$\begin{aligned}\text{Given } A_c^* \equiv \langle\langle d_1, v_1\rangle, \ldots, \langle d_{|V_c|}, v_{|V_c|}\rangle\rangle \mid \\ (\forall\ \langle d_i, v_i\rangle \in A_c^* : v_i \in V_c\ \ \&\ \ d_i \in [min(D_{v_i}),\ max(D_{v_i})]) \\ \&\ \ (\nexists\ \langle d_i, v_i\rangle, \langle d_j, v_j\rangle \in A_c^* : v_i = v_j), \\ d \text{ has an interval support in } c \iff \\ \exists A_c^* \mid (P_c(A_c^*)\ \ \&\ \ (\langle d, v\rangle\ \in A_c^*)\end{aligned} \quad (2.7)$$

The definition of $A_c^*$ is very close to the definition of $A_c$. The only difference lies in the values that $d_i$ can take. $A_c$ enforces the values to be in the domain, whereas $A_c^*$ only enforces them to be within the bounds of the domain.

**Domain and Bound consistency** Using the definitions of domain and interval support just defined, we can now define the notions of domain and bound consistency.

**Definition 2.12** (Domain consistency)**.** A domain-consistent filtering algorithm $F_c$ associated with a constraint $c$ is an algorithm that filters the domains such that any remaining value has a domain support in $c$.

Formally, on eq. 2.8.

$$\begin{aligned}F_c \text{ is domain consistent} \iff \\ \forall\ x \in V_c, \forall\ d \in D_{V_c} :\ d \text{ has a domain support in } c, \\ \text{after having propagated } F_c\end{aligned} \quad (2.8)$$

| 1, 2, 3, 4, 5 | 2 | 1, 2, 3, 4, 5 | 1 | 1, 2, 3, 4, 5, 6 | 6, 7, 9 | 3 | 6, 7, 9 | 6, 7, 9 |
|---|---|---|---|---|---|---|---|---|
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ |

Table 2.3: Sudoku example for consistency – Initial state

**Definition 2.13** (Bound consistency)**.** A bound-consistent filtering algorithm $F_c$ associated with a constraint $c$ is an algorithm that filters the domains such that, for each variable $v$, $min(D_v)$ and $max(D_v)$ have an interval support in $c$.

Formally, on eq. 2.9.

$$F_c \text{ is bound consistent} \iff \forall\, x \in V_c : min(D_v) \text{ and } max(D_v) \text{ have an interval support in } c, \text{ after having propagated } F_c \quad (2.9)$$

**Illustration** To illustrate the concepts of bound and domain consistency, we will take again the example of the *Sudoku* problem regarding the AllDifferent constraint that applies on each line. To be clear, let us at first define the AllDifferent constraint more rigorously.

**Definition 2.14** (AllDifferent constraint)**.** The AllDifferent constraint is a constraint $c$ that is applied on a set of variables $V_c$ (usually integer variables). It enforces that for a tuple of valid assignments $A_c$, each of the variables associated with it take a different value.

The predicate of the AllDifferent constraint can be defined formally as in eq. 2.10.

$$P_{\text{AllDifferent}}(A_{\text{AllDifferent}}) \equiv \forall\, \langle d_i, v_i\rangle, \langle d_j, v_j\rangle \ \in A_{\text{AllDifferent}} \mid v_i \neq v_j : d_i \neq d_j \quad (2.10)$$

On table 2.3 is displayed a row of a *Sudoku* problem. Some external constraints were applied that already constrained the domains of some variables. Depending on the consistency of the filtering procedure, a different set of impossible values will be discarded.

**A simple filtering** One way to implement the AllDifferent constraint is to perform a *binary decomposition,* which implies defining the *AllDifferent* constraint as a set of *NotEqual* constraints. As the name suggest, the NotEqual constraint ensures that two variables take a different value in all solutions.

Formally, on eq. 2.11.

$$P_{\text{NotEqual}}(\{\langle d_i, v_i\rangle, \langle d_j, v_j\rangle\}) \equiv d_i \neq d_j \quad (2.11)$$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ~~1, 2, 3,~~ 4, 5 | 2 | ~~1, 2, 3,~~ 4, 5 | 1 | ~~1, 2, 3,~~ 4, 5, 6 | 6, 7, 9 | 3 | 6, 7, 9 | 6, 7, 9 |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ |

Table 2.4: Sudoku example for consistency – After binary decomposition pruning

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4, 5 | 2 | 4, 5 | 1 | ~~4, 5,~~ 6 | ~~6,~~ 7, 9 | 3 | ~~6,~~ 7, 9 | ~~6,~~ 7, 9 |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ |

Table 2.5: Sudoku example for consistency – After bound consistent pruning

The filtering algorithm that is obtained by this manner ensures that all pairs of variables of the constraint take a different value. In fact, this filtering is the minimal filtering required to ensure the predicate.

Performing such filtering on table 2.3 we obtain table 2.4. We observe that the values 1,2 and 3 who were already used for some fixed variable have been removed from the domains of the unfixed variables.

Regarding the consistency, this filtering is actually even weaker than bound consistency ! To be bound consistent, the bounds of the domain of each variable must have an interval support. If we take the variable $x_6$ which has a minimum value of 6 in its domain, this value (for this variable) does not have an interval support. Said differently, there does not exist a tuple of assignments containing $x_6 = 6$ while $c$ is satisfied and the variables are assigned to a value that lies between the upper and lower bound of their domain.

**Bound-consistent filtering** If one performs a bound-consistent filtering, he would obtain the table 2.5. It is obtained by making the following deductions using bound consistency.

- $min(x_5) = 4$ has no interval support, so it is filtered out.
- Once 4 is removed, 5 the new $min(x_5)$ and has no interval support, it can therefore be filtered too.
- As 6 is the last possible value of $x_5$, it can be removed from the domains of $x_6$, $x_8$ and $x_9$.

**Domain-consistent filtering** Finally, if one would perform a domain-consistent filtering, he would detect that the state is actually inconsistent. As 8 is not contained in any domain, no value in the current stats of the problem has an interval support. In other words, there is no valid tuple of assignments that satisfies the constraint.

The reason why the state was considered bound consistent comes from the fact that interval support requires the assignment to have values that are between the minimum and maximum of the domain of the variable they are assigned to. 8 could in that case be chosen for the value of $x_6, x_8$ or $x_9$. This is not allowed in domain support, preventing domain consistency satisfaction.

### 2.2.4 Variable types examples

There are different types of variables in CP, for instance integer variables and boolean variables. They can be distinguished by what their domain contain and the way to act upon it. Integer variables have a domain made of a set of integer values. The domain of boolean variables is made of the {true, false} set.

There are also variables representing more complex entities, such as sequence variables that represent sequences of elements. As sequence variables have a significant importance in the problem addressed by this thesis, they are described in more detail in a dedicated part : chapter 3.

### 2.2.5 Scheduling constraints examples

In this section, we present two commonly used constraints in scheduling problems.

#### LessOrEqual constraint

We introduced in section 1.3.2 the notion of precedence between tasks. As we saw, the precedence constraint between two tasks yields that the execution-window of the former task must end before (or at the same time step as) the start of the execution-window of the latter. This is in practice modelled by the *LessOrEqual* constraint [18].

**Definition 2.15** (LessOrEqual constraint)**.** The LessOrEqual constraint is a constraint $c$ defined between two integer variables $v_i$ and $v_j$ (order matters) that imposes that for a tuple of valid assignments $A_c = \{\langle d_i, v_i\rangle, \langle d_j, v_j\rangle\}$, the value taken by variable $v_i$ ($d_j$) is lower or equal to the value taken by variable $v_j$ ($d_j$).

The predicate of the LessOrEqual constraint can be defined formally as in eq. 2.12.

$$P_{\text{LessOrEqual}(v_i, v_j)}(\{\langle d_i, v_i\rangle, \langle d_j, v_j\rangle\}) \equiv d_i \leq d_j \tag{2.12}$$

#### Disjunctive constraint

Another usual constraint for scheduling problems is the *disjunctive* constraint [18]. This constraint is defined for a set of tasks which have a fixed duration and enforces that the execution-windows of these tasks do not overlap.

**Definition 2.16** (Disjunctive constraint)**.** The disjunctive constraint is a constraint $c$ defined for a set of start variables $V_c$ and an array of durations *duration* associated with them. The constraint enforces that for a tuple of valid assignments $A_c$, the values are such that no execution-window overlaps. In other words, for any pair of tasks $v_i$ and $v_j$, either the task associated with $v_i$ must end before the task associated with $v_j$ starts or the reverse predicate must hold.

More formally, the disjunctive constraint can be defined as in eq. 2.13

For a constraint Disjunctive($V_{Disjunctive}$, *duration*) written $c$ for simplicity,

$$P_c(A_c) \equiv \forall \langle d_i, v_i \rangle, \langle d_j, v_j \rangle \in A_c \mid v_i \neq v_j : \\ (d_i + duration_i \leq d_j) \vee (d_j + duration_j \leq d_i) \quad (2.13)$$

**Filtering algorithms** The filtering algorithm defined for the disjunctive constraint in [18] is quite developed. It uses namely the following filtering algorithms : OverLoadCheck, Detectable Precedence, Not-Last algorithm and Edge-Finding.

Still, there exist other filtering algorithms that can also be used. For instance, the disjunctive can be modelled by performing a binary decomposition. In short, the binary decomposition can be viewed as posting for all pairs of variables $(v_i, v_j)$ related to the variables of the disjunctive constraint ($V_c$) a *binary disjunctive* whose behavior is described in eq. 2.14.

The binary disjunctive has the same requirement as the disjunctive restricted to two variables. It enforces that the time-windows of the tasks associated with $v_i$ and $v_j$ don't overlap.

$$P_{DisjunctiveBinary(v_i,v_j)}(\langle d_i, v_i \rangle, \langle d_j, v_j \rangle) \equiv \\ (d_i + duration_i \leq d_j) \vee (d_j + duration_j \leq d_i) \quad (2.14)$$

## 2.3 Search

As the name suggest, the search is the fact of searching for solutions using the model of the problem. The way this is done in CP is detailed in this chapter.

The structure of this chapter is outlined as follows. In section 2.3.1 we detail the abstract concepts of *problem state*, *branching choice*, *search space* and *search tree*, required to depict the concept of *search* itself. Afterward, we develop in section 2.3.2 how the decisions are made during the search, which defines the search tree.

Section 2.3.3 will combine the previous parts by detailing the search procedure itself also named *branch and prune* (not to be confused with *branch and bound* which is detailed in 2.3.4). Follows section 2.3.4 that will develop an extension of

the search procedure in case the problem appears to be an optimization problem (COP). Finally, section 2.3.5 will introduce Large Neighborhood Search which is another way of performing the search that trades optimality, i.e., if the search completes it is guaranteed to return the best solution, for the ability to scale with large search trees.

### 2.3.1 Search concepts

We define in the next subsection some important concepts related to the search of CP. Those concepts are briefly illustrated with an example afterward.

#### Definitions

**Definition 2.17** (Search space)**.** We define the search space of a problem as the set of all states of the problem model that may exist given the initial conditions remain fixed.

**Definition 2.18** (Search tree (in CP))**.** A search-tree is a representation or model of the way one can explore the search space by making branching decisions. Nodes of the tree, also named **branching points**, are sub-problems of the initial one. Branching points often define two or more alternative branching decisions.

Leaf nodes represent a state of the problem where all branching variables of the problem are fixed, yielding a solution to the problem if no constraints are violated. The root of the tree is the initial problem. The **branching procedure** is the algorithm that defines how the search tree is build.

**Definition 2.19** (Branching decision)**.** A branching decision is the action of making an assumption on the problem. By making a decision, the search goes from a problem state to another. A branching decision is not a deduction made by a constraint. As they are assumptions or guesses, they can be wrong and lead to an invalid state, forcing the search to roll back the decision.

#### Illustration

Using the *Sudoku* problem mentioned earlier, we can make the following illustration.

**Search space** The search space of the *Sudoku* problem is defined by all the different grids that one could obtain by filling one or several cells. As a consequence, the search defines as well the intermediate states as the candidate solutions where all variables are fixed. This set contains the solution(s) we are looking for but also (many) invalid states, such as the state where all gaps would have been filled with 0's.

**Branching decision** To solve a difficult *Sudoku*, you may need to guess the value of some variable and check if there is a solution using the deductions made based on this assignment. Guessing a value is denoted by a branching decision.

Note that performing an assignment is not the only application of a branching decision. Discarding a value from the domain of a variable or posting a new constraint are other possible cases.

**Search tree** The search tree mirrors the sequence in which the speculative value assignments are done. Trying all possible values for a given cell for all the cells recursively defines a search tree for instance.

### 2.3.2 Branching procedure – Building the search tree

#### Procedure

In order to find a solution, the search performs branching decisions. Those successive branching decisions define the search tree as the search is performed. The aim of those decisions is to allow the exploration of the search space.

Some branching procedures are more suited to some problems. We define by **search heuristic** a strategy used to perform a branching. They are called heuristics as those strategies are not proven to be efficient and suited to all problems, but have been observed efficient.

To begin, we will introduce a general-purpose one, here-under referenced as *first-fail*.

#### A first-fail branching procedure

**First-fail heuristic** The first-fail strategy relies on the principle *To succeed, try first where you are most likely to fail.* [1]. One way of interpreting this heuristic is the *smallest domain first (SDF)* variable ordering heuristic [1]. As all the variables have to be assigned to a value, the SDF heuristic suggests selecting first the variables with the smallest domain size.

Another way of interpreting the first-fail principle would be to try first the variables which are included in the most constraints. Both ideas match the idea of focussing on the critical variables first.

For simplicity, we detail next the SDF branching.

**Smallest Domain First branching** The presented branching procedure selects one variable $v$, and it generates a choice point with as many branches as the variable has possibles values in its domain. If the domain is $\{a, b, c\}$, the search

will generate 3 branches from that point, each branch being the assignment of $v$ to one of the values.

This procedure is applied repeatedly until a solution is found, or an inconsistent state is encountered.

### Illustration of the first-fail branching

To clarify the branching process, let's see an example of a tree that could be build using the first-fail branching for a toy example derived from the *Sudoku* problem. To keep it simple, let's consider a table of four variables ($x_1$ up to $x_4$) with an *all-different* constraint applied on them (see definition 2.14 for a more detailed explanation of the all-different constraint). The initial domains of the variables are depicted in table 2.6.

| 5, 6, 7 | 6, 7 | 7, 8 | 7, 8 |
|---|---|---|---|
| $x_1$ | $x_2$ | $x_3$ | $x_4$ |

Table 2.6: Variables and domains for the first-fail branching illustration

We depicted on figure 2.1 a search-tree that could be generated following the first-fail strategy. Let's pinpoint some aspects of its construction.

As stated in the branching procedure, the variable with the smallest domain in selected first. The first branching-point is therefore the assignment of variable $x_2$, and the two branches outgoing from the root-node assign one value (6 or 7) to the variable.

After having performed the branching decision, an assignment here, the domains of the variables are updated to exploit the information given by the branching decision (we will detail this aspect more in depth in section 2.3.3 that comes next). The update of the domains of the variables can lead to inconsistent states, noted $I$ on the figure. In such cases, the branching will not be continued.

If the state is valid or consistent, a new branching decision can be made. After having branched on variable $x2$, we can branch on variable $x1$ (we could have chosen another variable too, as they have equal domain sizes).

Once all variables are fixed, we obtain a leaf node of the search tree as we found a solution ($S$ states on the figure). Note that all variables are not per see fixed by the search, they can be fixed due to constraint propagation.

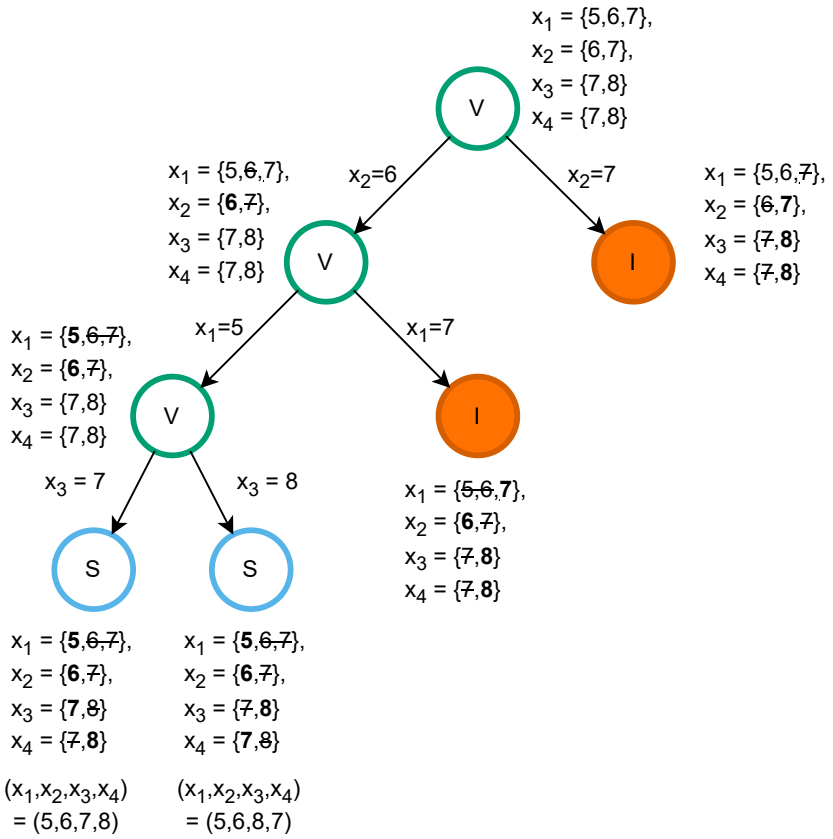



Figure 2.1: First-fail branching illustration – Removed values are crossed, fixed variables have their value in bold. I, V and S denote Invalid, Valid and Solution states respectively.

### 2.3.3 Branch and prune

**Branching, pruning and backtracking** As we defined the search tree, we are now ready to develop the search procedure itself that will walk this tree. The search process is also denoted as the *branch and prune* algorithm, which are the two main parts of the search procedure. The *backtracking* functionality is the last component of the search. Each of them are discussed in a dedicated section. A discussion on the importance of the filtering power of the constraints terminates this section about the search procedure.

#### The branch operation

The *branch* step is the fact of making some decision to move to a next state in the search tree. This step consists essentially in calling the branching procedure to

generate a new choice point of the tree and performing the first branch decision. Recursively taking the first branch is defined as a depth first tree traversal. This kind of search is therefore denoted *DFS* for *Depth First Search*. A more detailed explanation of DFS can be found in [20]. Section 2.3.5 will introduce another way of performing the search.

### The prune operation - Fixpoint algorithm

The **prune** step consists in the reduction of the domains of the variables given the new information acquired by the decision. This second step is achieved by a fixpoint computation.

**Fixpoint algorithm** The fixpoint algorithm, inspired from [18], is depicted in algorithm 1. Starting from a queue of constraints initially containing all the constraints of the problem (line 1), the algorithm repeatedly propagates constraints from the queue (line 6) and adds to the queue the constraints who's variable domains have been modified (line 8). Once the constraints are not able to reduce the domains of the variables anymore, the algorithm finishes.

**Algorithm 1** Fixpoint algorithm for a $CSP = \langle V, D_V, C \rangle$

**Ensure:** All constraint are satisfied if returns true, returns false otherwise.

```
1:  Q ← C
2:  D ← D_V
3:  try
4:     while |Q| > 0 do
5:        c ← dequeue(q)
6:        D' ← F_c(D) // compute the updated set of domains by propagating c
7:        V' ← {v_i ∈ V | D_{v_i} ≠ D'_{v_i}} // compute the updated set of variables
8:        C' ← {c' ∈ C \ Q | V_{c'} ∩ V' ≠ ∅} // compute the set of constraints not
          already present in the queue who's variable domains have been updated
9:        for c' ∈ C' do
10:          enqueue(c')
11:       end for
12:       D ← D'
13:    end while
14:    return true
15: catch { Inconsistency }
16:    return false
17: end try
```

#### Backtracking

**Backtracking procedure** In order to pursue the search when the state becomes inconsistent (modelled in algorithm 1 by the *Inconsistency* catch), the search engine needs to be able to get back to the last consistent state.

This procedure is called the **backtracking procedure** and can be implemented in different ways not detailed here.

The important point to understand is that any operation whether it is a domain reduction or a constraint addition done since the last consistent state will be erased when the backtracking is performed.

**When is backtracking performed** As stated above, inconsistent state requires the search to backtrack. Paradoxically, backtracking is also required when a valid solution is encountered. Indeed, once all values have been fixed and a solution found, we need to backtrack in case we would like to find other solutions.

#### Filtering Power trade-off

Looking at the definitions introduced in 2.2.3, it is clear that a domain consistent filtering is stronger compared to a bound consistent one. Such stronger filtering allows for a smaller search-tree, as the reduction of the variable domains is performant.

But this filtering power does not come for free. Domain consistent algorithms tend to have worse complexities compared to bound consistent algorithms. Consequently, the execution time of domain consistent filtering will usually be higher.

To summarize, there is no clear dominating method, as having a fast pruning could involve a bigger search tree and conversely. The choice, based on the total execution time, depends on the application.

### 2.3.4 Optimization – Branch and bound

So far, we mainly used the example of the *Sudoku* problem. A problem like the Travelling Salesperson Problem (see the introduction of this chapter) should not be resolved the same way. The search as defined previously would enumerate all possible cycles. While this would work fine, it would be inefficient and would not determine which is the best cycle one.

This section outlines the differences related with the resolution of such a problem. We will introduce next the concept of branch and bound at the core of the search in case of optimization problems.

**Branch and Bound (BnB)** To perform a search for optimization problems, CP relies on the concept of *Branch and bound* adapted to the CP framework. Let's first see the idea behind branch and bound before looking how it is adapted in CP.

A **branch and bound (BnB)** search algorithm maintains a *bound* whose value equals the objective value of the best solution encountered so far. The branch and bound search is expressed by two components : *branching* and *bounding*.

The *branching* step is identical from the search procedure defined for CSP's. It is responsible for generating the choice points of the search-tree.

The *bound* operation consists of the evaluation of the possible branches and the **pruning** (i.e. removal) of the branches that will not lead to a better solution. In more detail, for each branch, a lower-bound on the objective value of the solutions that can be obtained from that branch is computed. In case the computed value for a branch appears to be greater than the bound of the search, we can deduce that it is not worth exploring this branch, as all generated solutions are guaranteed to be worse (higher objective value in a minimization problem) compared to the current best solution.

**Branch and bound in CP** In CP, the *bound* is implemented as a variable whose domain is restricted during the search. Let us denote it as the *bound variable*. Each time a solution is found, the domain of the bound variable is restricted to be at most equal to the objective value of the new solution. As a consequence, any new solution or state of the search that would have a bound greater than the optimal one would generate a domain wipe-out of the bound variable, preventing the search to further explore this branch of the search tree.

It is worth to mention that the criterion for branch-pruning (the lower-bound computation) may not over-estimate the bound and discard a branch containing a solution, as the search should always be able to find the optimal solution if the search terminates.

**Solve and optimize** Let us recapitulate the two ways the search can be done.

- *Solve* – Performs a complete exploration of the search tree, the aim is to find all possible solutions of a CSP.
- *Optimize* – The objective is to find the best possible solution of a COP. The search will not explore branches when it knows for sure the solution would be worse compared to the current optimum. As a consequence, this search does not return all the solutions. To be guaranteed to have the best solution, the search needs to terminate.

### 2.3.5 Large Neighborhood Search (LNS)

**Large Neighborhood Search (LNS)** [24] is a search strategy used to solve optimization problems by applying a similar approach used by local search [21] algorithms to explore the search tree.

#### Local Search

Local search algorithms are search algorithms working by defining a neighborhood of solutions around an existing solution. At each iteration, the algorithm selects a new solution to explore among the neighborhood of the current solution. Once the solution is selected, a new neighborhood is defined and the algorithm repeats. Local search has two main phases:

- **Intensification phase** : in which the search focuses on the neighborhood close to the current solution because it is considered promising to find better solutions.
- **Diversification phase** : during which, the defined neighborhood is enlarged because the search considers that the locality of the current solution is not promising and wishes to escape it.

#### LNS neighborhood

To adapt the local search strategy to the constraint programming framework, the neighborhood of a solution is obtained by unfixing some branching variables of the model, enabling them to take other values. Or to remove constraints added during by a branching algorithm. This could be represented by the current place in the search tree be transported towards the root of the search tree. The action of unfixing variables is called to **relax** a variable. We will also use the terminology "**relaxed** solution" when talking about the solution for which some variables have been relaxed.

#### Relaxation heuristic

To get a relaxed solution, we need to define what are the variables that will be relaxed to obtain a neighborhood. This will be done with relaxation heuristics.

**Definition 2.20** (relaxation heuristic)**.** A relaxation heuristic is a method whose role it chooses variables to be relaxed. The selection performed could depend on multiple parameters such as the problem, the solution defined, other variables, the state of the search, etc. The selection can also be based on randomness.

As explained in [4], finding the promising neighborhood is as important as the branching heuristic used to do the optimization. It may be even more important on highly constrained problems, where it happens that a relaxed variable has no choice but to retake the same value because constraints with unrelaxed variables force it to do so. To define an efficient relaxation heuristic, it is therefore essential to relax together variables that share some similarities so that they can change their values together.

### LNS phases

To summarize, given an initial search providing a first solution of the problem, the LNS alternates between two phases of diversification and intensification to gradually improve the best solution found. These phases are :

- **Relaxation phase** : This is the diversification phase. In this phase, a neighborhood of the current solution is defined by relaxing the current best solution with the help of a relaxation heuristic. Relaxation is what enables the search to escape from less promising parts of the search tree by discarding some choices that a branching procedure made.
- **Optimization phase** : This is the intensification phase. In this phase, a DFS with BnB is used on the obtained relaxed solution to improve the current obtained solution. When a new solution is reached, it is registered as the new best solution.

Depending on the relaxation and the defined neighborhood, the size of the subtree to be explored during optimization could be quite large and perhaps not so promising. To avoid losing too much time, the LNS can stop the optimization phase with some conditions, such as the number of inconsistencies reached during the search.

### First solution

As for the local search, the LNS requires a first solution to perform a relaxation on it and start the search.

We are quite free when choosing the branching heuristic to find this solution, since it is not required to be the same for the optimization phase. We can therefore choose a heuristic that is the most suited for the situation.

# Chapter 3

# Overview of the sequence variable

The Job-Shop problem addressed in this master thesis contains sequences of tasks. The tasks are represented by the nodes in the sequence. A new data-structure has been implemented in [26] that will be used to model those sequences : the sequence variable. This new data-structure will help us to model the Job-Shop problem. All notations introduced in this chapter and used in the next ones are inspired from [5], [26] and [27].

## 3.1 The sequence variable data-structure

This variable represents a sequence of nodes. It embeds the current state of the sequence with the nodes being member of the sequence as well as the nodes not (yet) inserted. The sequence variable is used to model the state of the sequence that is built throughout the search. It provides a flexible representation of the state, supplies information about the current state and the actions that can be performed, such as where a node can be inserted within the sequence.
More formally, the sequence variable is defined as:

**Definition 3.1** (Sequence variable)**.** The set of all nodes associated with the sequence is denoted by $\mathcal{X}$. A sequence over $\mathcal{X}$ is denoted by $\overrightarrow{S}$ and the set of its elements $S$. All nodes in a sequence are different. A precedence between a node $p$ and a node $q$ is defined as $p \overset{\overrightarrow{S}}{\prec} q$ and $p \overset{\overrightarrow{S}}{\rightarrow} q$ signifies that $p$ directly precedes $q$. Each sequence requires a *begin* and *end* node. They are respectively represented by $\alpha$ and $\omega$. And finally, the sequence variable is denoted by $Sq$ and its domain ($D_{Sq}$) is represented by $\langle \overrightarrow{S}, I, P, E, R \rangle$ where :

- $\overrightarrow{S}$ is the sequence,
- $I$ the set of tuples containing all potential insertions in $\overrightarrow{S}$,

- $P, E, R$ the sets of all possible, excluded and required nodes. A node can only belong to one of these sets. Member nodes, nodes of the $S$ set, are also in the required set $R$.

**Definition 3.2** (Insertion)**.** The insertion of a non-member node $q$ after a non-excluded node $p$ of a sequence $\overrightarrow{S}$ is denoted by $(p, q)$.

**Definition 3.3** (Member insertion)**.** The insertion of a non-member node $q$ after a member node $p$ of a sequence $\overrightarrow{S}$ is denoted by $(p, q)$.

It is defined as adding the node $q$ in $\overrightarrow{S}$ such that the node $p$ directly precedes $q$. The successor of $p$ becomes $q$, the successor of $q$ is the previous successor of $p$ and the predecessor of the previous successor of $p$ is $q$. The node $q$ becomes required.

**Definition 3.4** (Indirect insertions)**.** As some nodes of the set may not have been inserted yet, we can define an indirect insertion of a node as an insertion that can be performed once another node is inserted. More formally, an indirect insertion for a node $i$ is an insertion such that there exist another non-inserted (but insertable) node $n$ such that node $i$ has a direct insertion in the sequence once node $n$ is inserted.

### 3.1.1 A state for each node

In order to model the sequence being build, each node has a state :

- *Excluded* – The node can not be inserted anymore in the given sequence.
- *Possible* – The node can still be inserted in the given sequence.
- *Required* – The node has to be inserted in the given sequence for the solution to be valid.
- *Member* – The node is part of the given sequence.

When a node is a member, it is always required (but the opposite is not true). If a possible node is inserted, it is required at the same time.

In the sequence depicted on figure 3.1, the different states of a node are represented. The nodes $a$ and $b$ are considered as *required*, $c$ is a *possible* node and $d$ is *excluded*. All states can be concurrently present in a sequence.

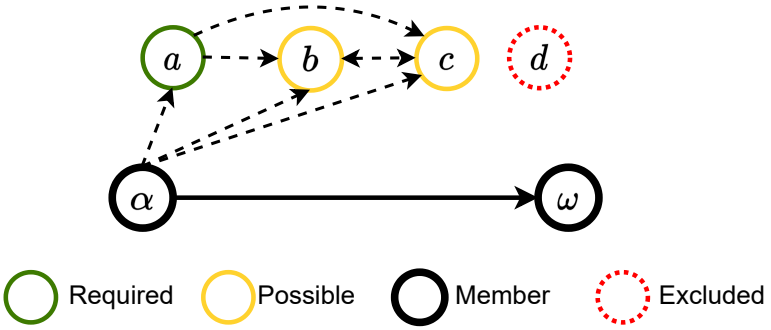



Figure 3.1: Example of a sequence where each node's state is represented, where the dotted arrows represent possible insertions and in bold the member precedences

### 3.1.2 The state of the sequence

As said above, the sequence variable data-structure maintains a set of useful information regarding the sequence being built. To enumerate some of them :

- The set of possible nodes $P$, the set of excluded nodes $E$, and of required nodes $R$.
- For each non-inserted node, the set of nodes after which it could be inserted. The set that contains the tuples of all possible insertions $(p, q)$ is denoted by $I$ (see eq. 3.1). For a node $x$, the set of its insertions is denoted by $I^x$ (see eq. 3.2).

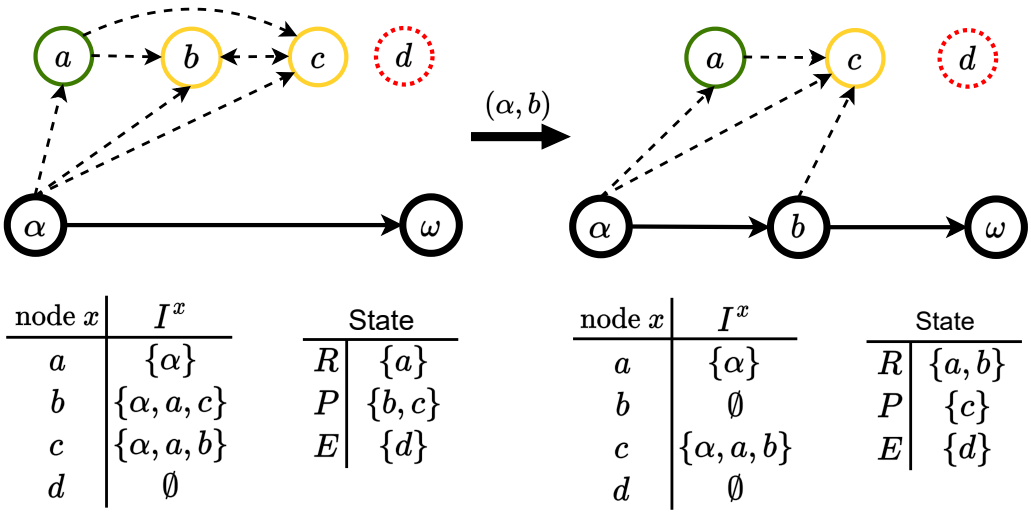
$$I = \{(p, q) \mid p \in (P \cup R),\ q \in P \cup (R \setminus S)\} \tag{3.1}$$

$$I^x = \{p \in (P \cup R) : (p, x) \in I\} \tag{3.2}$$

- For each non-inserted node, the set of nodes before which it could be inserted.

All this information is updated on insertion or exclusion of a node.

This schema 3.2 is provided to understand better the definitions given before. All insertions are now represented, even the indirect ones, and the table contains some information stored in the structure of a sequence.




| node $x$ | $I^x$ |
|---|---|
| $a$ | $\{\alpha\}$ |
| $b$ | $\{\alpha, a, c\}$ |
| $c$ | $\{\alpha, a, b\}$ |
| $d$ | $\emptyset$ |

| State | |
|---|---|
| $R$ | $\{a\}$ |
| $P$ | $\{b, c\}$ |
| $E$ | $\{d\}$ |

| node $x$ | $I^x$ |
|---|---|
| $a$ | $\{\alpha\}$ |
| $b$ | $\emptyset$ |
| $c$ | $\{\alpha, a, b\}$ |
| $d$ | $\emptyset$ |

| State | |
|---|---|
| $R$ | $\{a, b\}$ |
| $P$ | $\{c\}$ |
| $E$ | $\{d\}$ |

Figure 3.2: Figure represents a sequence and the different information stored while an insertion is made.

## 3.2 Implementation and operations on the sequence variable

After having discussed the definition of each element composing the sequence variable data structure, we will now provide an overview of its implementation and the operations that can be applied to it.

### 3.2.1 Sequence State Implementation

The sets $P, E$ and $R$ are implemented using the `StateTriPartition` data structure, which enables saving and restoring the state when the backtracking procedure is called.

The set $I$ is implemented as an array of `StateSparseSet` data structures, where each element in the array corresponds to $I^x$. The variable that stores the insertions according to the precedent is implemented with the `StateSparseSet` data structure. The same data structure is also used to implement the set of the successor insertions.

Additionally, there are two `final int` variables used to store the begin and end nodes. These variables are declared as `final` to prevent them from being changed once initialized.

The `StateInt` data structure is also utilized to implement the count of member nodes in the sequence variable. This data structures are described in [22].

### 3.2.2 Sequence variable operations

In the context of the Job-Shop problem, several operations of the sequence variable ($Sq$) interface are used. They are detailed below.

- `isMember`/`isRequired`/`isPossible`/`isExcluded`($Sq$, $x$). These four operations allow knowing, respectively, if the node $x$ is member, required, possible or excluded. The complexity is $\Theta(1)$.
- `getMember`/`getRequired`/`getPossible`/`getExcluded`($Sq$) functions return the set of member ($S$), required ($R$), possible ($P$) or excluded ($E$) nodes. This is done with a complexity depending, respectively, on the size of $S$, $R$, $P$ or $E$ .
- `pred`/`succ`($Sq$, $x$) returns, respectively, the predecessor or the successor of the node $x$ in the sequence variable $Sq$ in $\Theta(1)$.
- `insert`($Sq$, $p$, $x$) inserts the node $x$ after the node $p$ in the sequence variable $Sq$ in $\Theta(P + R)$.
- `exclude`($Sq$, $x$) excludes the node $x$ from $Sq$ in $\Theta(P + R)$.
- `nextMember`($Sq$, $x$) returns the successor of the member node $x \in S$. The time complexity is $\Theta(1)$.
- `fillMemberPredInsert`($Sq$, $x$) fills an array with all possible insertions $p \mid (p, x) \in I^x \ \ \& \ \ p \in S$.
- `removePredInsert`($Sq$, $x$, $p$) removes the insertion $p \rightarrow x$ by deleting $(p, x) \in I$ in $\Theta(P + R)$.

Although not explicitly mentioned in the description of each operation, it is important to note that whenever necessary, all information stored in the sequence variable structure is updated. This ensures that the sequence variable remains consistent and reflects any changes made during the execution of operations.

A notable observation is that the majority of operations are performed in constant time complexity, $\Theta(1)$, while some operations may have a complexity of $\Theta(|S|/|R|/|E|/|P|)$. This efficiency is one of the most intriguing aspects of the approach. However, it comes with the trade-off of increased memory usage due to the storage of information such as the $R$, $E$, $S$, and $I$ sets. In addition, using the `StateSparseSet` data structure requires saving the state to ensure proper restoration, which increases memory requirements.

## 3.3 Related work

The sequence variable has found application in various other problems, including the Vehicle Routing Problem (VRP), the Dial a Ride Problem (DRP), and the Patient Transportation Problem (PTP). The article cited as [5] presents the results specifically for the VRP, while the paper referenced as [27] presents the results for the DRP and PTP.

# Chapter 4

# Job-Shop model with sequence variable

In the preceding sections, we introduced the Job-Shop problem and discussed sequence variables independently. We will now integrate these concepts by presenting a model that uses sequence variables to address the Job-Shop problem.

## 4.1 Variables

### 4.1.1 Machines

The model we used (eq. 4.1 and 4.2) relies on a specific data-structure known as the *sequence variable Sq*. A sequence $Sq_m$ models each machine $m$. Scheduling a specific task is performed by inserting a node representing that task on it's machine at a specific place (i.e. after some already inserted node). The set of nodes $\mathcal{X}_m$ of a sequence $m$ represent all the tasks to be performed on machine $m$.

In some context, we will refer to a unique sequence. In this case, the indices $(m)$ will be omitted.

**Definition 4.1** (Model)**.**

$$Sq_m \coloneqq M_m \quad \forall m \in \{1, \ldots, n_m\}$$

$$\text{with} \quad D_{Sq_m} = \langle \overrightarrow{S}_m, I_m, P_m, E_m, R_m \rangle \tag{4.1}$$

$$\mathcal{X}_m \coloneqq T_{j,m} \; j \in \{1, \ldots, n_j\}, m \in \{1, \ldots, n_m\}$$

$$\text{where } \mathcal{X}_m \text{ is unique for each machine } m. \tag{4.2}$$

The most direct interest of this data-structure to model the machine's schedules is the fact that it defines a relative order on the tasks it contains, rather than fixing a value of the start-time of the tasks.

As we said before, each sequence represents a machine $m$ and must contain at the end a fixed number of tasks. All nodes represent a task, so they are required as a solution should contain all tasks in its schedule. With this predicate, the sets $E_m$ and $P_m$ of eq. 4.1 will be empty and $R_m$ includes all nodes (Formally, eq. 4.3).

$$E_m = \emptyset \quad \forall m \in \{1, \ldots, n_m\} \tag{4.3}$$
$$P_m = \emptyset \quad \forall m \in \{1, \ldots, n_m\} \tag{4.4}$$
$$R_m = \mathcal{X}_m \quad \forall m \in \{1, \ldots, n_m\} \tag{4.5}$$

### 4.1.2 Time windows

The objective of this problem depends on time. On the other side, the sequence variables defines a relative order on the member nodes. Each task is linked with a start variable (eq. 4.6). There are therefore a total of $n_m \times n_j$ start variables. The domain of each variable is the range of value between its earliest starting time and its latest starting time (see section 1.3.1 for the definitions of *est* and *lst*).

**Definition 4.2** (Start variables)**.**

$$v \coloneqq start_i \quad \forall i \in \{1, \ldots, n_j * n_m\} \tag{4.6}$$

where $D_v = \{est_v, \ldots, lst_v\}$,
where est = earliest starting time, and lst = latest starting time.

**Definition 4.3** (End variables)**.**

$$end_i \equiv start_i + duration_i \quad \forall i \in \{1 \ldots n_j * n_m\} \tag{4.7}$$

where :

- $D_{end_i} = \{est_i + duration_i, \ldots, lst_i + duration_i\}$
- $start_i$ is defined in eq. 4.6
- $duration_i$ is a fixed and unique value for each task.

### 4.1.3 Shared *begin* and *end* nodes

Sequence variables are build with a fixed *begin* node $\alpha$ and *end* node $\omega$. Those nodes mark the first and last nodes of the sequence. They are specified at sequence definition and never change. We could arbitrary choose which task should be scheduled at first and at last, but this would already fix part of the search.

Another solution is to add 2 *fake* tasks (with 0 duration) that do not represent any real task. It was chosen to use those 2 nodes shared among all the sequences as *begin* and *end* node. We are allowed to share the *begin* and *end* nodes because they don't impact the solution. While they serve no purpose for the scheduling of the tasks, the end node can however be used to compute the makespan. Indeed, as it must take place after all the jobs, the *end* node starts after the last node/task across all sequences the value of the makespan.

## 4.2 Constraints

### 4.2.1 Job and machine precedence

As said earlier, *lessOrEqual* constraints are posted between the consequent activities of a same Job to ensure their precedences. As the order of the tasks of a job is known from the start, those constraints are posted once for all at the problem definition.

On the other side, the order of the tasks on a given machine being part of the solution to be founded, the non-overlapping aspect of the execution-windows of the tasks scheduled on a same machine must be handled during the search. This is the purpose of the *Channeling* constraint introduced in section 4.2.2.

### 4.2.2 Channeling constraint

The *Channeling* is a constraint that is used to ensure nodes with associated duration to maintain consistent temporal order with respect to their order in a sequence $\overrightarrow{S}$ for a sequence variable $Sq$. Each node $i$ in the sequence is associated to a variable $start_i$, $end_i$ and a duration $duration_i$

This constraint is based on the constraint *TransitionTimes* from [5] which can be used for routing problems with duration and transitions times between nodes. Since we only have durations, we can simplify the definition of the *TransitionTimes* constraint to obtain the one for the *Channeling* constraint.

**Definition 4.4** (Channeling)**.** For a sequence variable $Sq$, the *Channeling* constraints ensure that, for each node $i$ preceding $j$ in the sequence $\overrightarrow{S}$, the variable $end_i$ is less or equal to the variable $start_j$.

$$\texttt{Channeling}(Sq, [start], [end]) \equiv \left\{ \overrightarrow{S} \in D_{Sq} \mid \forall i, j \in \overrightarrow{S}, i \overset{\overrightarrow{S}}{\prec} j \Longrightarrow end_i \leq start_j \right\} \tag{4.8}$$

### Minimal Filtering

The minimal filtering for this constraint to be correct is for the start and end variables to have valid values. To do this, each time a node is inserted in the sequence, a *lessOrEqual* constraint is posted regarding its current predecessor and current successor in the sequence.

In the case of multiple insertions in the sequence at the same time, we can actually make an improvement if the newly inserted nodes are consecutive. Looking at figure 4.1, we can see that if we post two *lessOrEqual* for the successor and predecessor nodes, we will have a duplication of constraints for node $d$ and $e$. To solve this, we post the *lessOrEqual* for the successor only if it was already in the sequence. Doing so, we process node $d$ first, and we only post for the predecessor $c$. After, when we process $e$, we will post the two constraints for $d$ preceding $e$, and $e$ preceding $\omega$.

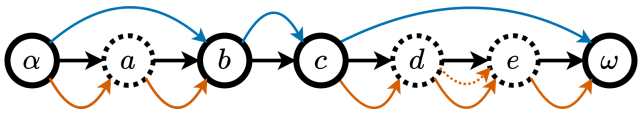



Figure 4.1: Example of the minimal filtering of the *Channeling* constraint. The dotted nodes are the newly inserted nodes. The curved arrows represent the *LessOrEqual* constraints between two nodes. The blue curved arrows on top are the constraints present before the insertions, and the orange below are the new ones. The dotted curved arrow depict the avoided duplication of constraints.

**Complexity** Since the minimal filtering delegates the work of filtering the start variables to *lessOrEqual* constraints, evaluating with precision the complexity of this minimal filtering depends on other factors like the other constraints of the model. We will therefore evaluate the complexity by analyzing the number of *lessOrEqual* constraints posted by the *Channeling* constraint. In the worst case, each node is the sequence will post 2 *lessOrEqual* constraints. This gives us a total of $(|S| * 2) - 3$ constraints. If they are propagated once at each call of the fixpoint procedure, this gives us a complexity of $\mathcal{O}(|S|)$.
It is important to note that the *TransitionTimes* constraint performs the same filtering by traversing the sequence $\overrightarrow{S}$ from the node $\alpha$ to $\omega$ and back to prune the start variables. We can therefore also evaluate the complexity of this filtering to also be $\mathcal{O}(|S|)$.
The advantage of the filtering with *lessOrEqual* constraints is that we gain in locality because if the impact of an insertion is local, we only need to propagate the constraints that are impacted by the variation instead of running through the whole sequence as for the *TransitionTimes* constraint. On the other hand, posting

numerous constraints puts more work on the fixpoint procedure and backtracking procedure, which needs to keep tracks and remove *lessOrEqual* constraints when necessary.

### Filtering insertions

The *filteringInsertions* algorithm is an optional filtering that can be enabled to remove impossible insertions in the sequence. It is propagated each time a node is inserted or when the domains of the start and end variables are modified.

Algorithm 2 depicts the workings of the *filteringInsertions* process, which works as follows. For each non-member node *current*, the insertions whose predecessor belongs to the sequence are listed. Fist, at line 7, we check that *current* can start before the member predecessor *pred*. Second, at line 11, we check that the member successor of $pred : succ$ can still start before the end of *current*, if *current* was inserted between the member nodes. If the two previously mentioned checks are not satisfied, the insertion is thus removed.

In the case of the Job-Shop problem, we can perform more pruning. Indeed, in the way the problem is modelled we will never encounter nodes with only indirect insertion, i.e., insertions that are not linked to a member node but between a non-member node. We can therefore exclude the nodes if they can not be directly inserted in the sequence. This is done at line 18. Furthermore, since all nodes of the problem are required regarding their sequence, meaning that they need to be inserted for the solutions to be valid. Those exclusions will actually allow the *Channeling* constraint to detect inconsistencies during the search.

**Algorithm 2** filteringInsertions algorithm

**Require:** start, end, duration, $Sq$
**Ensure:** Pruning impossible insertions

```
1: requiredNotMemberNodes ← required not member nodes of Sq
2: for current ∈ requiredNotMemberNodes do
3:    insertionsPoint ← all predecessor member nodes of the current node
4:    foundDirectInsert ← false
5:    for pred ∈ insertionsPoint do
6:       succ ← the member node succeeding the pred node
7:       if min(end_pred) > max(start_current) then
8:          removePredInsert(Sq, current, pred)
9:       else
10:         earliestDeparture ← max(min(end_pred), min(start_current))
11:         if earliestDeparture + duration_current > max(start_succ) then
12:            removePredInsert(Sq, current, pred)
13:         else
14:            foundDirectInsert ← true
15:         end if
16:      end if
17:   end for
18:   if not foundDirectInsert then
19:      exclude current node from seq
20:   end if
21: end for
```

**Complexity** The complexity of this filtering algorithm is, in the worst case, $\mathcal{O}(|R \backslash S| \times (|S| - 1) \times (|P| + |R|))$. $|R \backslash S|$ comes from the for loop on required non-member nodes. $|S| - 1$ comes from the for loop on all predecessor member nodes (the number of member insertions is bounded by the number of member nodes). And $|P| + |R|$ comes from the insertion deletion performed by `removePredInsert` 3.2. In the best-case scenario, the complexity is $\mathcal{O}(|R \backslash S| \times (|S| - 1))$.

### Enforce last insertion

One more filtering can be performed for the *Channeling*. Since the required node must be in the sequence, we can insert those nodes if they only have one last direct insertion and if there are no other nodes that could be inserted after the same predecessor. Looking at figure 4.2, we can see that we can insert node $c$, since its insertion is the only insertion that must happen for the solution to be valid.

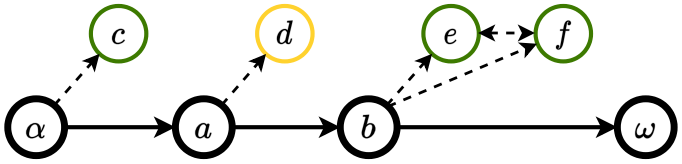



Figure 4.2: Example of sequence for the filtering to enforce the last insertion of the required nodes.

**Complexity** The complexity of this filtering algorithm is $\mathcal{O}(|R \backslash S| \times (|P| + |R|)$. This is induced by the algorithm that performs a $\mathcal{O}(1)$ check (number of member insertions) together with a possible insertion with $\mathcal{O}(|P| + |R|)$ complexity for all the $|R \backslash S|$ required not-member nodes.

### Overall complexity

In practice, the filtering algorithm for enforceLastInsertion is embodied in the filteringInsertions algorithm. Indeed, since we check for all required not member nodes to be inserted in algorithm 2, the check to enforce the last insertion can be done after line 3 if the number of member insertion is 1. The complexity of the filteringInsertion will therefore bound the overall complexity of the Channeling constraint.

# Chapter 5

# Branching procedures for the Job-Shop modelled with sequence variables

In the previous chapters, we introduced how the job-shop was modelled using the sequence variable data-structure (chapter 1) and how the search can be performed in CP (chapter 2). In the present chapter, we will expand on methods that exploit the knowledge of the problem or common branching heuristics to provide an efficient search tree.

As our branching schemes are similar in many aspects, we defined a *template branching*, presented in section 5.1 that is used by all the branching schemes presented in this chapter (at least partly, as in section 5.2.4). A few more branching schemes will be introduced in chapter 6, but their structure and purpose are different.

As will be explained in section 5.1, the branching heuristics we designed can be separated in several components, the design heuristics for each component are detailed in a dedicated section each.

Section 5.2 will introduce heuristics to select a good node to perform its insertions. Follows section 5.3 that details heuristics about which insertion should be preferred, and performed prior to others for a given node. Finally, section 5.4 will elaborate on heuristics to choose a good sequence or set of sequences for the branching process.

## 5.1 Template branching

### 5.1.1 Variable and value selection

A branching procedure can be described by two dimensions : the *variable selection* and the *value selection*. Considering the job-shop problem modelled with sequence variables and the branching procedures of this section, the *variable selection* corresponds to the selection of the node being inserted, whereas the place in the sequence where the node is inserted refers to the *value selection*. Recall from chapter 4 that the sequence variables model the machines of the job-shop problem whenever referenced in the next chapters.

### 5.1.2 Branching procedure interface

Many branching schemes (i.e. branching procedures) we defined share a similar pattern of selecting a given node using a selection criterion before trying all the insertions for that node. We therefore defined a template algorithm where three components can be implemented as desired by a branching procedure implementation, while the remaining common aspects are handled by the template algorithm.

Those three components defined by a specific branching are,

- A *node selection criterion* – used to select the node to be inserted among the required nodes. It is modelled as a method that assigns to each node a value. The node with maximal value being the most valued one.
- A *sequence selection criterion* – that defines the subset of non-fixed sequences (i.e. with at least one insertable node) that will be used to search for the best node using the node selection criterion. If *none* is specified, all sequences of the problem are used. It is modelled as a predicate for a sequence that states if the sequence should be used or not.
- A *branching order* – which is at its root a sorting method on the different insertions that form the branches of each choice point. In practice, the method is to be applied on the list of member predecessors of the node to be inserted, but this is equivalent.

### 5.1.3 The template algorithm

After having introduced the interface implemented by the branching procedures, let's see the algorithm of the template itself. It is depicted in algorithm 3 and works as follows.

**Getting the branches** At first, we check if there are some unfixed sequences (line 1), returning the *EMPTY* symbol if this is not true. The *EMPTY* symbol is returned to notify that the branching does not supply branching points anymore.

Next, we call the *getBestNode* (line 5) method that is responsible for the selection of the best node to be inserted regarding a specific heuristic. If no node is found, the state is inconsistent, as we should have found a node to insert given some sequence(s) are non-fixed.

Afterward, we sort the insertions given the *branchingOrder* heuristic that is given (line 10).

Finally, we create a branch for each insertion of the node and return that array of branches.

**Algorithm 3** Template branching scheme

**Require:** *SqList*, *sequenceSelector*, *getNodeCriterion* and *branchingOrder*
**Ensure:** Returns a list of procedures
1: **if** isAllFixed(SqList) **then**
2: return EMPTY
3: **end if**
4: $nodeCriterion \leftarrow getNodeCriterion()$
5: $(bestNode, bestSq) \leftarrow$ getBestNode(nodeCriterion);
6: **if** $bestNode = -1$ **then**
7: **throw** Inconsistency
8: **end if**
9: $predecessorList \leftarrow$ Possible member predecessor nodes of the $bestNode$
10: $predecessorList \leftarrow branchingOrder.sort(bestNode, bestSq, predecessorList)$
11: $branches \leftarrow empty\ list$
12: **for** $pred \in predecessorList$ **do**
13: $branches.add($**lambda** $() \rightarrow$ insert $bestNode$ after $pred$ in $bestSq)$
14: **end for**
15: **return** branches

**Get the best node** Algorithm 4 depicts the getBestNode method. In a nutshell, the algorithm iterates over the non-fixed selected sequences to find among the required-non-member nodes the best node to be inserted.

In more detail, the algorithm goes as follows. It iterates over the sequence variables (line 4) and the non-member required nodes (line 8) selecting the node which has maximal value while having at least one insertion (line 9).

It can be mentioned that fixed or non-selected sequences are excluded from the maximization process (line 5) and that the process stops in case a node is guaranteed to be the best one (line 13).

**Algorithm 4** GetBestNode

**Require:** $SqList$, $sequenceSelector$ and $nodeCriterion$
**Ensure:** Return the best node regarding $nodeCriterion$ and its related sequence variable, or -1 if none is founded.

```
1: bestNode ← −1
2: bestValue ← −∞
3: bestSq ← −1
4: for Sq ∈ SqList do
5:    if isFixed(Sq) or not sequenceSelector(Sq) then
6:       continue
7:    end if
8:    for node ∈ requiredNotMemberNodes(Sq) do
9:       if nodeCriterion(node) > bestValue and hasInsertion(node, Sq) then
10:         bestNode ← node
11:         bestValue ← nodeCriterion(node)
12:         bestSequence ← Sq
13:         if bestValue = ∞ then
14:            return (bestNode, bestSq)
15:         end if
16:      end if
17:   end for
18: end for
19: return (bestNode, bestSq)
```

## 5.2 Branching heuristics

In this section, we will introduce different variable selection heuristics. Said differently, we detail heuristics to select the best node to be inserted.

### 5.2.1 First-fail branching

The first branching procedure presented is denoted as the first-fail branching in our experiments, as it follows a first-fail strategy. It mirrors the example of first-fail branching presented in 2.3.2, performing a *smallest domain first* variable selection as it selects the node that has the least number of insertions in its sequence.

Regarding the template branching defined in 5.1, we define the getter method of the node selection criterion as showed on algorithm 5.
The function `nMemberPredInsert` returns the number of member nodes of the sequence that are a possible predecessor of the node given as argument.

**Algorithm 5** getNodeCriterion for first-fail branching procedure

**return lambda** $(Sq, node) \rightarrow -Sq.nMemberPredInsert(node)$

### 5.2.2 Maximum duration branching

An alternative branching strategy that has been implemented focuses on the duration of each task. In this approach, the chosen node is the one whose corresponding task has the maximum duration. The rationale behind this particular branching method is the expectation that it would be relatively simpler to insert a node representing a shorter-duration task when the sequence is nearly complete, as compared to inserting a node representing a longer-duration task.

The branching heuristic directly translates to the node selection criterion, whose getter method is depicted on algorithm 6.

**Algorithm 6** getNodeCriterion for maximum duration branching procedure

**return lambda** $(Sq, node) \rightarrow duration[Sq][node]$

### 5.2.3 Min-max-slack branching

The branching detailed in this section relies on the notion of slack, which is introduced first. Afterward, the branching procedure itself is presented as an implementation of the node criterion method relying on a helping function. Finally, an example illustrates the behavior of the presented algorithm. See [25] for another example of slack definition and a branching procedure based on it.

#### Slack

The *slack* metric as used here is related to the degree of freedom of a node after its insertion. The degree of freedom translates to the comparison between the execution window (or a variation of it) of the inserted node after insertion and the duration associated with that node. The slack metric is specific to each insertion, as the execution window after insertion depends on the predecessor and successor nodes in the sequence between which the node is inserted.

We defined two different slack-metrics, that are detailed below. In the remainder of this section *node* will denote the node being inserted and *pred* and *succ* the predecessor and successor (successive) nodes between which *node* is inserted.

**Full-slack** The full-slack metric is the metric matching the most the intuition given so far. It is equal to the size of the execution window after having inserted the node, from which the duration of the node itself is subtracted.

**Definition 5.1** (Full-slack)**.** The full-slack metric for a sequence $\overrightarrow{S}$, two consecutive member nodes *pred* and *succ* of $S$ and a node *node* that is directly insertable between *pred* and *succ* is defined as the size of the execution window of *node* after insertion minus the duration of *node*.

Formally, on eq. 5.1.

For a sequence $\overrightarrow{S}$,

$$\forall\ pred, succ \in S, \forall\ node \in R \backslash S \mid (pred \overset{\vec{S}}{\rightarrow} succ\ \ \&\ \ pred \in I^{node}) :$$
$$fullSlack(node,\ pred,\ succ,\ \overrightarrow{S}) \equiv newLct_{node} - newEst_{node} - duration_{node},$$
$$\text{where } newEst_{node} = max(ect_{pred},\ est_{node}),$$
$$newLct_{node} = min(lct_{node},\ lst_{succ}) \quad (5.1)$$

An illustrative example of the full-slack computation is given on figure 5.1.

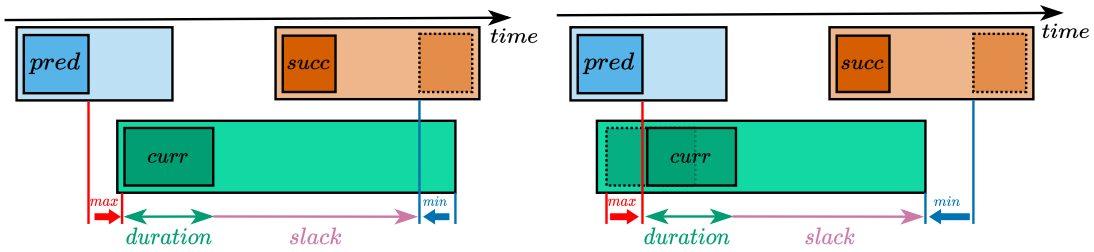



Figure 5.1: Full-slack metric illustration

Please note that, considering only valid insertions, the full-slack will always be greater or equal to 0. Indeed, if the insertions are valid, there is room for the nodes' duration, but potentially not more.

**Left-slack** The full-slack defined above relies namely on that fact that an inserted node *node* can delay the successor *succ* before which it is inserted. In order to favor insertions that would not delay the successor node on insertion (potentially augmenting the makespan of the problem), we designed a variant of full-slack named left-slack.

The left-slack computation differs from full-slack for the *newLct* value computation. *newLct* will be constrained by $lct_{node}$ and $est_{succ}$ instead of $lct_{node}$ and $lst_{succ}$ for full-slack. This mirrors exactly the aspect that *succ* can not be delayed.

We can give a definition for the left-slack computation. It is identical to the full-slack computation, except regarding the *newLct* value.

**Definition 5.2** (Left-slack)**.** The left-slack metric for a sequence $\overrightarrow{S}$, two consecutive member nodes *pred* and *succ* of $S$ and a node *node* that is insertable between *pred*

and *succ* is defined as the size of the execution window of *node* after insertion minus the duration of *node*, not considering *succ* can be delayed. In other words, $start_{succ}$ is considered fixed to $est_{succ}$ for the computation.

Formally, on eq. 5.2.

For a sequence $\overrightarrow{S}$,

$$\forall\ pred, succ \in S, \forall\ node \in R \backslash S \mid (pred \xrightarrow{\overrightarrow{S}} succ\ \ \&\ \ pred \in I^{node}) :$$
$$leftSlack(node,\ pred,\ succ,\ \overrightarrow{S}) \equiv newLct_{node} - newEst_{node} - duration_{node},$$
$$\text{where } newEst_{node} = max(ect_{pred},\ est_{node}),$$
$$newLct_{node} = min(lct_{node},\ est_{succ}) \quad (5.2)$$

An illustration of the left-slack metric is given on figure 5.2.

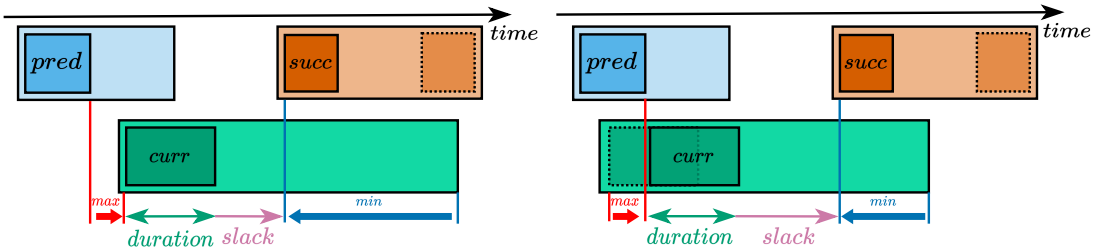



Figure 5.2: Left-slack metric illustration

It can be noticed that unlike the full-slack metric, the left-slack metric can be negative. This occurs when fixing $start_{succ}$ to $est_{succ}$ would not allow *node* to be inserted because the available space would be too small. This situation is depicted in figure 5.3. The left-slack metric strives for no delaying *succ* but does not prevent it.

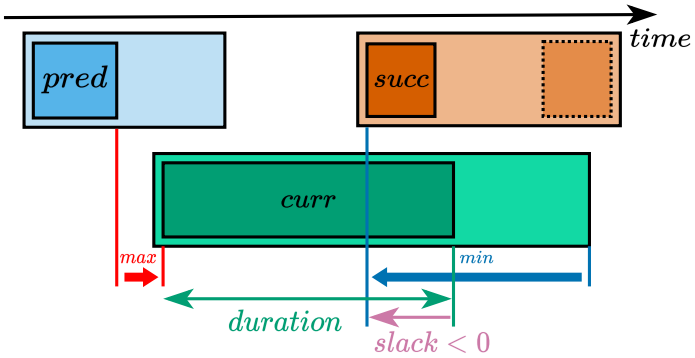



Figure 5.3: Left-slack metric illustration – negative slack use case

**Slack upper-bound** As we said, the slack value depends (independently of the slack type) on the predecessor and successor nodes as they constrain the execution-window after insertion. In order to be more efficient, the algorithm presented in the next section relies on an upper-bound of the slack value that escapes this dependency.

The upper-bound is based on a property of the slack computation : as the execution window after insertion of *node* is constrained by $ect_{pred}$ and $lst_{succ}/est_{succ}$ for full/left slack, the final slack of *node* is upper-bounded by the bounds displayed in eq. 5.3 and 5.4.

$$UB_{fullSlack} = lst_{succ} - ect_{pred} \tag{5.3}$$

$$UB_{leftSlack} = est_{succ} - ect_{pred} \tag{5.4}$$

A visual representation of the upper bound is depicted in figure 5.4.

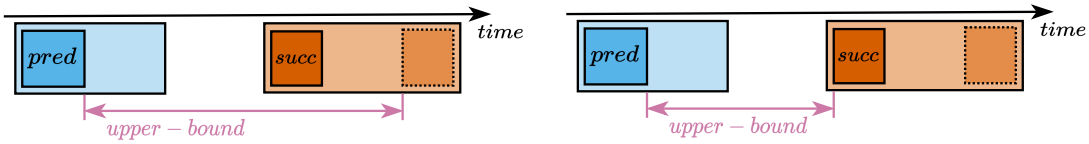



Figure 5.4: Slack upper-bound illustration – Full-slack metric on the left side and left-slack metric on the right side.

### Min-max-slack branching

**Heuristic** The core idea of the min-max-slack branching is to insert primarily the node that has the smallest slack among its member insertions. In more detail, we compute for each node the biggest slack among all its member insertions, named **maximal-insertion-slack** value. The node with the smallest or minimal *maximal-insertion-slack* is chosen for insertion. By inserting the node with smallest maximal-slack we focus on more critical nodes, nodes that are more likely to generate inconsistencies.

**Node criterion** Relying on a method to compute the maximal-insertion-slack of a node, we can define the *getNodeCriterion* method for the min-max-slack branching as in algorithm 7.

**Algorithm 7** getNodeCriterion for min-max-slack branching procedure

**Require:** $Sq, slackCalculator$
**Ensure:** returns the node Criterion for the maximal-insertion-slack branching procedure
  **for** $Sq \in SqList$ **do**
    **if** $sequenceSelector(Sq)$ **then**
      $computeMaxSlack(Sq, slackCalculator)$
    **end if**
  **end for**
  **return lambda** $(Sq, node) \rightarrow -maxSlackForNodeList[Sq][node]$

**computeMaxSlack** One way to compute the maximal-insertion-slack of a node would be to iterate over all its member insertions and compute the slack value for each of them. Fortunately, there is room for optimization here. Using an upper-bound pre-computation, we can try to optimize the computation process.

We detail below the steps of the algorithm 8. It considers only one sequence, but the algorithm scales for multiple sequences with no concerns.

- For each member node in the sequence (except for $\omega$) we compute the upper-bound of the slack value valid for any member insertion performed between this node and the next node in the sequence (line 2).
- When considering the member insertions of each node, we start by sorting the insertions (defined by the member predecessor) by decreasing value of their slack upper-bound value (value that was already computed for each member node) (line 14).
- The real slack values for all the member insertions of a node are computed iteratively following the sorted order of decreasing slack upper-bound value. Here is the trick : When iterating over the member insertions, if the exact maximal slack value found so far is greater than the upper-bound of the slack value of the next member insertions (minus the duration of the node) we can stop the process as we are guaranteed to have the maximal value (line 20). All other member insertions have lower or equal slack upper-bound.

**Algorithm 8** computeMaxSlack

**Require:** $Sq$, $slackCalculator$

**Ensure:** computes the maximal-insertion-slack of all the insertable nodes of a sequence.

```
1: // Compute the upper-bound value for all member nodes
2: for member ∈ memberNodes(Sq) do
3:    if member = Sq.end() then
4:       slackUpperBound[member] ← −∞
5:    else
6:       slackUpperBound[member] ← slackCalculator.UB(Sq, member,
         Sq.nextMember(member))
7:    end if
8: end for
9: maxSlackForNode ← −∞
10: insertableNodes ← requiredNotMember(Sq)
11: // Iterate over all insertable nodes
12: for node ∈ insertableNodes do
13:    memberPredInsertions ← memberPredInsert(Sq, node)
14:    memberPredInsertions.sort(lambda (i, j) → slackupperBound[j] −
       slackupperBound[i])
15:    // Iterate over member predecessor
16:    // following decreasing slack-upper-bound order
17:    for nodePred ∈ memberPredInsertions do
18:       // Stop if current maximal slack is bigger then
19:       // the upper-bound of the remaining insertions.
20:       if maxSlackForNode ≥ slackUpperBound[nodePred] − duration_node
          then
21:          break
22:       end if
23:       predSlack = slackCalculator.slack(Sq, nodePred, node,
          Sq.newMember(node))
24:       maxSlackForNode ← max(maxSlackForNode, predSlack)
25:    end for
26:    maxSlackForNodeList[Sq][node] ← maxSlackForNode
27: end for
```

**Illustration** Let's illustrate the behavior of the algorithm on an example. Considering a sequence depicted on figure 5.5. The first step is to compute the upper-bounds for all member nodes. Let the upper-bound values be as in table 5.1.

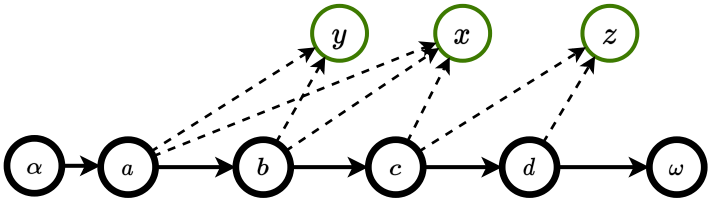



Figure 5.5: Example of a sequence for the min-max-slack algorithm – Only member insertions are displayed.

| 0 | 3 | 6 | 1 | 5 |
|---|---|---|---|---|
| $\alpha$ | a | b | c | d |

Table 5.1: Min-max-slack example : upper-bounds for member nodes excluding $\omega$

The next step is summarized in table 5.2. For each node, we sort the member insertions regarding their upper-bound slack value (first and second lines).

The slack computation is depicted on line 3. For the sake of simplicity, we consider that the duration of the nodes are 0 in this example. Consequently, the node's slack values are compared to $UB_{node}$ rather than $UB_{node} - duration_{node}$, which would be a stronger (smaller) upper-bound.

Considering node $x$, we start with insertion $b$ and compute a slack value of 2. As $2 < 3$, we need to compute the slack value of $a$, let it be 2. As the current maximal slack value of $x$ is 2 which is greater than the upper-bound of the next insertion $UB(c) = 1$, we are guaranteed that that slack value of insertion $c$ would be lower than 2 and do not need to compute it.

The computation process is similar for node $y$ and $z$. For $y$, as the slack value of insertion $b$ (4) bounds the upper-bound of insertion $a$ (3) we don't need to compute it. For $z$, as the slack value of insertion $d$ (3) bounds the upper-bound of insertion $c$ (1) we don't need to compute it.

| Nodes | x | y | z |
|---|---|---|---|
| Member insertions | b, a, c | b, a | d, c |
| Upper-bound | 6, 3, **1** | 6, **3** | 5, **1** |
| Slack | 2, **2**, * | **4**, * | **3**, * |

Table 5.2: Min-max-slack example : optimized computation of the maximal-insertion-slack value based on upper-bound computation. Non-computed values are denoted by a star (*). Values that allowed to stop the computation for a node are in bold.

### 5.2.4 Regret MVI branching

The *Regret MVI* is a branching heuristic seeking to minimize the makespan by making branching choices that are based on the impact that insertions have on the makespan. To represent this impact, we introduce a new value we called MVI.

**Definition 5.3.** The **MVI** or Makespan Variation on Insertion is the value reporting the increase of makespan caused by an insertion in a sequence of our Job-Shop model with sequence variables. The algorithm 9 shows how the computation is performed. If it happens that the insertion of a node in the sequences lead to other insertions in the sequences, the returned MVI is the variation divided by the number of inserted nodes (line 14).

The *Regret MVI* branching is, more specifically, a minimax regret branching [2, 15]. Those types of branching are looking to minimize a maximum criterion called **regret**. For each branching variable, the regret is computed as the difference between the two best values of an **evaluation function** that evaluate the choices for the variable. The regret thus represents the increase of the evaluation function that the branching could "regret" if the second choice was made instead of the first one. The minimization is performed by selecting the variable with the maximum regret and branching first for the choice with the minimal evaluation value.

For our model, the evaluation function is the computation of the MVI and the regret of a node is the difference between the two lowest MVI of the node. There are however some special cases we have to address:

- If the node has only one insertion, instead of returning a null regret, we return an infinite value. The motivation behind this is that since the node only had one insertion, performing it early will allow future nodes to better evaluate their impact on the makespan and thus have better regrets for the next branching points.
- If the computation of a MVI leads to an inconsistency in the search, then the insertions are not taken into account.

Once the node is chosen, a branch is made for each insertion in its sequence and sorted by increasing order of MVI.

#### Weaknesses

The main disadvantage with the *Regret MVI* heuristic is that the function to *computeMVI* is highly computational. Indeed, looking at algorithm 9, we can see that we need to save the state (line 3) of the solver because we need to be able to backtrack (line 12) to undo all the changes caused by the insertion. We

also need to call the fix point (line 6) to compute the MVI. In section 5.4 we will introduce a technique to reduce the number of nodes processed by the node selection criterion. As there will be less nodes for which the *computeMVI* has to be called, the branching procedure should therefore be faster.

**Algorithm 9** algorithm computeMVI

**Require:** SqId, pred, node, SqList, makespan, solver
**Ensure:** returns the MVI of the insertion pred→node, in sequence SqId
1: mvi ← min(makespan)
2: nInserted ← sum of the amount of member nodes in all sequences
3: solver.saveState()
4: **try**
5: SqList[SqId].insert(pred,node)
6: solver.fixPoint()
7: mvi ← min(makespan) - mvi
8: nInserted ← sum of the amount of member nodes in all SqList - nInserted
9: **catch** { Inconsistency }
10: **return** null
11: **finally**
12: solver.restoreState()
13: **end try**
14: **return** mvi/nInserted

## 5.3 Ordering the branches

Every branching heuristic introduced above in section 5.2 (except 5.2.4) works by finding the best node following their criterion and branching on all possible insertion of that node. An improvement can be made by sorting the branches created with a sorting procedure to direct the search towards parts of the search tree that may minimize the makespan. The sorting is done thanks to sorting heuristics that will evaluate the quality of insertions. The best insertions will be branched on first.

Three sorting heuristics have been implemented, the first one depends on the slack, the second on the position in the sequence, and the last one on the MVI.

They define a *branchingOrder* with associated *sort* method that is used in the template branching of section 5.1.

**SortMaxSlack** The SortMaxSlack heuristic relies on the principle of maximizing the remaining *flexibility* of the sequence after insertion. The flexibility of the sequence is interpreted as the amount of slack space that remains after the insertion

of the node. In short, the heuristic orders the insertions by decreasing value of slack. Recall that the slack of an insertion is defined by the slack of the inserted node after it has been inserted in the sequence.

Note that the slack definitions are the one introduced in section 5.2.3. We defined two distinct definitions of the slack (left-slack and full-slack). When performing a branchingOrder based on the SortMaxSlack heuristic while using the min-max-slack branching, the same slack-metric is used for both in our experiments.

**SortEarliest** As the name suggests, insertions are sorted by increasing order of the place at which the node is inserted in the sequence. The earliest possible insertion is performed first.

**SortLowMVI** This sorting heuristic sort the insertions by computing the impact they cause on the makespan. More exactly, the insertions are sorted by decreasing value of $MVI$ as introduced at definition 5.3.

## 5.4 Sequence selection

Another approach we have explored to enhance the branching was to select a subset of sequences upon which the node selection criterion is applied. The idea behind this heuristic is to trade some accuracy of the node selection, as we only use a subset of the nodes for the maximization process, for performance, as handling less nodes should make the branching procedure run faster.

In general, two ideas have emerged: one based on the number of member nodes in the sequence, and another based on the insertions between non-member nodes and member nodes.

**Most filled** The first sequence selector is named `mostFilled` which, as its name suggests, selects the fullest sequence. We hypothesized that this sequence selector should be inefficient (compared to the less filled heuristic, for instance). We implemented it in order to verify our assumption.

**Less filled** The second one is the `lessFilled`, unlike the first, selects the sequence with the fewest member nodes. The idea behind this heuristic is to gradually fill all the sequences

**Less insertions on size** This selector `lessInsertionsOnSize` prioritizes sequence with small number of possible insertions in the sequence, compared to

their size. The idea is to insert primarily in the more constrained sequences. The criterion for each sequence is :

$$\sum_{x \in R \setminus S} \frac{|I^x \cap S|}{\texttt{Number of member nodes}}$$

# Chapter 6

# Large Neighborhood Search

After the introduction of the model of the Job-Shop with sequence variables and the branching procedures, we can now look at the different technique to perform optimization search on our model.

Using the branching procedures of section 5.2, we can use a depth first search (DFS) with a branch and bound (BnB) strategy to search for optimal solutions.

Although BnB is used to reduce the size of the explored search tree, the performance of this reduction depends on the chosen branching heuristics and can still lead to a large tree with a time-consuming exploration. An issue here is that early branches taken in the tree could lead the search to do intensification on part of the tree that contains less optimal solutions, since branches needs to be fully explored before being backtracked on. A way to solve this issue is to use local search techniques to introduce diversification in the search. This will be done with the help of the Large Neighborhood Search (LNS).

In this chapter we will present in section 6.1 the branching heuristics developed to find a good first solution Then, in section 6.2 we will introduce multiples relaxations heuristics used for our model.

## 6.1 Initial search

Branching procedures used to preform optimization may not be suited to find a first good solution. This could be caused by the heuristic being too greedy and making poor branching choices because the minimal makespan was not well estimated enough.

To gain the time that the branching heuristic takes to reach a good estimation of the makespan, we start the LNS with an initial search using a DFS with a heuristic designed to find a "good enough" **first solution**. This first solution will give us an upper-bound of the makespan in order to prune, from the beginning,

non-valuable branches of the search tree.

We implemented tree similar branching procedures oriented to have a good first solution. The main inspiration for those procedures is the ***SetTimes*** algorithm presented in [10].

### 6.1.1 Adapting *SetTimes*

The *SetTimes* algorithm is a branching heuristic used for optimization problems, including scheduling problems. In the context of the Job-Shop problem, it would use the *start* variables as decision variables and branch to fix the variable with the lowest earliest starting time (*est*) to this value. In other words, the solutions are built by gradually fixing the task that can be processed first to its earliest possible position in the schedule.

Using the *SetTimes* algorithm in combination with our model and the sequence variable is not that convenient. Indeed, since the branching is performed on the *start* variables, we lose the flexibility of the insertions given by the sequence variables.

Our implementations of the first solution branching thus use insertions as the base choice for the branches produced, and consequently does not proceed by fixing the *start* variables. Like the *SetTimes* algorithm, the node that is chosen to be branched on is still the node associated to the task with minimal *est*.

It is how the insertion is treated that will vary between our different branching algorithm.

**Latest Insertion** Some of our implementations use the method of inserting the task with the lowest *est* at its latest insertion in its sequence. The sequence would act like a queue, since the tasks get progressively added at the end of the sequence. The intent of inserting at the latest insertion point is to prevent delaying already inserted tasks, since inserting at the earliest point would just be postponed, thus breaking the idea of the *SetTimes* algorithm.

### 6.1.2 Branching procedure

We will now introduce the developed branching procedures to find the "good enough" first solution for the LNS.

#### Earliest Start

In this alternative, the goal is to insert the nodes into the sequences as soon as possible and prevent the nodes to be inserted between two nodes that are already

members of the sequence. Otherwise, it would mean that the new inserted nodes could be inserted anywhere and perhaps postpone the already inserted nodes.

It works as follows: The selected node is the one with the smallest *est*. We take the smallest *lct* to break ties. Then it is inserted at the earliest point in the sequence where an insertion is possible. To avoid that nodes are inserted before this one, all possible insertions of non-member nodes before this node will be deleted. This figure 6.1 represents the different states of a sequence. The first step depicts the possible insertions. We suppose that the node $a$ has the earliest starting time. It is inserted in the sequence, and the possible insertions between the begin node $\alpha$ and $a$ are removed. The last schema shows the state of the sequence and the possible insertions after the insertion.

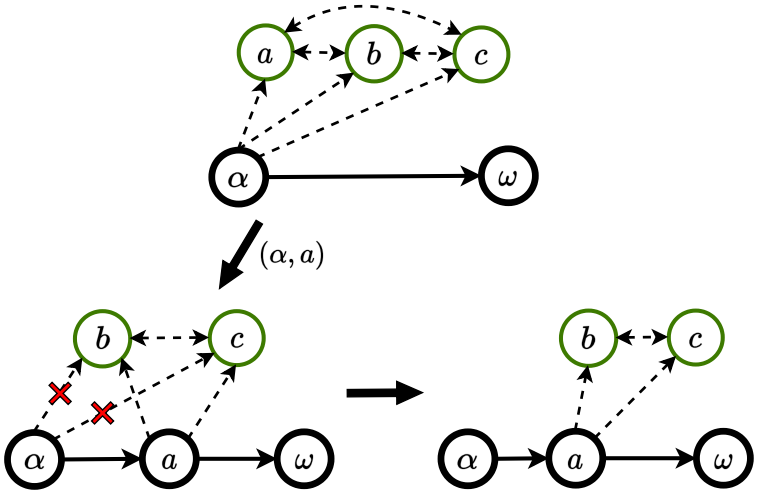



Figure 6.1: Changes in the sequence state for the insertion of the node with smallest *est* $a$, after node $\alpha$

### Postponed Set

In this variant, the idea of postponing the nodes was kept from the algorithm of the paper, hence the name. The main procedure of the algorithm is the following :

All nodes start as initially non-postponed.

- Select the non-postponed insertable node with smallest Est.
- Create two branches : insert the node on the first branch and postpone the insertions on the second branch. Remove all insertions that could occur just before the inserted node to prevent it from being delayed.
- Back to first step.

If all nodes are postponed, stop the search. A postponed node is allowed again for insertion once a predecessor node has been inserted. This idea tries to mirror the *SetTimes* algorithm, where a node can be inserted again once its start time would have changed.

#### Schedule Early

This branching heuristic works by selecting the task with the lowest *est* (variable selection) and making branches for all member insertions of this node (value selection). The branches are then sorted by the order of the insertion in its sequence, such that the latest insertion is performed first.

## 6.2 Relaxation heuristics

In this section, we will introduce multiple relaxations heuristics that we implemented for our model of the Job-Shop problem with sequence variables. To have a consistent state after relaxation, all heuristics maintain the machine order of the unrelaxed nodes.

Most figures in this section represents selections of nodes to be relaxed made by the relaxation heuristics. When not explicitly mentioned, the instances used to represent the selection, are ft06 and ft10 from [7].

### 6.2.1 Relaxation Random

The *relaxation random* randomly keeps a number of nodes up to a said percentage. This quite trivial relaxation does not consider any similarities between the nodes and will thus be used as a reference when evaluating the other relaxations. Figure 6.2 shows a selection of tasks to be relaxed.

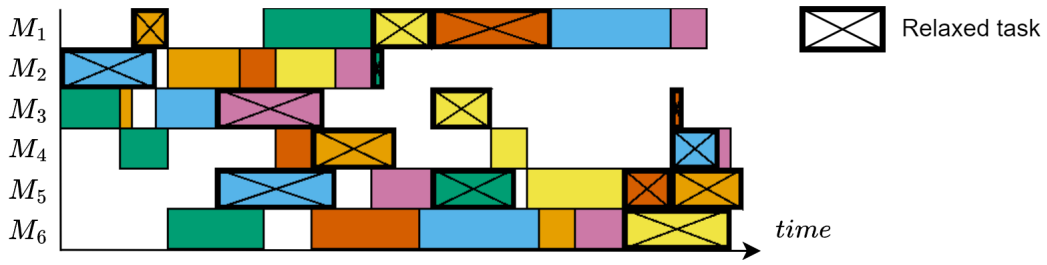



Figure 6.2: Example of selection of the *relaxation random* with 60 percent of tasks to be kept.

### 6.2.2 Relaxation Job

The *relaxation job* is based on the *resource based* relaxation form [4] which is a heuristic seeking to remove tasks (nodes) that executes on the same resource. The similarity chosen is thus tasks that execute on the same job. When called, the heuristic relaxes a defined number of jobs randomly selected. The idea behind this relaxation heuristic is to enable tasks to find more suitable position in the sequences such that the makespan is minimized. Figure 6.3 depicts a selection of jobs to be relaxed.

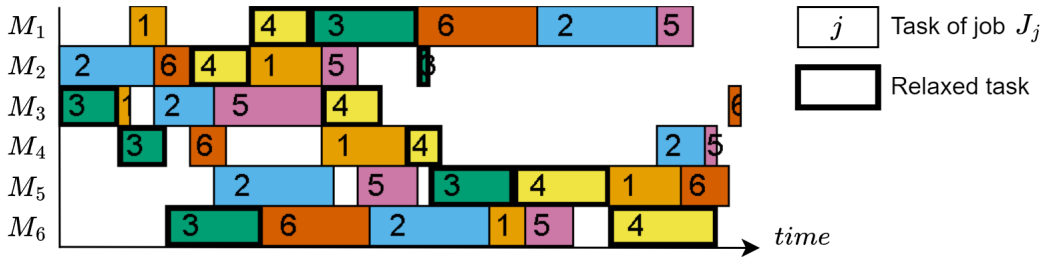



Figure 6.3: Example of selection of the *relaxation job* with 2 jobs to be fully relaxed. Here, the selected jobs are the jobs with identifier 3 and 4.

### 6.2.3 Relaxation Machine

The *relaxation machine* is similar to the previous relaxation in terms of inspiration and workings, the variation is that machines are relaxed in the place of jobs.

However, this relaxation could suffer from the definition of the Job-Shop problem itself if not enough machine are relaxed because job are performed in a defined order and assign a task on each machine. If the predecessor and successor on a job of a relaxed task are not relaxed, the said task could be forced to stay fixed temporally. Looking at figure 6.4, we can see an execution of the relaxation machine with one machine relaxed leading to no possible decrease of the makespan. Indeed, the relaxed task $T_{3,2}$ is fixed between $T_{3,1}$ and $T_{3,3}$ because it is the only place it can take.

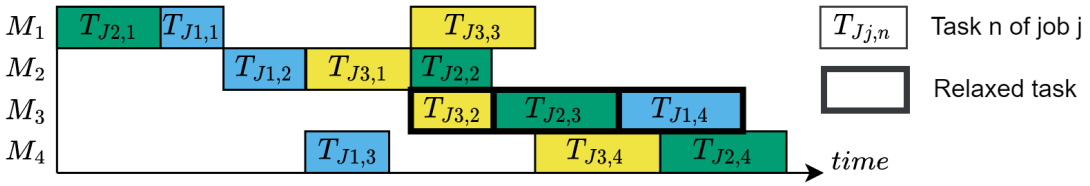



Figure 6.4: Example of *relaxation machine* with one machine relaxed. On this example the machine $M_3$ is relaxed, but it leads to no improvements in the minimization of the makespan.

### 6.2.4 Relaxation Order on Job and Order on Machine

The *relaxation order on job* and *order on machine* are based on the *relaxation random time window* from [4]. This heuristic seeks to relax nodes that are likely to be conflicting in the same in a specific time slot. As we can see on the figure 6.5, tasks of the instance ft06 [7] with the same order on their jobs have a tendency to be regrouped on the same machines.

This means that these relaxations can be effective for Job-Shop problems where the jobs have similar workflow, i.e., the jobs of the problem define tasks that are to be performed on similar machines with a similar order.

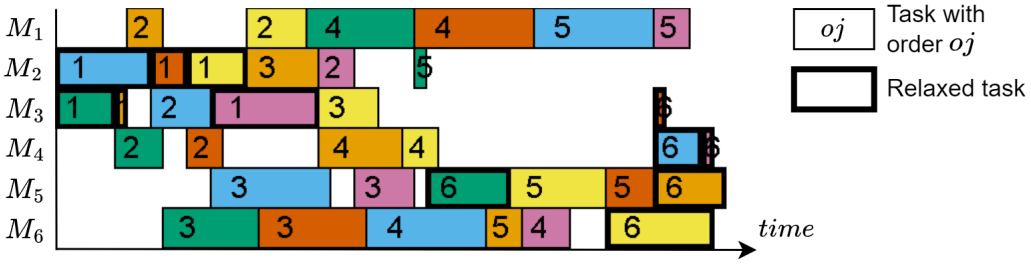



Figure 6.5: Example of selection of the *relaxation order on job* with 2 orders to be relaxed. Here, tasks with order 3 and 4, are relaxed.

### 6.2.5 Relaxation Critical Path

The *relaxation critical path* focuses on relaxing tasks from the critical path (section 1.5) of the solution. The motivation behind this heuristic is that since the critical path is defined as the set of critical tasks that dictates the makespan, not relaxing any critical tasks will prevent any minimization of the makespan.

However, only counting on the critical tasks for this relaxation could not be sufficient. Indeed, the number of the critical tasks could be small enough that it

does not allow the LNS to make progress on the minimization of the makespan if they are not able to change position in their sequences. On the contrary, as the makespan gets minimized, the size of the critical path could increase outsize an acceptable rage due to the tasks becoming increasingly close to each other.

To solve those issues and to control the number of tasks relaxed, the relaxation *Critical Path* takes two parameters : *minToRelax* and *maxToRelax* to control the minimum and maximum number of relaxed tasks, respectively. The critical tasks are randomly chosen for relaxation until *maxToRelax* nodes are relaxed, If there are less than *minToRelax* critical tasks, the relaxation selects randomly non-critical tasks for relaxation until meeting this threshold

The algorithm 10 shows the working of the relaxation. It works as follows (all ratios are in the range $[0, 1]$).
First, the critical path of the current best solution is computed (line 2) with the ratio of critical tasks (line 3). Next, we compute the ratio of critical tasks to relax (line 4) and the ratio of non-critical tasks to relax if needed (line 7). Finally, we iterate over all tasks in the problem (line 9-18) and relax them accordingly (randomly if needed).

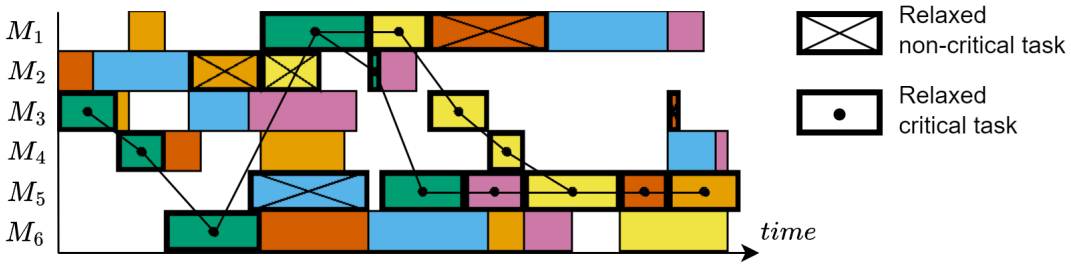



Figure 6.6: Example of selection of the *relaxation critical path* with a *minToRelax* and *maxToRelax* of 0.5. The lines between the critical tasks shows the precedence of the critical path.

**Algorithm 10** *relaxation critical path*

**Require:** $minToRelax, maxToRelax, SqList, problemInstance$

```
1: nTasks ← total number of tasks of problemInstance
2: criticalPath ← set of critical tasks for the current best solution
3: criticalRatio ← size(criticalPath) / nTasks
4: criticalToRelax ← min(1,maxToRelax/criticalRatio)
5: nonCriticalToRelax ← 0
6: if criticalToRelax = 1 then
7:    nonCriticalToRelax ← (minToRelax − criticalRatio)/(1 − criticalRatio)
8: end if
9: for Sq ∈ SeqList do
10:   for task ∈ Sq do
11:      rand ← random ∈ [0, 1]
12:      if task ∈ criticalPath and rand ≤ criticalToRelax then
13:         relax(task)
14:      else if task ∉ criticalPath and rand ≤ nonCriticalToRelax then
15:         relax(task)
16:      end if
17:   end for
18: end for
```

### 6.2.6 Shaw Relaxation

As presented in [23], the goal of this relaxation is to relax nodes with high similarity. The similarity can be defined in different ways depending on the problem. For our problem, we think it could be good to relax nodes that are on the same machine, they have approximately the same time window and the order of the corresponding activities is more or less the same. Many other similarity criteria between two nodes can be defined.

This is the pseudocode 11 for this relaxation. There are two arguments for this function: the number of nodes to relax and a parameter $p$. The $p$ parameter represents how similar the selected node is compared with the previously selected node. The closer it is to infinity the more similar the node will be, and the closer it is to 1 the more randomly the node will be chosen. The randomness is present to avoid selecting at each relaxation the same nodes.

**Algorithm 11** Node selection for *Shaw relaxation*

**Require:** $n \geq 0, p \geq 0, relatedMatrix$
**Ensure:** Get n nodes to be relaxed
1: $idx \leftarrow$ empty list
2: $v \leftarrow$ random $\in \{0, ..., n\}$
3: $node \leftarrow idx[v]$
4: $idx \leftarrow v$
5: **while** idx.size() $< n$ **do**
6: $v \leftarrow$ random $\in \{0, ..., idx.size()\}$
7: $orderNodes \leftarrow$ idx.sortby(- relatedMatrix[node])
8: $r \leftarrow$ random $\in [0, 1]$
9: $i \leftarrow pow(r, p)\cdot$(orderNodes.length - 1)
10: $idx \leftarrow$ orderNodes[i]
11: **end while**

This relaxation has three parameters that will change its behavior and influence its performance : the number of nodes to relax, the value of $p$ and maybe the most important one the criterion of node similarity.

We will use a criterion taking into account the time window and the order :

$$criterion(i, j) = overlap(i, j) + order(i, j) \quad \forall i, j \in \overrightarrow{S}$$

where,

$$\begin{aligned} overlap(i, j) &= abs(start_i.min() - start_j.min()) \\ &\quad + abs(start_i.max() - start_j.max()) \\ order(i, j) &= abs(order_i - order_j) \end{aligned} \tag{6.1}$$

where $order_i$ represents the order of the task $i$ and $start_i$ represents the start variable for the task $i$

To illustrate how it works, this figure 6.7 shows the relaxed nodes for $p = 100$ to remove a maximum of randomness. We can see that the relaxed nodes all have an order close to each other. In reality, the value of p will never be that large to avoid getting stuck around the same type of nodes, and so a smaller value allows us to add some randomness.

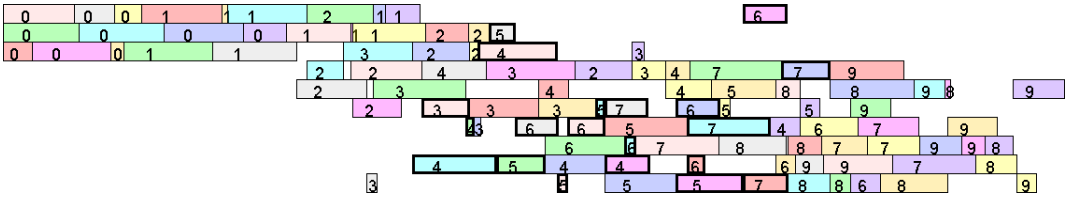


Figure 6.7: Figure representing the relaxed nodes (larger border) decided by Shaw's algorithm with parameter $p = 100$ and $nToRelax = 20$. The job order is written on the task.

### 6.2.7 Relaxation Alternate

The *relaxation alternate* is a meta-relaxation which purpose is to alternate through a list of multiple relaxation heuristics. The heuristic iterates over the list of relaxations at each relaxation phase and execute the selected heuristic. All the relaxation algorithms introduced in this chapter can be used by the *relaxation alternate*.

The motivation behind this heuristic is that it could happen that given a solution, a relaxation heuristic would be unable to find tasks to relax such that the search is able to find new solutions. It thus makes sense to alternate between multiple relaxations to variate the similarities of relaxed tasks and increase the chance of finding new solutions if a relaxation heuristic becomes ineffective.

# Chapter 7

# Results and analysis

## 7.1 Experimental setup and Job-Shop instances

### 7.1.1 Setup

All data presented in this section were calculated on an M5 Amazon EC2 instance, which has the following features: Intel Xeon Scalable processor with new Intel Advanced Vector Extension (AVX-512) instruction set, 32vCPU, 128 GiB memory. The operating system is of the Ubuntu family. The code executes in the Java 17 language and uses the framework of the MiniCP solver [18].

### 7.1.2 Job-Shop Instances

The instances used to perform the evaluations are the ft10 from [7], multiple instances from [16] and the existing instances of Mini-CP available in [19]. These last instances are named following the pattern "jobshop-$j$-$m$-$i$" where $j$ denotes the number of jobs, $m$ the number of machines and $i$ an identification number for the instance. For readability purpose, we will refer to those instances with the short format "$j$-$m$-$i$". We also define the size of an instance to be defined by the number of job and machine and is written "$j$-$m$".

## 7.2 Assessment models and searches

To evaluate the branching procedures and the model of the job-shop using the sequence variables, we will compare it to two existing models, not using the sequence variable data-structure and therefore not working with insertions, as well as different branching schemes.

**Models and naming conventions** In the remainder of this chapter, the first-fail heuristic will refer to the specific *smallest-value-first* branching procedure [1] but applied to different variables.

The below-described models all use *LessOrEqual* constraints to ensure the precedences between tasks of a same job. They also all use a disjunctive constraint to ensure the non-overlap of the execution-windows of the activities. Still, two version of the disjunctive constraint are used, inducing two different models. Both disjunctive constraint versions refer to the ones introduced in section 2.2.5 and provided by [18].

The models and search combinations used are the following :

- Using the classic disjunctive with developed filtering algorithms,
  - ClassicDisjunctiveStart – This setting uses a first-fail branching on the start variables.
- Using the binary decomposition of the disjunctive constraint,
  - DisjunctiveBinFFStart – Is a setting that uses a generic first-fail applied on the start-variables of the problem.
  - DisjunctiveBinFFPrecedences – Uses a first-fail branching that branches on the precedences between the tasks.

    In more detail, the branching procedure progressively fixes the precedence between pairs of tasks. The list of precedences (modelled as boolean variables) are initially ordered such that the precedences who have a small slack are branched first. The slack of a precedence is a metric based on the slack of each activity after having fixed the precedence. See [19] or [25] for a more detailed explanation of the slack computation for the binary disjunctive.

    This ordering is done once before starting the search, it is only based on the initial schedule where no task is yet fixed.
  - DisjunctiveBinSlack – Is a setting close to *DisjunctiveBinFFPrecedences*. The branching is done on task precedences, but not in a first-fail manner. The branching procedures select the precedence which has the smallest slack given the current state of the problem during the search (variable selection) and fixes the precedence in order to maximize the remaining slack on the first branch, the second branch containing the inverted precedence (value selection).

## 7.3 Constraints comparison

The first analysis that we can do is the impact of the constraints on the execution time required to explore completely the search tree in optimization search and on the size of the tree.
To obtain the data, we used five instances of size 5-5, five of size 6-6 and five of size 7-7.

We have three constraints to consider, who lead in our case to four configurations:

- using the minimal constraints required to solve the problem (channeling constraint only)
- using the disjunctive constraint
- using the filteringInsertions filtering
- combining the disjunctive constraint and the filteringInsertions filtering

To do this analysis, the *first-fail* branching and the *full slack* branching order have been used.

**Execution time** First, we can compare execution times according to the constraints used in the model. We can observe in this graph 7.1 that adding the disjunctive constraint leads to a slower execution time than having only the minimal constraints. Just as clearly, we can see that using the filteringInsertions filtering alone enables instances to be resolved more quickly than with any other combination. Finally, it is unclear whether the combination of the disjunctive constraint and the filteringInsertions filtering is to be preferred over not using any constraint at all. This suggests that the combinations of both constraints is not beneficial for the overall performance.

Considering only the execution time, we can conclude that the filteringInsertions filtering allows improving the search performance.

**Number of choices and failures** Now, we will attempt to understand the source of this difference in execution time using the graphs of figure 7.2 and figure 7.3, which respectively represent the number of choices and the ratio of failures to choices in the search tree for different instances, based on the constraints used in the model.

In figure 7.2, we can observe that the two slowest models (minimal and disjunctive) also have the highest number of choices. This is consistent with the idea that the configurations having the highest number of nodes (large trees) tend to have the highest execution time.

Regarding the models with the least number of choices, we observe that adding the disjunctive constraint to the filteringInsertions filtering allows reducing the number of choices made.

Looking now at figure 7.3, we can formulate an additional argument for the slowness of minimal and disjunctive settings. Indeed, we observe those models have the highest failure ratio which, combined with the high number of expanded nodes, should lead to a long search requiring time-consuming backtracking.

Regarding the two small-node-expanding models on the other side (filteringInsertions and filteringInsertions with disjunctive), we see that unlike for the number of choices, it is the filteringInsertions model that performs best regarding the number of failures.

**Conclusion** In conclusion, we aim to find a compromise between execution time and the size of the search tree. The filteringInsertions configuration performs best because it has a low failure on choice ratio and a low number of choices, even if the number of choices is even lower when adding the disjunctive constraint.

Therefore, in the upcoming analyses, we will include the filteringInsertions filtering in the model, assuming it is the best choice regardless of the branching type, although we have not verified this assumption.

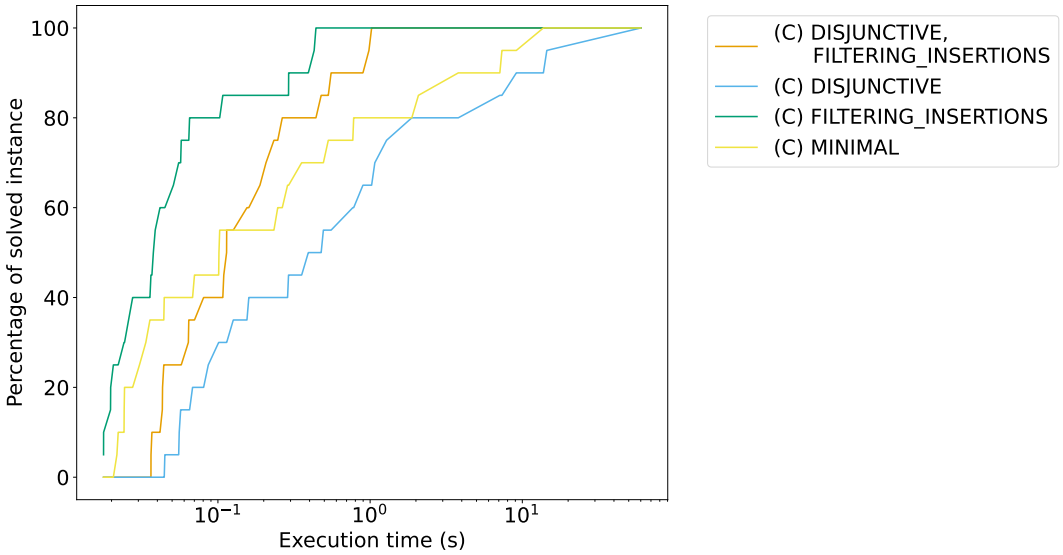



Figure 7.1: Comparison of the number of solved instance according to the execution time for each possible constraint combination for default branching that includes ***first-fail*** **branching** and ***full slack*** **branching order**

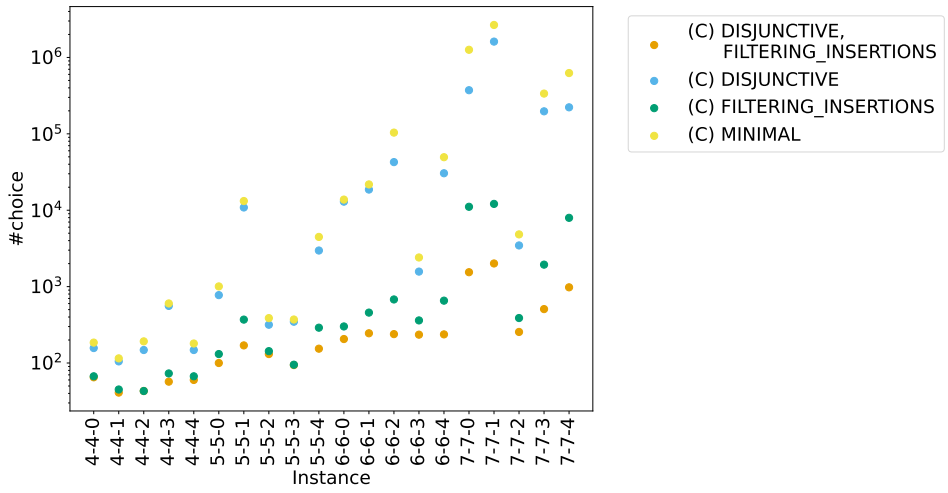



Figure 7.2: Comparison of the number of choices according to instances for each possible constraint combination for default branching that includes ***first-fail*** **branching** with ***full slack*** **branching order**

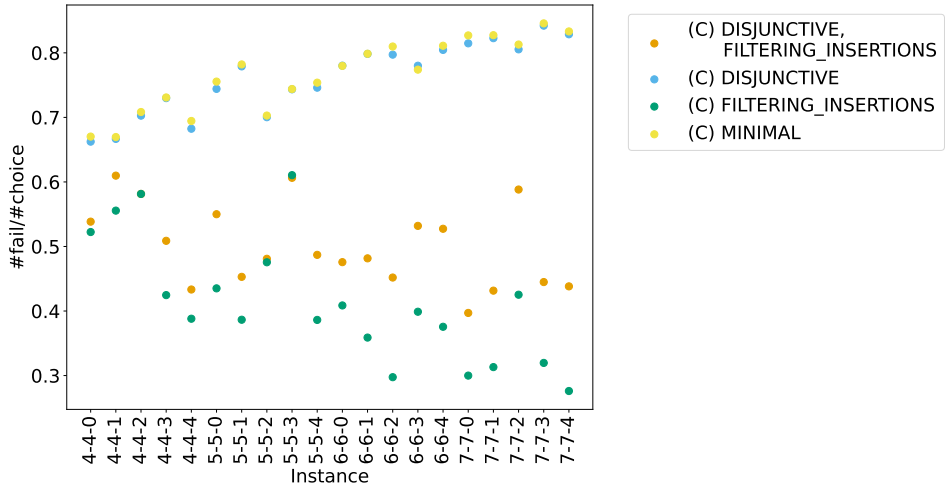



Figure 7.3: Comparison of the ration between the number of failures and the number of choices according to instances for each possible constraint combination for default branching that includes ***first-fail*** **branching** with ***full slack*** **branching order**

## 7.4 DFS comparison

Now that we know the constraints to be used in the model, we'll concentrate on evaluating the effectiveness of the different branching strategies, branching orders and sequence selectors. Initially, we assess the impact of each branching strategy

on the execution time for different branching orders. Subsequently, we identify the optimal branching order for each branching and analyze the influence of the sequence selector with and without its application. Finally, a comprehensive comparison is conducted, considering each branching with its respective best branching order and sequence selector (if applicable).

To obtain the data, we used five instances of size 5-5, five of size 6-6, five of size 7-7 and five of size 8-8. Except for the last comparison, we do not include the data set of size 8-8. We also set a timeout after 1000 seconds.

### 7.4.1 Branching order comparison

For this analysis, we compared the four branching ordering based on the three branching procedures accepting a branching ordering : *min-max-slack*, *maximum duration* and *first-fail* branching procedures. We will progressively analyze the three graphs (7.4, 7.5 and 7.6) and try to motivate the observed results.

**Full-slack dominance** In general, regardless of the branching type used, we can say that the *full-slack* branching order is the best approach. With this branch ordering, 100% of the instances can be solved within an optimized complete search, ranging from 0.8 to 3 seconds depending on the branching type. As a consequence, the *left-slack* ordering is always less efficient compared to *full-slack*. For both the *first-fail* and *min-max-slack* branching procedures, the *left-slack* branching order takes even the longest time to reach 100% of solved instances. We expand below on the possible grounds of such performances.

The *left-slack* branching order was designed to favor insertions not increasing the makespan by not considering the delay in the slack calculation. The drawback of this approach is to ignore the potential makespan increase when there is no other choice than delaying the next activity. This trade-off depends on the state of the schedule during the search. The almost generalized lower performance of the left-slack heuristic suggests delaying the next activity is frequent, making the left-slack heuristic less suited.

Regarding the other branching order, a more branching-specific analysis has to be conducted.

***Low MVI* branching** We observe on the graphs that the *low MVI* branch ordering performs best when combined to the maximum duration branching. The min-max-slack branching leads to a less efficient combination, but the first-fail branching is the worst case.

Looking at its metric, the *low MVI* branching order is the most accurate measure, as it performs insertions to directly observe the impact on the makespan.

However, it does not provide any information when the makespan does not change, which can potentially hinder its performance in the search process.

The *first-fail* branching procedure completely ignores the durations of the inserted nodes, as it selects the node with the smallest number of insertions. On the other side, the max-duration branching inserts primarily nodes with high duration, as the name suggests. This should imply that the makespan would often be increased when performing the insertions. Having insertions with increase of the makespan values the *low MVI* metric which could explain that the *low MVI* metric is more performant for the max-duration branching procedure compared to the first-fail branching procedure.

***Early* branching** The last branching order to consider it the *early* branching. It is expected to be less effective because inserting a node at its earliest position does not guarantee that no other nodes will be inserted before it. As the heuristic is not expected to be efficient but is quite close to the other heuristics in our experiment, we can deduce that the performance variation between our best heuristic and a bad one is quite small.

**Conclusion** In conclusion, for the upcoming data collection and analysis, we will use the *full-slack* branching order as the default ordering unless explicitly stated otherwise as this branching order has been observed to be the best in terms of execution time for solving instances, in most cases, regardless of the percentage of solved instances.

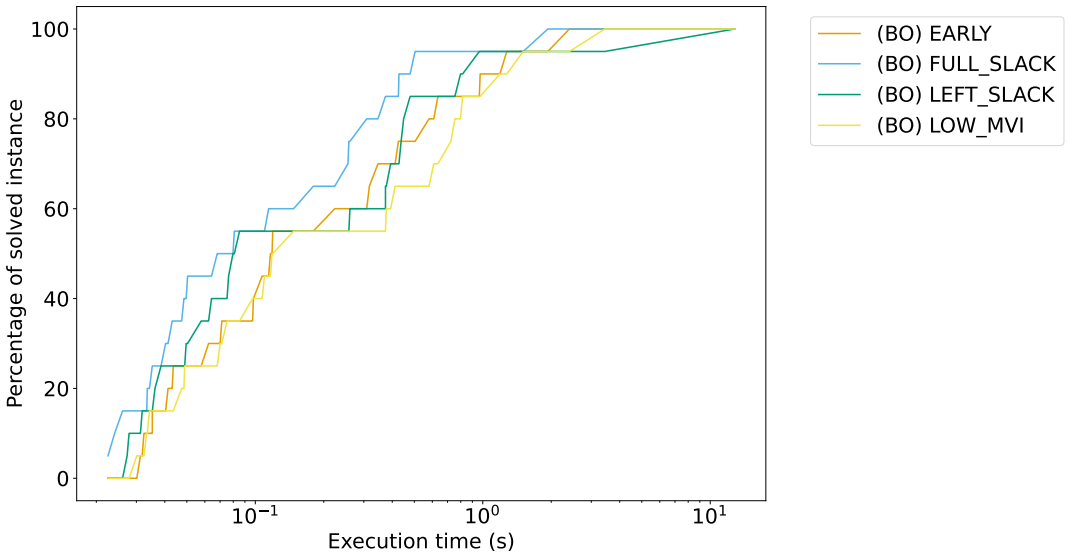



Figure 7.4: Comparison of the number of solved instance according to the execution time for each branching order for the ***first-fail* branching**

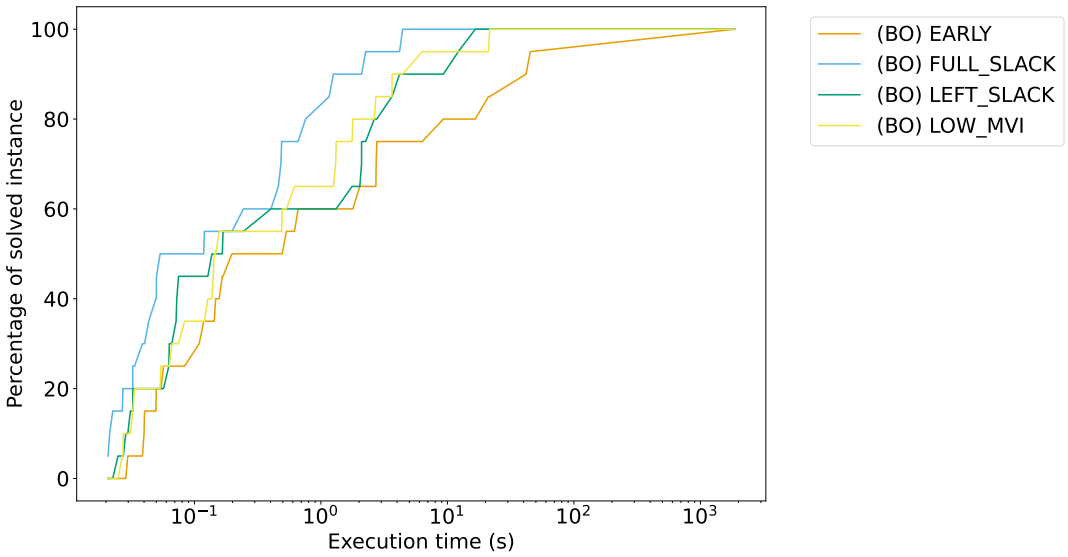



Figure 7.5: Comparison of the number of solved instance according to the execution time for each branching order for the ***maximum duration*** **branching**

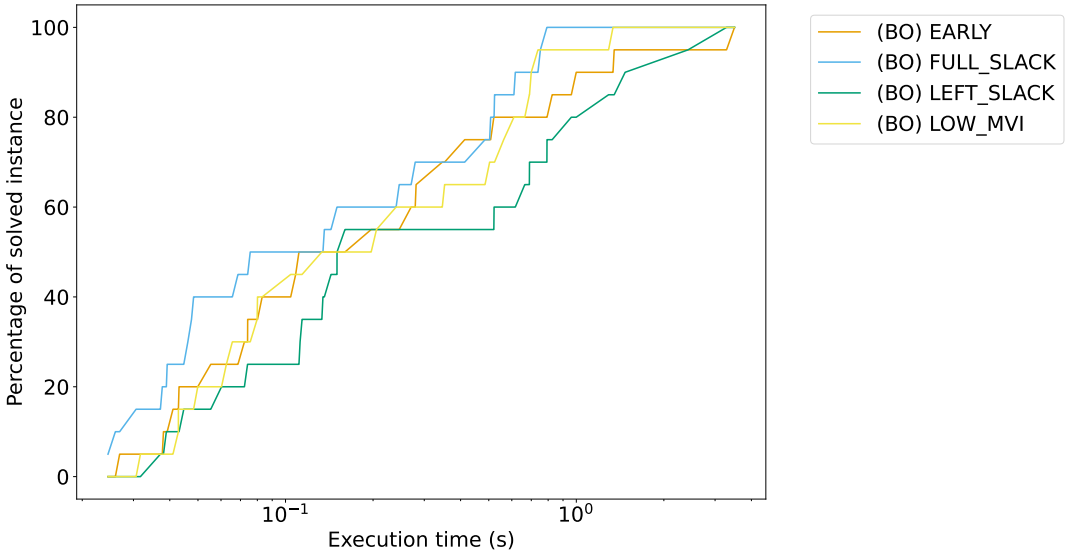



Figure 7.6: Comparison of the number of solved instance according to the execution time for each branching order for the ***min-max-slack*** **branching**

### 7.4.2 Sequence selector

We select the best branching order for each branching, *full-slack* branching order, to analyze the impact of the sequence selector on the execution time to make a

complete optimize search.

***First-fail* branching** For the *first-fail* branching, we can observe in the graph 7.7 that the use of the sequence selector has a bad impact on the execution time. We have a difference of at least 5 seconds to reach 100% of resolved instances between a search without any sequence selector and a search with one of these sequence selectors.

***Maximum duration*, *min-max-slack* and *regret MVI* branching** For the other three branching, we can see in the figures 7.8 7.9 7.10 that using the *less filled* sequence selector and not using a sequence selector have a fairly similar behavior in terms of execution time. For the *maximum duration* branching, the use of the *less filled* sequence selector solves 100% of the tested instances a few seconds faster, whereas for the other two branching procedures, using the sequence selector is a few seconds slower.

As we might have expected, the sequence selector that chooses the most filled branch, *most filled*, is quite slow in execution time. This selection will completely fill a sequence before moving on to the next one, which may have a negative impact on the makespan.

**Conclusion** For all branching procedures, we can conclude that not using a sequence selector is preferable, even if it is sometimes two or three seconds faster. The reduction of the number of nodes to be processed by the best node selection does not balance against the drawback of missing a better node to be selected for insertion.

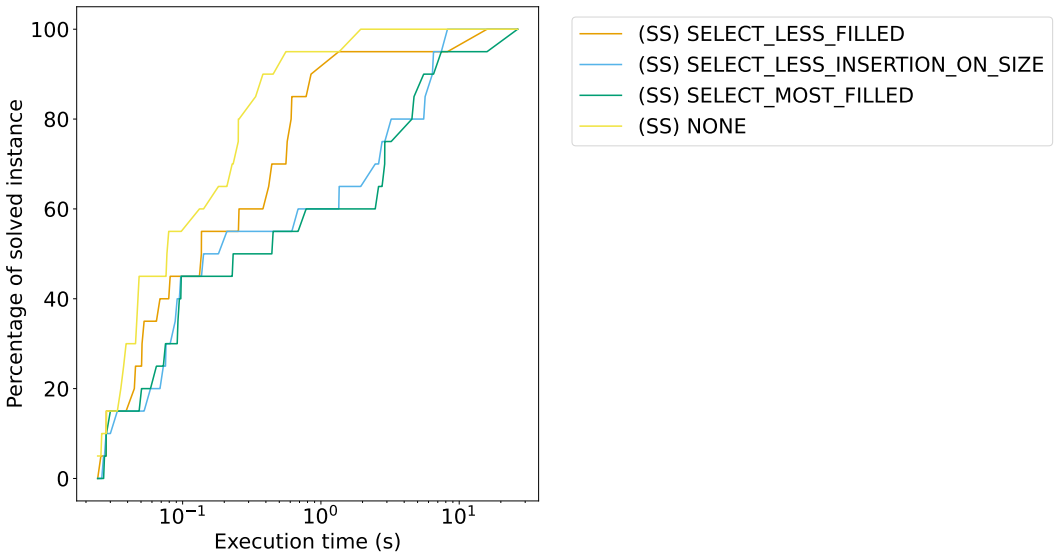



Figure 7.7: Comparison of the number of solved instance according to the execution time for each sequence selector for the ***first-fail*** **branching** with ***full slack*** **branching order**

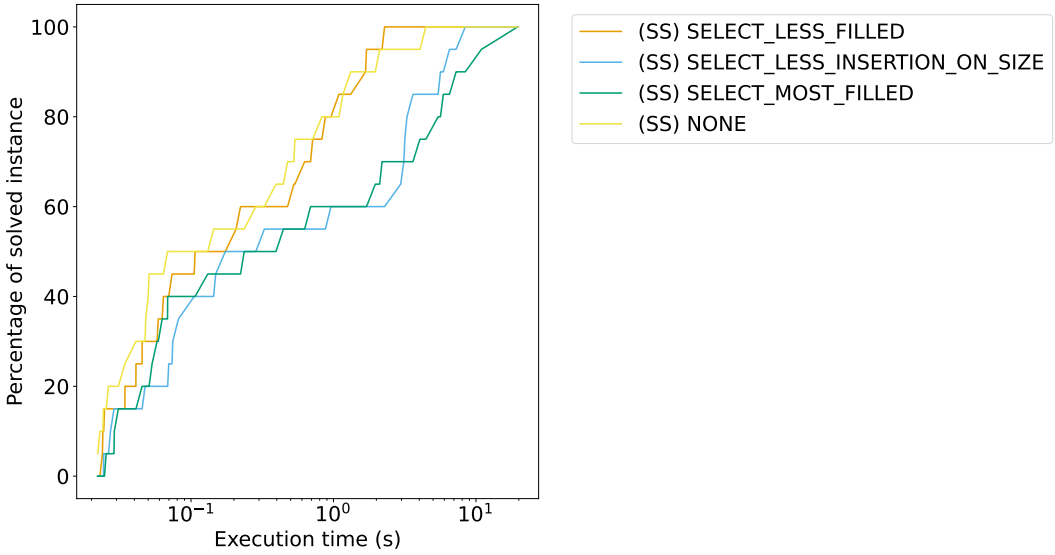



Figure 7.8: Comparison of the number of solved instance according to the execution time for each sequence selector for the ***maximum duration*** **branching** with ***full slack*** **branching order**

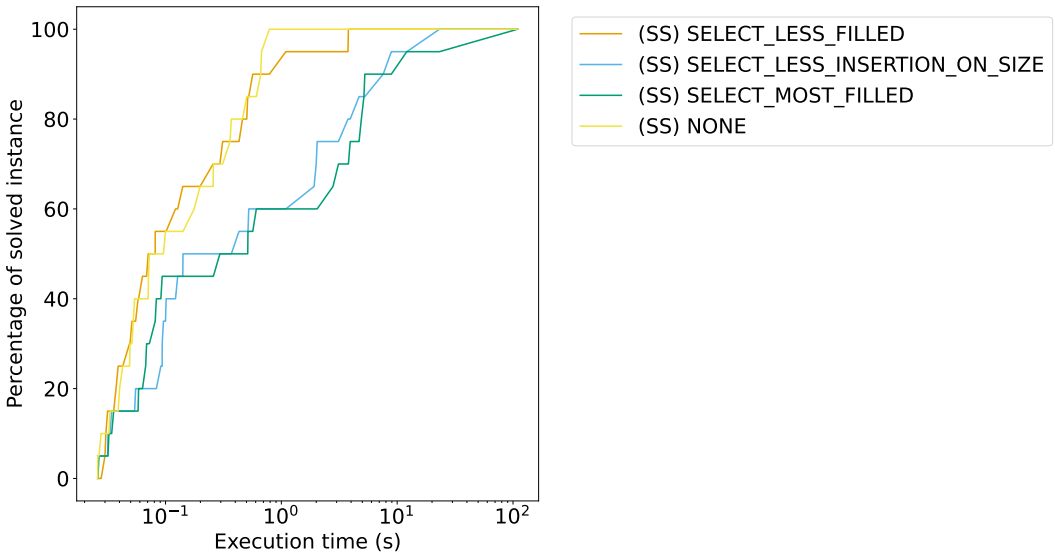



Figure 7.9: Comparison of the number of solved instance according to the execution time for each sequence selector for the ***min-max-slack*** **branching** with ***full slack*** **branching order**

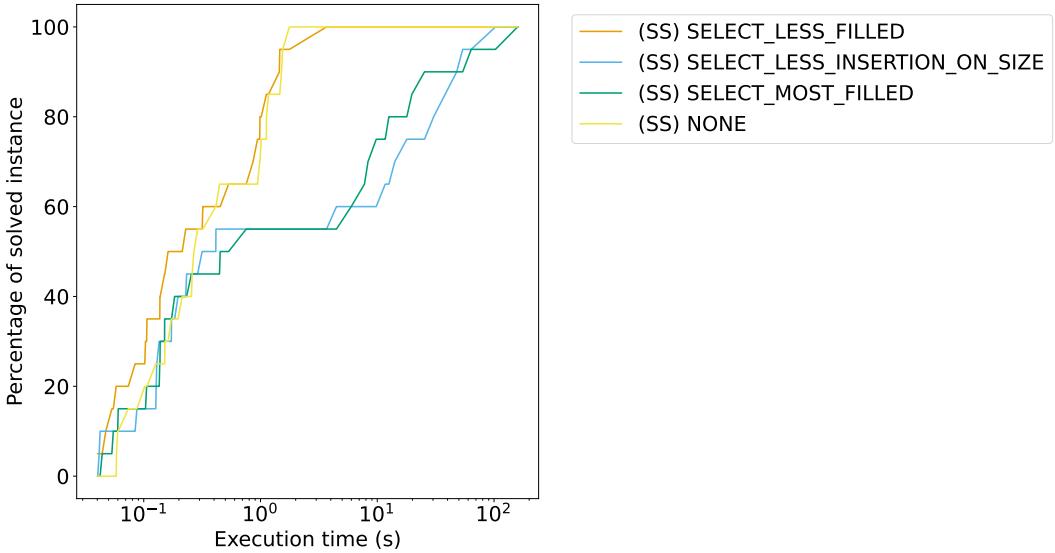



Figure 7.10: Comparison of the number of solved instance according to the execution time for each sequence selector for the ***regret MVI*** **branching**

### 7.4.3 Final comparison

To conclude this analysis, we will analyze the different branchings with their respective optimal branching order. We will compare all branchings with the

*full slack* branching order and without sequence selector, and also we will compare our branchings with the already implemented models introduced in section 7.2.

### Sequence branchings comparison

**Execution time** The first aspect we can compare between these branchings is the execution time required to solve 100% of the instances using an optimized complete search. We can observe in this graph 7.11 that, for up to 50% of instances solved, all branchings except for the *regret MVI* branching have very similar execution times. The *regret MVI* branching, on the other hand, is slightly slower by a few tenths of a second. Subsequently, this branching becomes the second fastest in reaching 100% of instances solved. The fastest branching to reach 100% is the *min-max-slack first* branching.

**Number of choices and failures** After comparing the number of choices 7.12 and the ratio of failures to choices 7.13, we can see that the *regret MVI* branching tends to have the smallest number of choices. However, if we look at the value of the ratio for this branching, it tends to be the highest compared to the other branchings. This indicates that a large number of branches in the search tree do not yield any results.

On the contrary, the *first-fail* and *min-max-slack first* branchings, which sometimes have a high number of choices, have a lower ratio.

In conclusion, if we were to select the best branching, we would choose the one with the fastest execution time, the lowest number of choices, and a ratio that tends towards zero. Unfortunately, none of the branchings satisfy all three criteria simultaneously. However, the *min-max-slack first* branching is a good compromise as it has the fastest execution time to reach 100% of instances solved. It also has a generally manageable number of choices and a relatively low ratio, averaging between 0.3 and 0.4.

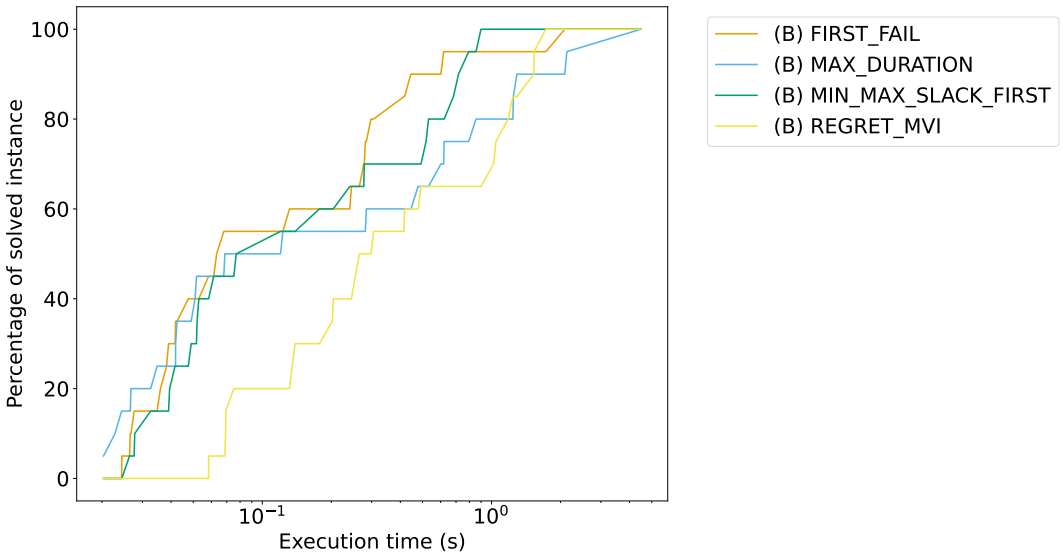



Figure 7.11: Comparison of the number of solved instance according to the execution time for each branching for ***full slack*** **branching ordering** and **without sequence selector**

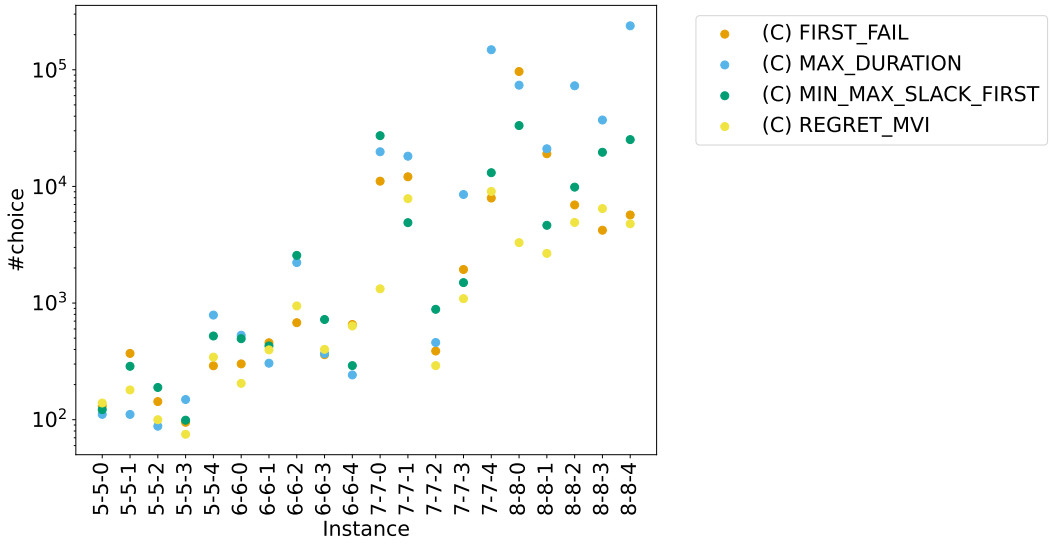



Figure 7.12: Comparison of the number of choices according to instances for each possible constraint combination for each branching for ***full slack*** **branching ordering** and **without sequence selector**

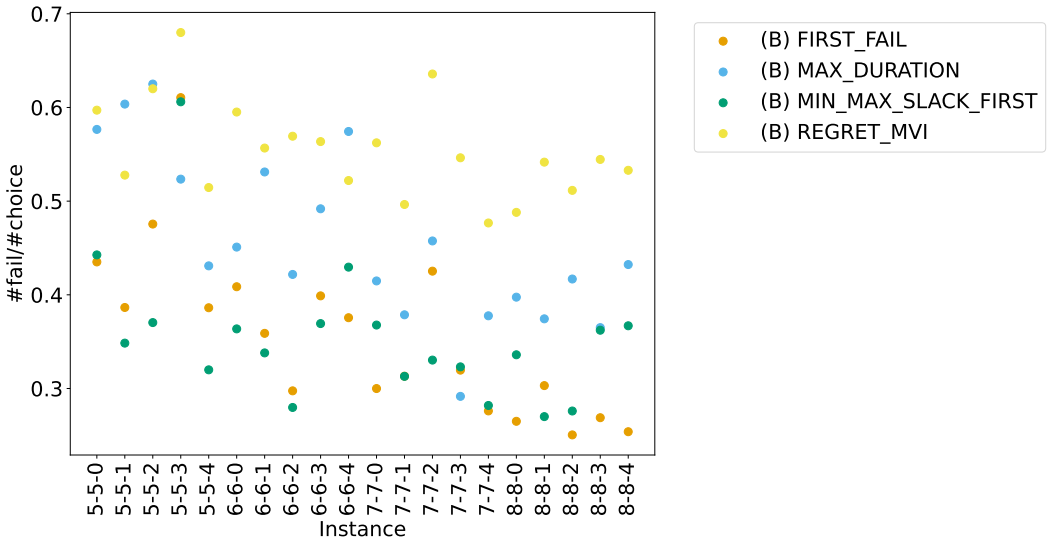



Figure 7.13: Comparison of the ratio between the number of fails and the number of choices according to instances for each possible constraint combination for each branching for ***full slack*** **branching ordering** and **without sequence selector**

### Cross-model branching comparison

Now, we compare our models with the various branchings to other already implemented models on the same instances as the other analysis, except the instances of size $8 \times 8$ because they take too much time.

We can observe 7.14 that two models are faster than ours. The fastest model is the *DisjunctiveBinSlack* model, which is based on the disjunctive binary constraint and a slack-based branching. Next, there is the *DisjunctiveBinFFPrecedences* model, which relies on the disjunctive binary constraint and a *first-fail* branching on precedences. Finally, we have all our models, which are faster than the two other implemented models based on a start-based branching.

We can conclude that our models are more efficient than models with branching based on start variables, but are less efficient by a few tenths of a second than the models based on the disjunctive binary constraint and a slack-based branching or *first-fail* branching on the precedences.

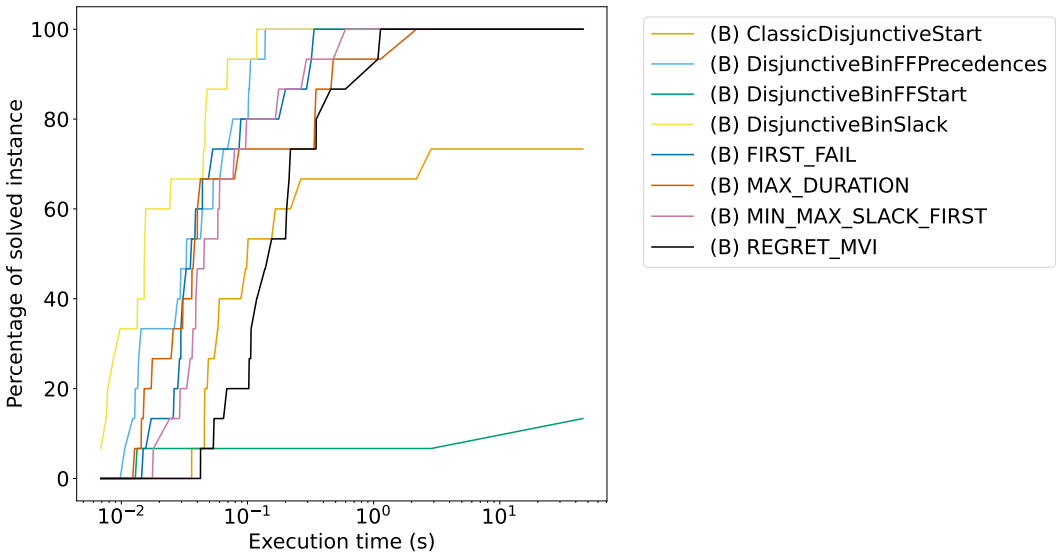



Figure 7.14: Comparison of the number of solved instance according to the execution time for each branching, that we have implemented, for ***full slack*** **branching ordering** and **without sequence selector** and for each

## 7.5 LNS comparison

Now that we have analyzed the branchings for a DFS (Depth-First Search), let's move on to analyzing the LNS (Large Neighborhood Search) approach. In this part, we will first compare the different DFS branchings that can be used to find the initial solution. Then, we will analyze and compare the various relaxations that we have implemented.

### 7.5.1 First solution

To find the first solution to begin the LNS, we will compare the different branchings discussed above and three branchings that have been implemented especially to find a good first solution. To compare all these types of branchings we will use a different metric : $\frac{finalMakespan - optimium}{optimum}$.

We can observe from this graph 7.15 that, in general, the branching *maximum duration* gives the first makespan closest to the optimum makespan value.

The *postponed set*, *earliest start* and *schedule early* branchings have the same values as first makespan. So these branches, although implemented differently, give the same first makespan value.

Since in future analyses we'll only be making comparisons on the *ft-10-10* instance, we'll use the branching that gives the best first makespan for this specific

instance, i.e., *min-max-slack first* branching.

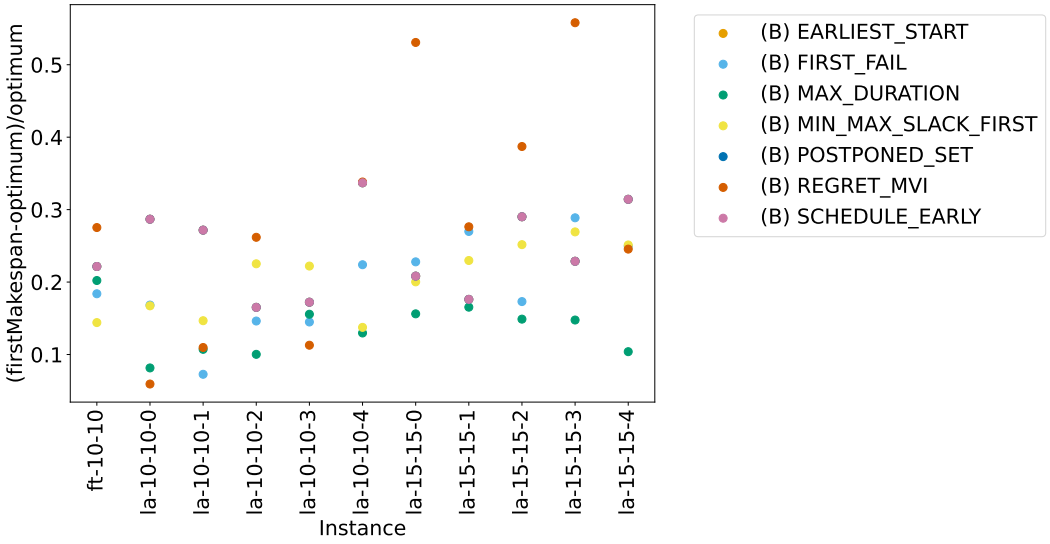



Figure 7.15: Comparison of the difference between the first makespan and the optimum divided by the optimum according to several instances for each branching

### 7.5.2 Relaxation comparison

Our goal is to find the best branching and parameter to achieve the best results. To do this, we will try to strike a balance between the execution time until the last found makespan and the value of that last makespan.
For all analysis, we fixed the number of failure limit to 1500 before the optimization phase stops, the maximal number of restarts to 400 and a timeout on the total execution time to 3 minutes. In addition, as the relaxations have a random factor, we perform the same search 21 times with a seed ranging from 0 to 20.

#### *Critical path relaxation*

The first relaxation we will analyze is the *critical path* relaxation, which was defined in the previous section 6.2.5.

**Execution time** Initially, we can observe that the execution time increases as we increase the values of the parameters (minToRelax, maxToRelax). This is logical because the fewer nodes we relax, the more the search encounters failure cases, resulting in both the failure and restart limits being reached quickly, without finding new solutions.

**Final makespan** Therefore, we observe that the relaxation parameters that enable fast search are also associated with the highest last makespan. With fewer nodes being relaxed, it becomes more challenging to reach new solutions.

**Conclusion** A good choice of branching and parameters is the *min-max-slack first* branching and $(60, 60)$ as parameters. Although we can observe that the optimal solution is never reached, the median value is one of the lowest regardless of the branching and parameters used. Additionally, the variation, due to the random factor present in the relaxation, is relatively low, ensuring a result close to the median. The execution time is also reasonable, ranging between 10 and 20 seconds 7.16, which is a good compromise to achieve a final makespan that has a high probability of being between approximately 940 and 950 7.17.

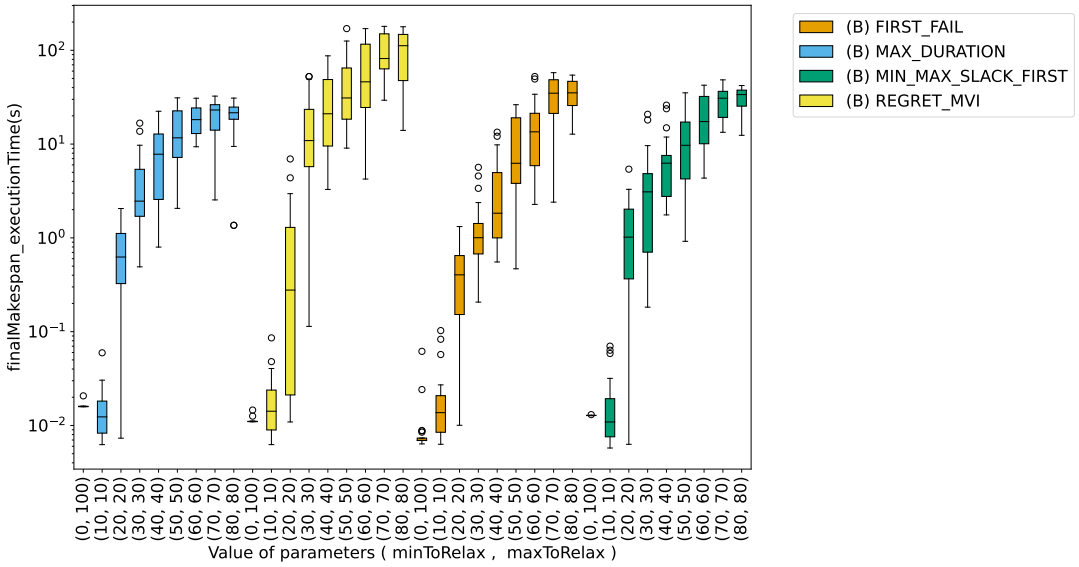



Figure 7.16: Comparison of the execution time of each branching according to the parameters for the ***critical path* relaxation** for the ft10 instance

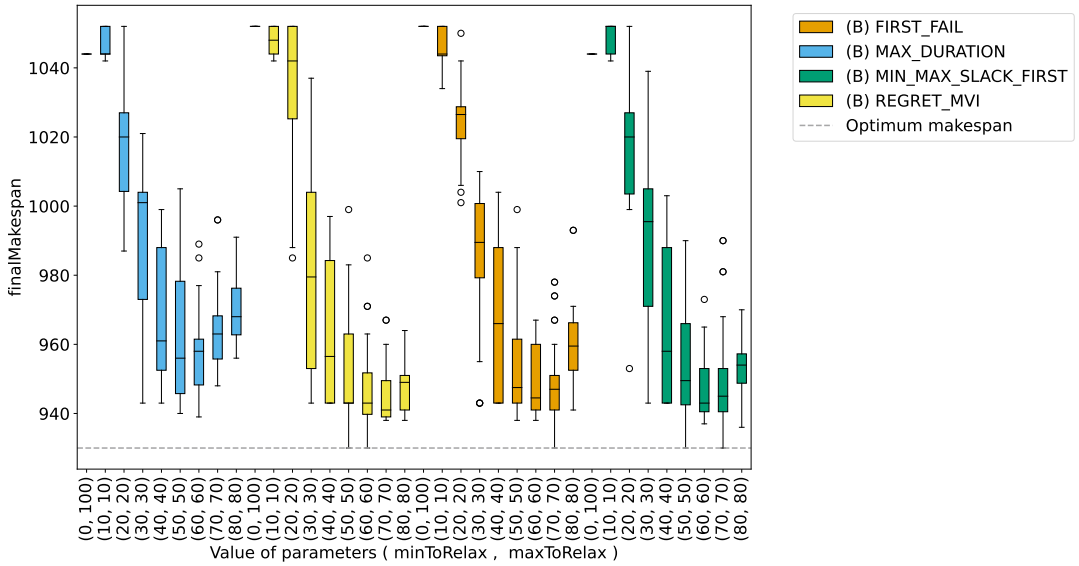



Figure 7.17: Comparison of the final makespan of each branching according to the parameters for the ***critical path* relaxation** for the ft10 instance

### *Relaxation random*

For the *relaxation random*, it's more difficult to distinguish a better branching and a good parameter, as this is a totally random relaxation for which we have a wide range of values between the first and third quartile 7.19. The best branching with the best parameter is again the *min-max-slack first* branching with 30% of node kept. Another possibility will be the *regret MVI* branching with 40% of nodes kept but the execution time to achieve the last found makespan is fifty seconds slower 7.18.

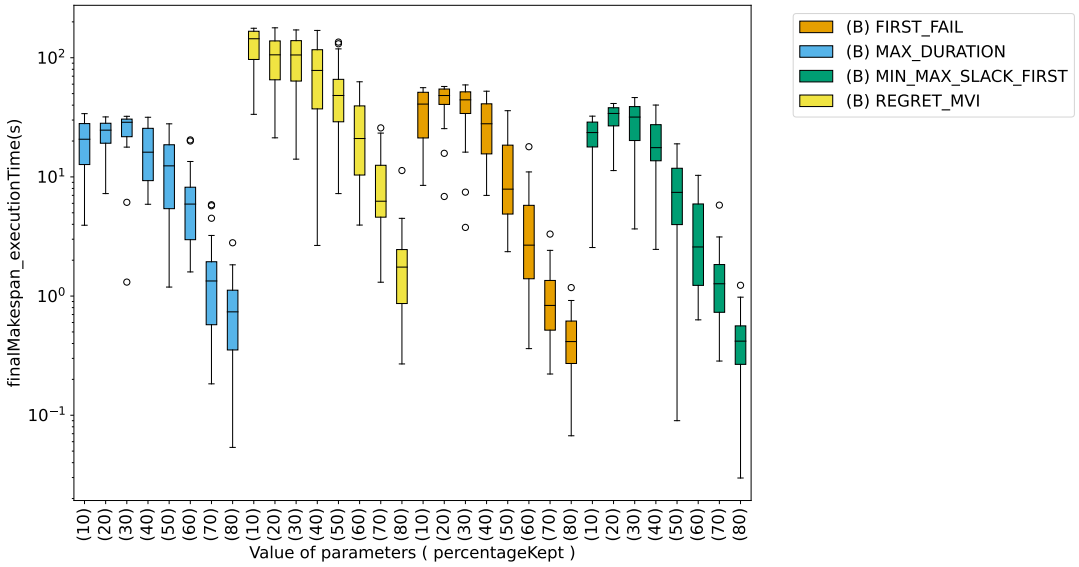



Figure 7.18: Comparison of the execution time of each branching according to the parameters for the ***relaxation random*** for the ft10 instance

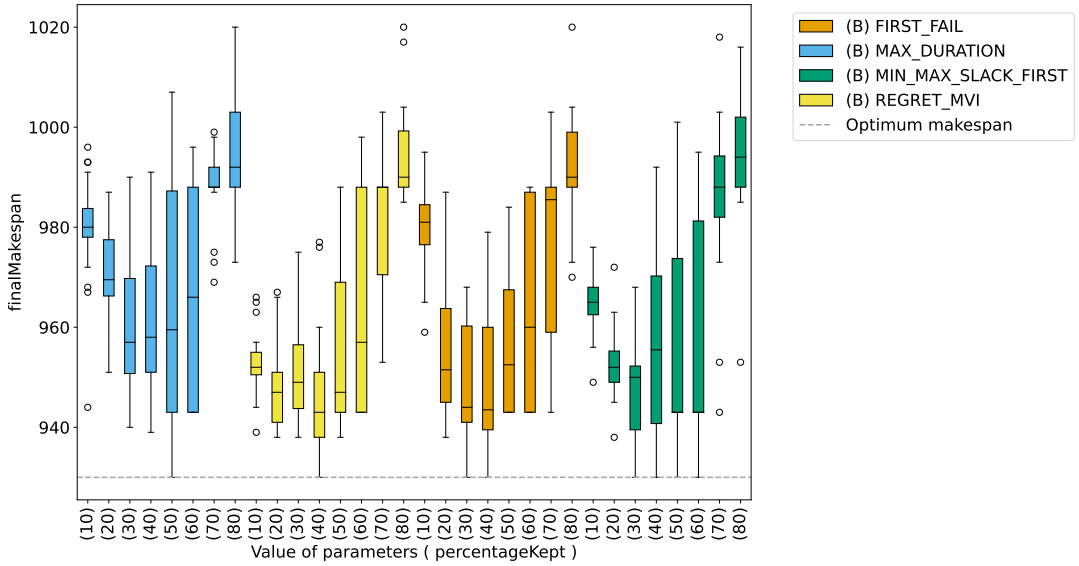



Figure 7.19: Comparison of the final makespan of each branching according to the parameters for the ***relaxation random*** for the ft10 instance

### *Relaxation job*

For this *relaxation job*, unlike the *random* relaxation, we observe in the graph 7.21 that for the same parameters, regardless of the branching, the final makespan

values are quite similar.

**Final makespan** For a parameter value of 4 or 5, it is almost guaranteed that the makespan value will fall within a very narrow range, as the values between the first and third quartiles are close together. We can already determine that we will choose either 4 or 5 as the parameter value.

**Execution time** Now, to determine which one and which branching, we will analyze the execution time 7.20. There is not much difference, except that the *regret MVI* branching is generally slower. Furthermore, it is preferable to keep the nToRelax parameter equal to 4, as the execution time is a few seconds faster. Additionally, the branching with the lowest median and smallest final makespan is the *min-max-slack first* branching.

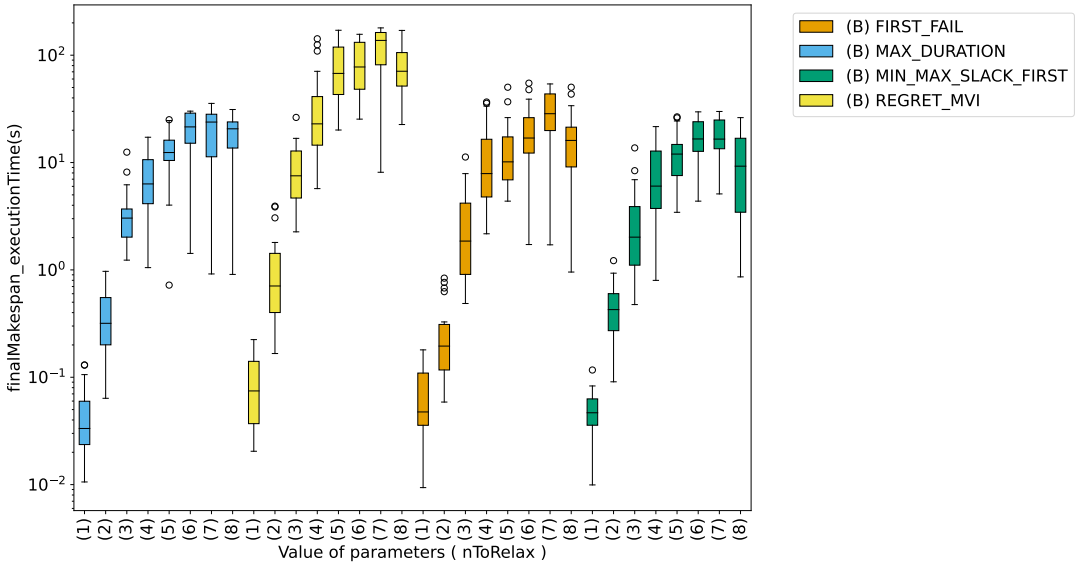



Figure 7.20: Comparison of the execution time of each branching according to the parameters for the ***relaxation job*** for the ft10 instance

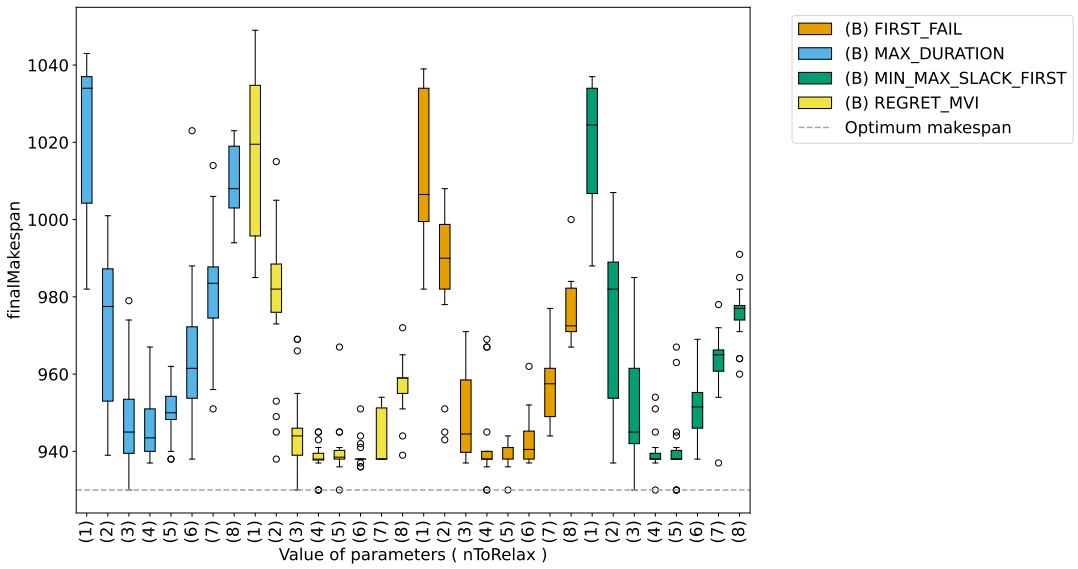



Figure 7.21: Comparison of the final makespan of each branching according to the parameters for the ***relaxation job*** for the ft10 instance

### *Relaxation machine*

For the *relaxation machine*, we can observe from the graph comparing the final makespan 7.23 that it is the opposite of the *relaxation job*. The range of values, including the first and third quartiles, is very wide. It is not really possible to determine the best branching with its optimal parameter.

This behavior was expected because relaxing nodes on the same machine when the number of machines to relax is small does not lead to significant improvements. Nodes often depend on multiple nodes that are spread across different machines, as demonstrated in the explanation of the critical path. When the number of machines is large, it becomes easier to achieve better solutions.

However, the results still remain highly random due to the random selection of fully relaxed sequences. This randomness contributes to the wide range of possible values for the final makespan.

**Execution time** Regarding the execution time, we can observe in this graph 7.22 that the time to reach the last found makespan is generally a matter of seconds.

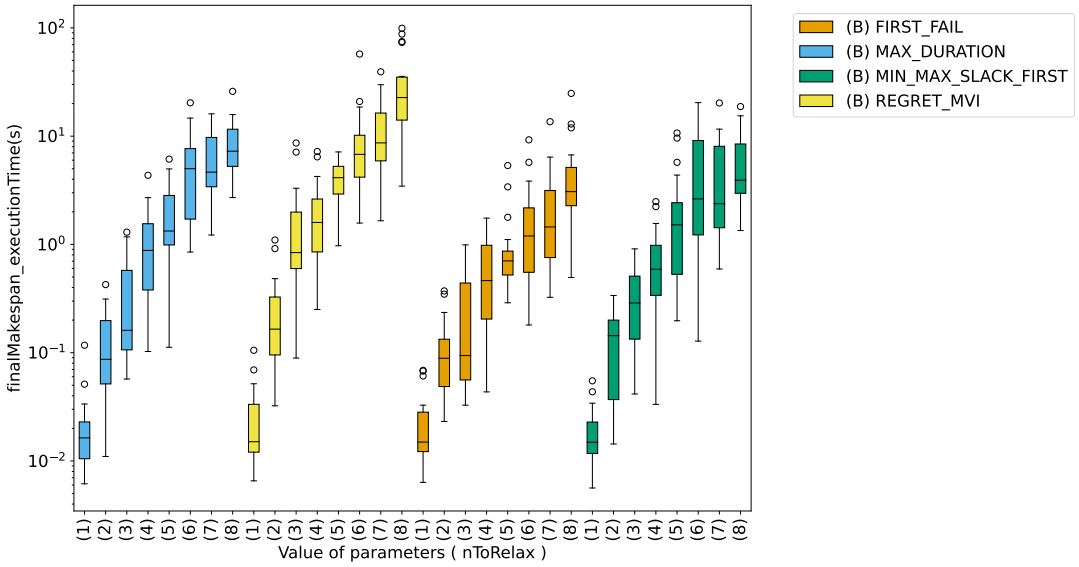



Figure 7.22: Comparison of the execution time of each branching according to the parameters for the ***relaxation machine*** for the ft10 instance

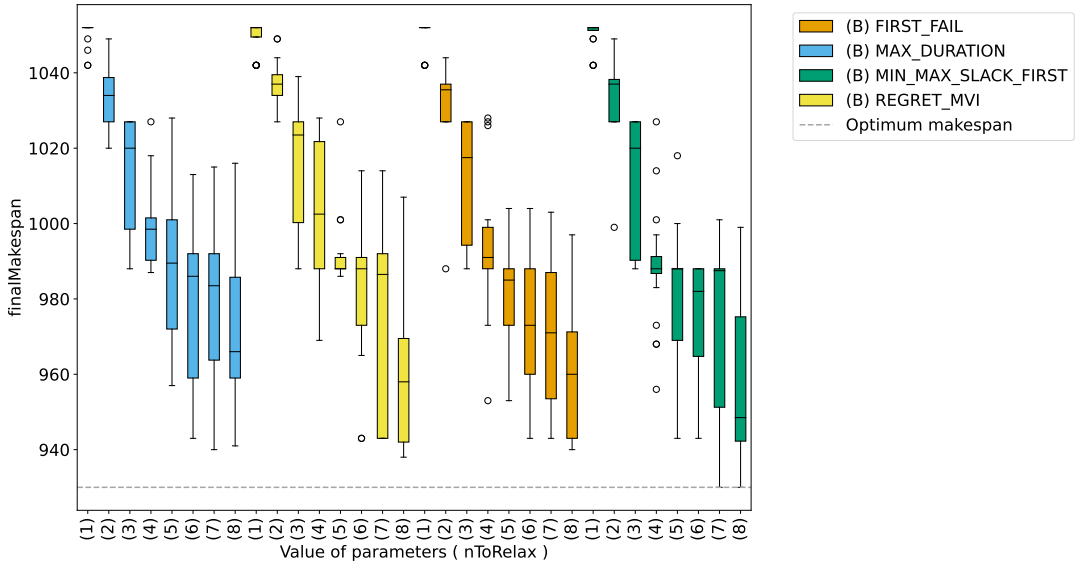



Figure 7.23: Comparison of the final makespan of each branching according to the parameters for the ***relaxation machine*** for the ft10 instance

### *Relation order on job and order on machine*

For the two next following relaxations, *order on job* and *order on machine* relaxations, we will do a unique analyze because the behavior is very similar.

**Final makespan** If we analyze the graph of the final makespan, we can observe in this graph 7.25 that, for the *order on job relaxation*, it is the *min-max-slack first* branching with nToRelax equal to 7 that is the best. And for the *order on machine relaxation* 7.27, it is the *min-max-slack first* branching with nToRelax equal to 6 that is the best. The range of the most likely value of makespan is the same for the two relaxations is between approximately 955 and 938. And we can also observe, in the two cases, the optimum has been reached at least one times.

**Execution time** For the execution time, we can see from both graphs 7.24 and 7.26 that reaching the final makespan takes approximately twenty seconds, which is a good execution time to have a final makespan close to the optimum.

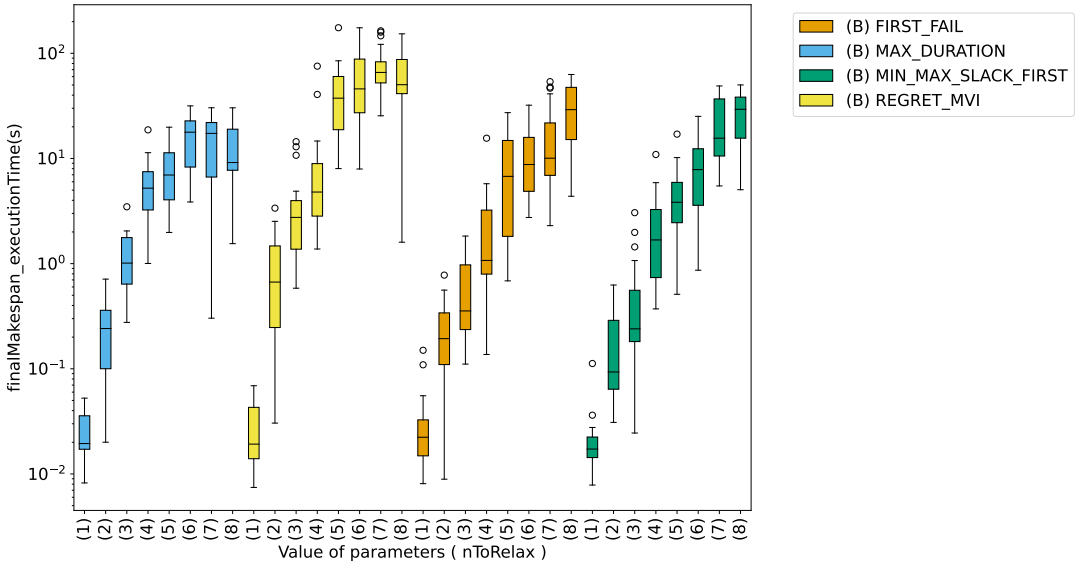



Figure 7.24: Comparison of the execution time of each branching according to the parameters for the ***order on job relaxation*** for the ft10 instance

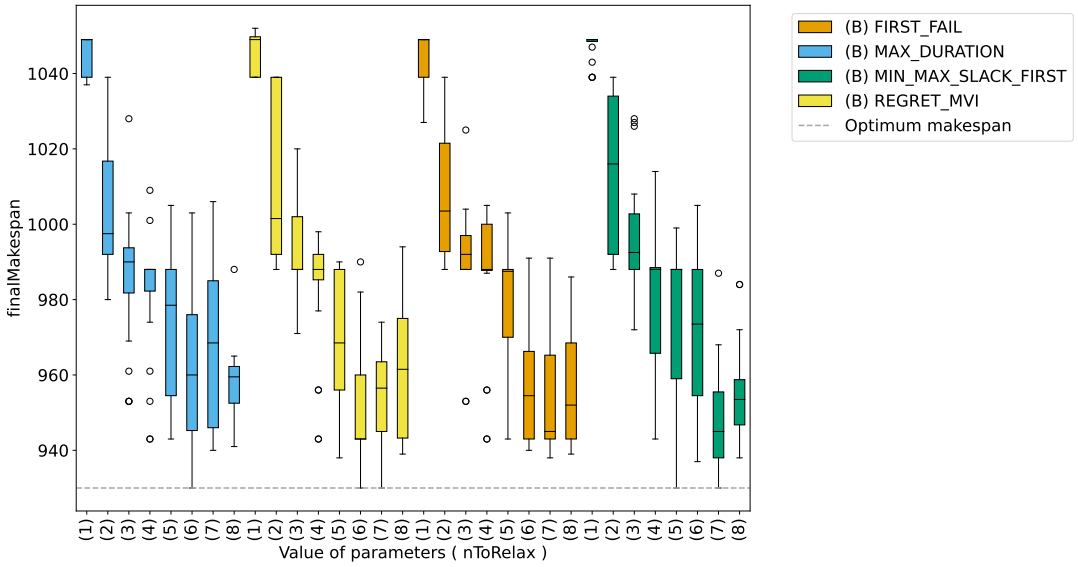



Figure 7.25: Comparison of the final makespan of each branching according to the parameters for the ***order on job relaxation*** for the ft10 instance

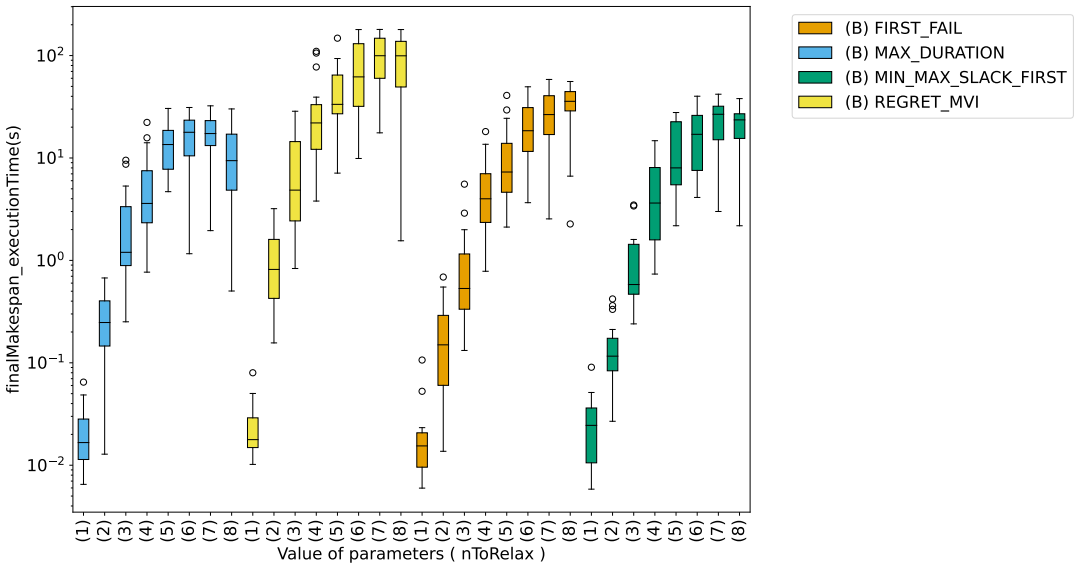



Figure 7.26: Comparison of the execution time of each branching according to the parameters for the ***order on machine relaxation*** for the ft10 instance

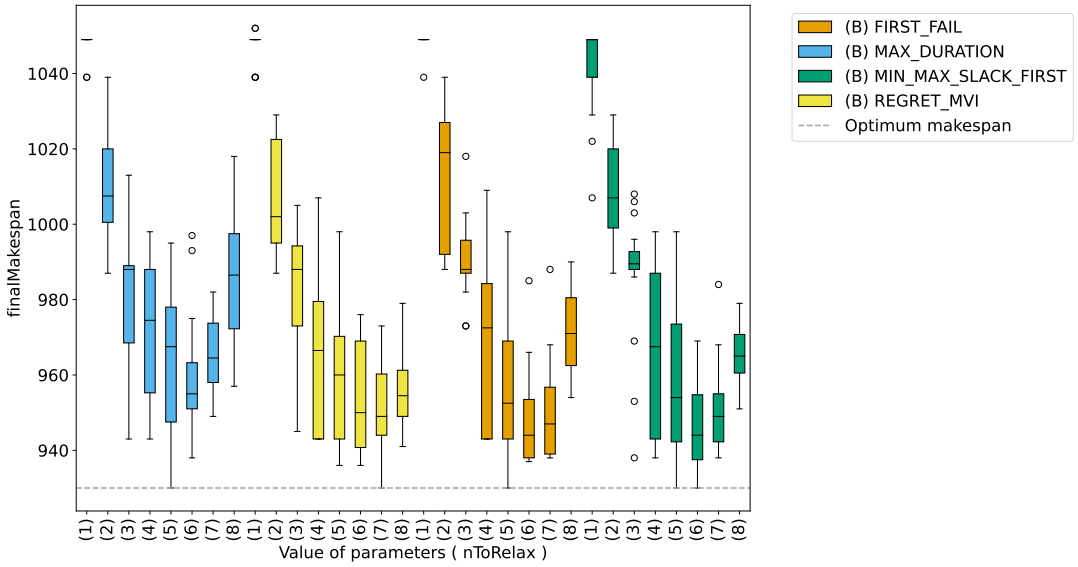



Figure 7.27: Comparison of the final makespan of each branching according to the parameters for the ***order on machine relaxation*** for the ft10 instance

### *Shaw relaxation*

In order to analyze the *Shaw relaxation*, we need to consider multiple parameter values. To facilitate the analysis, we have created separate graphs for each branching strategy. After selecting the top five parameter values for each branching strategy, we will compare them to determine the best branching strategy with the optimal parameters.

***First-fail* branching** For the *first-fail* branching strategy, if we observe the graph in figure 7.28, the top five parameter values with the best median makespan are: $(2, 50)$, $(3, 70)$, $(4, 70)$, $(5, 70)$, $(6, 50)$. The most promising parameter values are $(4, 70)$ and $(5, 70)$, with a makespan range around 940, a slightly smaller median, and the optimum makespan being achieved.

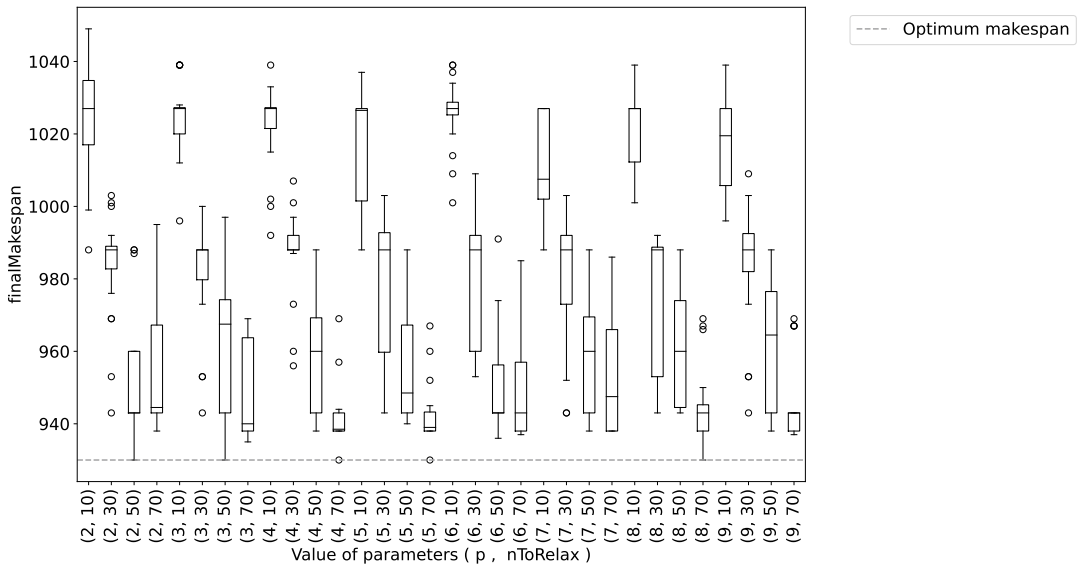



Figure 7.28: Comparison of the final makespan of ***first-fail*** **branching** according to the parameters for the ***Shaw relaxation*** for the ft10 instance

***Maximum duration*** **branching** For the *maximum duration* branching strategy, if we observe the graph 7.29, the top five parameter values with the best median makespan are: $(3, 70)$, $(4, 50)$, $(5, 50)$, $(7, 50)$, $(7, 70)$.

The parameter values $(3, 70)$ and $(7, 70)$ appear to be more reliable in terms of the makespan final range being very narrow. Although their median may not be the smallest, they provide more assurance about the final result compared to parameter values such as $(4, 50)$. Therefore, if obtaining a consistent and predictable makespan is a priority, choosing the parameter values $(3, 70)$ or $(7, 70)$ would be a sensible decision.

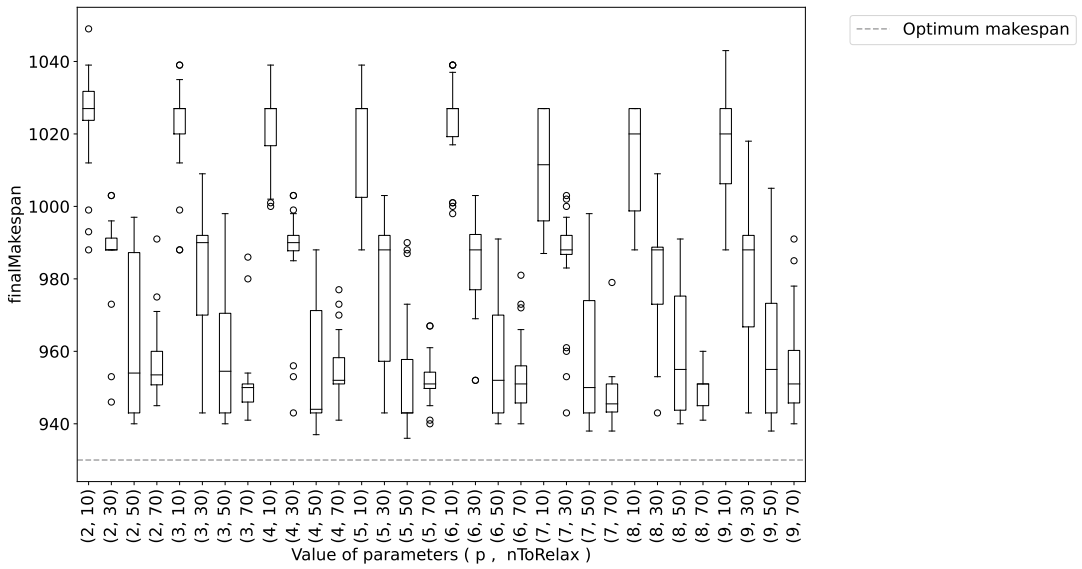



Figure 7.29: Comparison of the final makespan of ***maximum duration*** **branching** according to the parameters for the ***Shaw relaxation*** for the ft10 instance

***min-max-slack first*** **branching** For the *min-max-slack first* branching, if we observe the graph 7.30, the top five parameter values with the best median makespan are: $(2, 70)$, $(4, 70)$, $(6, 70)$, $(7, 70)$, $(8, 70)$.

All these values of parameters seems promising. The optimum makespan is reached and the range of value between the first and third quartile (represented by the box) is tight.

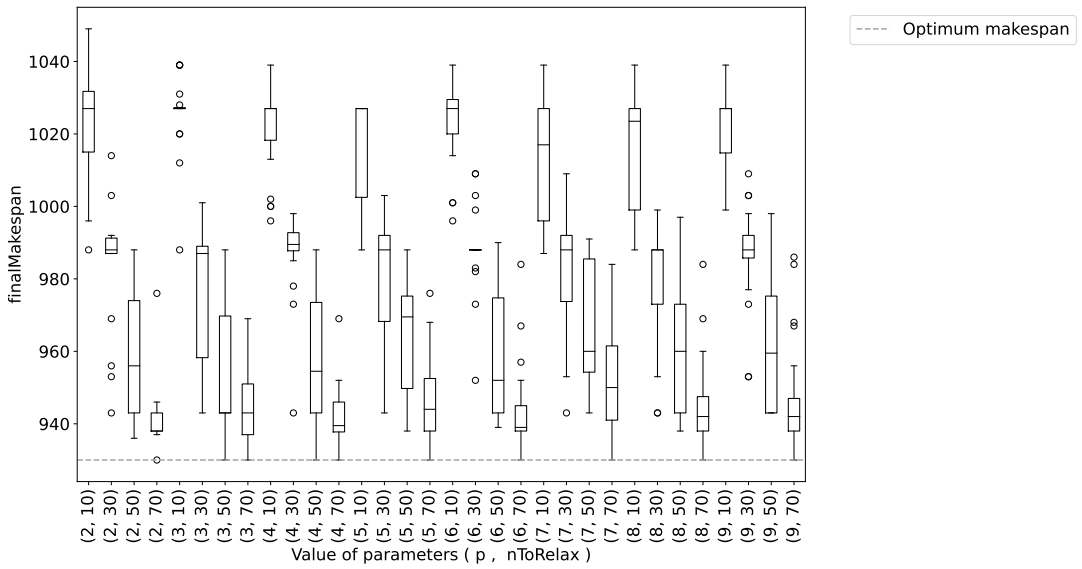



Figure 7.30: Comparison of the final makespan of ***min-max-slack first*** **branching** according to the parameters for the ***Shaw relaxation*** for the ft10 instance

***Regret MVI*** **branching** For the *regret MVI* branching, if we observe the graph 7.31, the top five parameter values with the best median makespan are: $(2, 50)$, $(2, 70)$, $(4, 70)$, $(5, 70)$, $(7, 70)$.

Indeed, the parameter value $(2, 50)$ seems to have a wider range of makespan final values compared to the other parameter values. While the median may be relatively low, the makespan final can vary between 960 and 945, indicating a higher degree of uncertainty in the final result. It might be preferable to prioritize parameter values with a narrower range to achieve more consistent and reliable results.

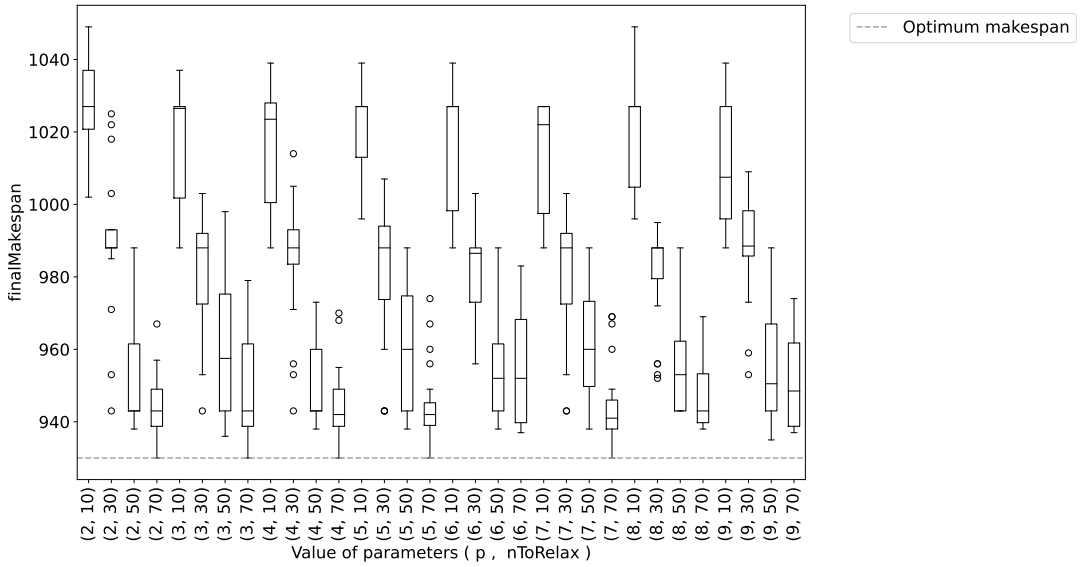



Figure 7.31: Comparison of the final makespan of ***Regret MVI*** **branching** according to the parameters for the ***Shaw relaxation*** for the ft10 instance

**Final makespan** From the analysis of the makespan final values in the graph 7.32, it can be observed that the *min-max-slack first* branching appears to be the most promising. It consistently achieves lower makespan final values across different parameter combinations. However, it's worth noting that the *Regret MVI* branching also shows promising results, except for the parameter value $(2, 50)$, which has a wider range of makespan final values.

Considering the overall performance and consistency, it seems that the *min-max-slack first* branching is a preferable choice for obtaining better makespan final values in the *Shaw* relaxation.

**Execution time** To choose the best values of parameter, we will also compare the execution time in this graph 7.33. Indeed, the parameter combination $(6, 70)$ for the *min-max-slack first* branching seems to offer a good trade-off between execution time and makespan final value. With an execution time of around ten seconds, it has a high probability of achieving a makespan final value ranging from 945 to 938, and there is even a possibility of reaching the optimum.

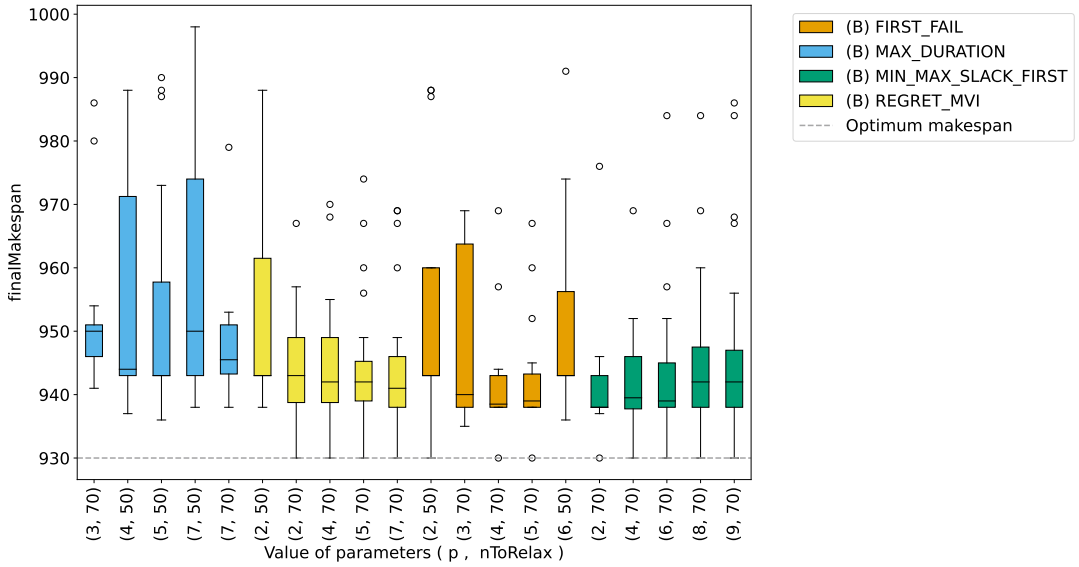



Figure 7.32: Comparison of the final makespan of each branching according to the parameters for the ***Shaw relaxation*** for the ft10 instance

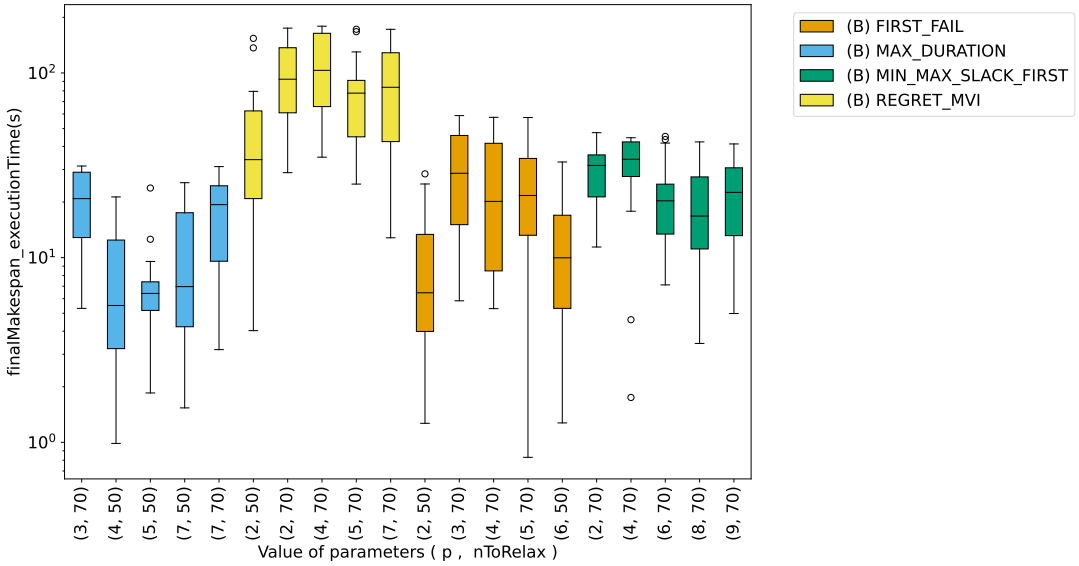



Figure 7.33: Comparison of the execution time of each branching according to the parameters for the ***Shaw relaxation*** for the ft10 instance

### *Relaxation alternate*

To conclude these analyses, we will analyze the last relaxation, namely *alternate* relaxation, which involves alternating between the previously analyzed relaxations.

To do so, we tested two *alternate* relaxations for the two most promising branchings identified in the previous analyses, namely the *regret MVI* and *min-max-slack first* branchings. For each of these branchings, we considered the top three performing relaxations and the three relaxations that yielded poorer results.

We compare these *alternate* relaxations with each individual relaxation that makes up the alternated relaxations.

**Final makespan** One of the most interesting results is that *alternate* relaxation with the least good relaxations, i.e. *order on job*, *order on machine* and *random*, produces good results for *min-max-slack first* branching. As can be seen from this graph 7.34, the final makespan will have a better value with this alternate relaxation.

The *alternate* relaxation with the best relaxation produces also good results, although *job* relaxation is better when used on its own.

**Execution time** Regarding the execution time, there is no significant change observed in the graph 7.35 for this branching strategy.

For the other branching, the *regret MVI* branching, we can see in this graph 7.36 that *alternate* relaxations do not give better results and that it's better to use relaxations on their own than alternately.

For the execution time 7.37, we cannot guarantee that it will be necessarily shorter, and therefore, the LNS approach may not always be faster.

In conclusion, the *alternate* relaxation is a good option for the *min-max-slack first* branching, even if the execution can be slower.

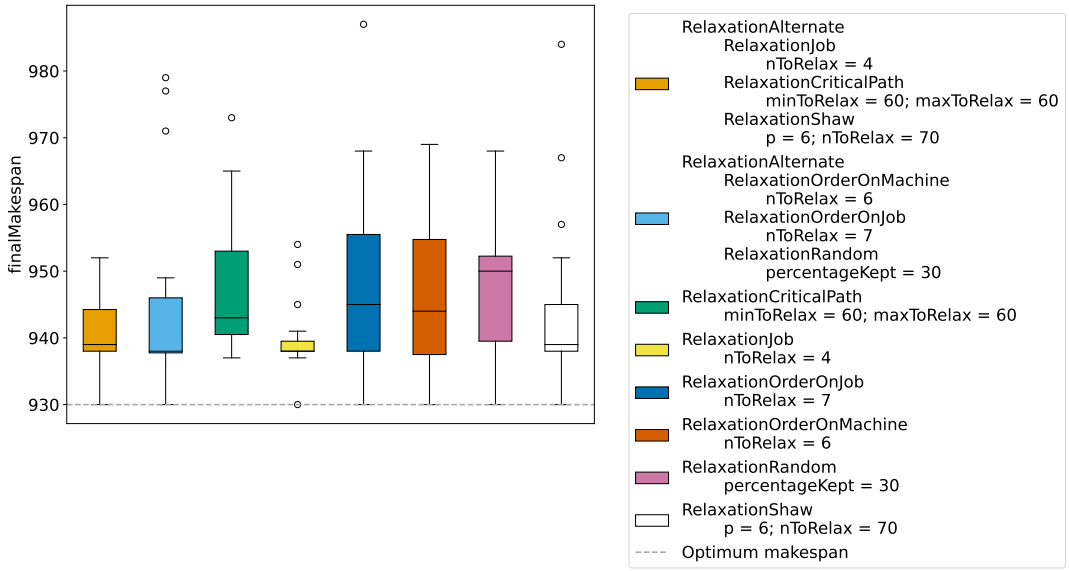



Figure 7.34: Comparison of the final makespan for ***min-max-slack first*** **branching** according to multiple relaxations for the ft10 instance

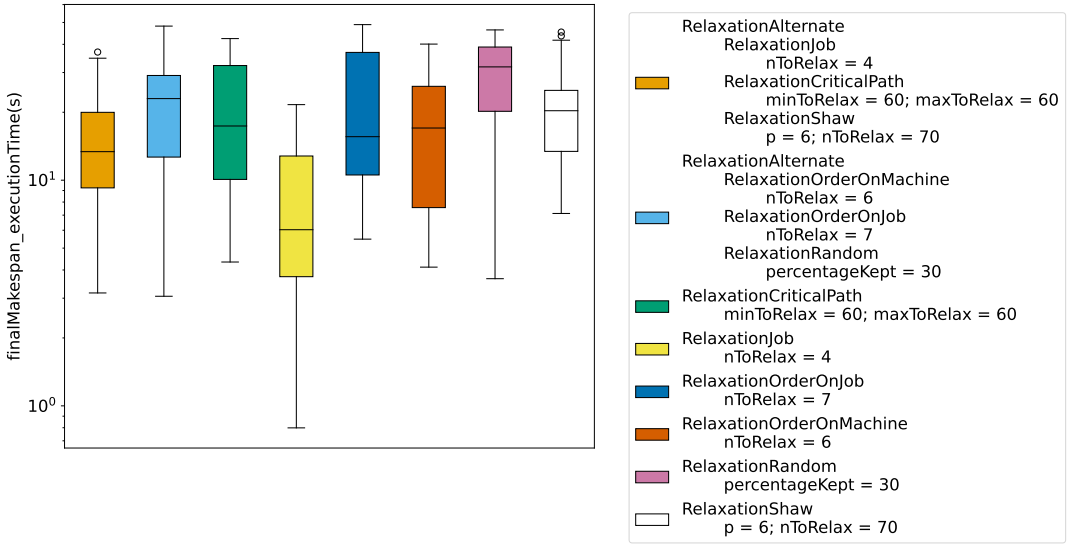



Figure 7.35: Comparison of the execution time for ***min-max-slack first*** **branching** according to multiple relaxations for the ft10 instance

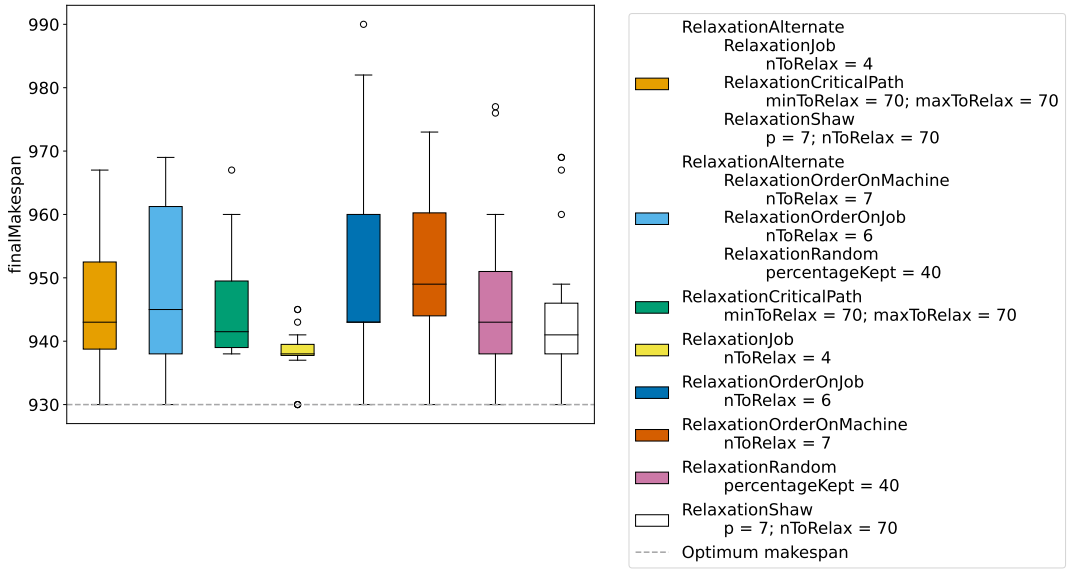



Figure 7.36: Comparison of the final makespan for ***regret MVI*** **branching** according to multiple relaxations for the ft10 instance

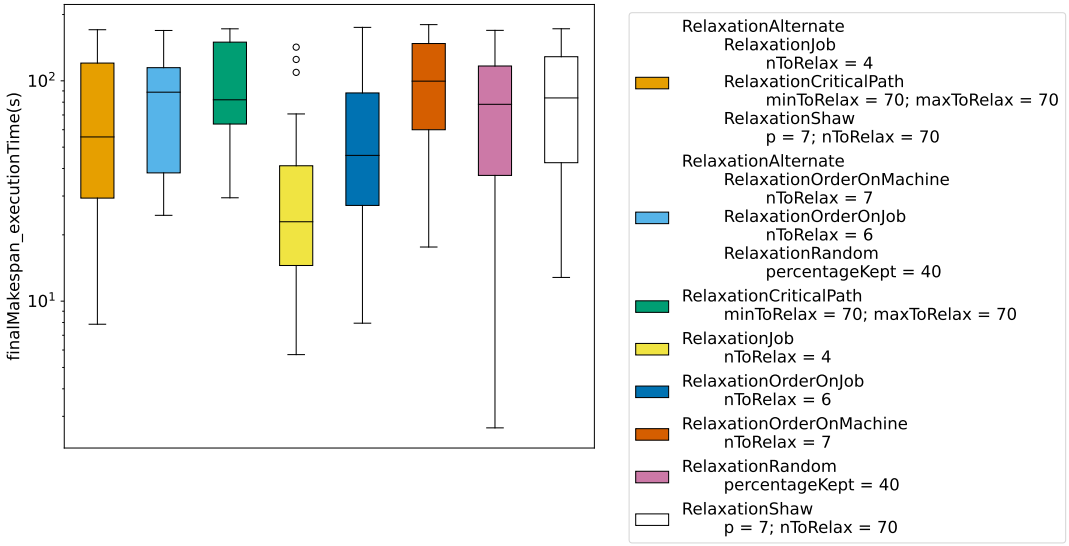



Figure 7.37: Comparison of the execution time for ***regret MVI*** **branching** according to multiple relaxations for the ft10 instance

# Chapter 8

# Conclusion

The primary objective of this research was to implement the Job-Shop problem using Constraint Programming and the sequence variable and analyze the effectiveness of this approach in achieving good results.

The implementation objective has been successfully accomplished by modeling the problem and implementing various search techniques. Of course, due to time constraints, we did not had the time to explore some ideas.

The secondary objective was to evaluate whether the proposed modeling approach, utilizing heuristics and the sequence variable, could outperform conventional modeling approaches for the problem.

The first objective has been achieved, demonstrating the successful implementation of the Job-Shop problem using Constraint Programming and the sequence variable. However, upon comparing our models with existing ones, it is evident that the utilization of the sequence variable, as currently implemented, does not yield superior results when employing Depth-First Search (DFS).

Nevertheless, it is worth noting that several relaxation heuristics for Large Neighborhood Search (LNS) have shown promising results by achieving the optimum solution within a reasonable time frame of approximately ten seconds. It is important to mention that our relaxation heuristic for the sequence variable was not directly compared to other models of the literature using the LNS framework.

In conclusion, while the implementation objective has been accomplished, the utilization of the sequence variable in the modeling of the Job-Shop problem did not lead to improved results when employing DFS. However, promising results were obtained through the relaxation heuristic for LNS, warranting further exploration and comparison with alternative relaxation models for different branching strategies.

## 8.1 Future works

In this section, we present some propositions for improvement and for further exploration.

### 8.1.1 Channeling constraint

A good improvement would be to avoid posting the lessOrEqual constraint and instead filter the minimum and maximum bounds of the time window. This filtering avoids posting constraints and therefore StateInt variables on the StateManager.

### 8.1.2 Shaw relaxation criterion

For the *Shaw relaxation*, one of the most important things is the definition of the criterion of similarity between two nodes. One avenue for exploration would be to test different definitions for this criterion and determine which one yields the best results.

### 8.1.3 Relaxation alternate

For the time being, for this relaxation, we change heuristics as soon as we need to carry out a new relaxation. Another possibility would be to change relaxation only after a certain amount of time has passed without the solution having been improved or according to a percentage

### 8.1.4 Single insertion branching procedure

All our branching procedure introduced in section 5.2 all rely on the same principle of producing a branch for each insertion on the sequence variable for the selected node to be inserted. While this technique is efficient in the case of our model of the Job-Shop with sequence variables, it could however cause issues with models which that rely on indirect insertions. Indeed, when a node is selected to be inserted, in all the branches following the branching point, the node is considered as inserted even if the node could be inserted thanks to an indirect insertion. This is not the case for our model since no node will only be able to get inserted in part of the sequence through an indirect insertion.

An idea to solve this issue, would be to use branching procedures that only perform one insertion per branching point, the other branch would be to disallow the insertion. To disallow an insertion, it is not sufficient to remove the member insertion, since the node could still be inserted between the two member nodes with an indirect insertion, leading to numerous equal solution. We therefore need

to remove all insertions of the inserted node between the two member nodes where the insertion was made. To do this, we could for instance develop new constraints to perform filtration on the sequence. We could also adapt the sequence variable to register the information of the nodes that are not allowed to be inserted between specific member nodes.

Single insertion branching procedure could thus be used for other problems than the Job-Shop or to explore new branching techniques such as *Conflict Ordering Search* [9].

# Index

activity, 1
AllDifferent constraint, 14
assignment, 9

backtrack, 23
branch and bound (BnB), 24
branching points, 18
branching procedure, 18

candidate solution, 11
channeling constraint, 35
consistency
    bound consistency, 14
    domain consistency, 13
constraint, 9
constraint optimization problem (COP), 10
constraint propagator, 12
constraint satisfaction problem (CSP), 10
critical path, 6
critical task, 6

decision (branching), 18
disjunctive constraint, 17
diversification, 25
domain, 9
duration, 1

earliest start, 58
end, 3
end variable, 34
execution-window, 3

filtering algorithm, 12
filteringInsertions, 37
fix (variable), 9
full-slack, 46

inconsistency (CSP), 11
indirect insertions, 28
insertion, 28
intensification, 25

job, 1
Job-Shop, 1

Large Neighborhood Search (LNS), 25
left-slack, 46
lessOrEqual constraint, 16
LNS, 25

machine, 1
makespan, 1
maximal-insertion-slack, 48
member insertion, 28
model (of Job-Shop using sequence variables), 33
MVI (makespan variation on insertion), 52

optimization(LNS), 26

posted, 8
postponed set search, 59
precedence, 5

propagate, 12
prune (branch), 24
prune (domains), 22

regret, 52
relaxation
    alternate, 66
    critical path, 62
    job, 61
    machine, 61
    order on job, 62
    order on machine, 62
    random, 60
    shaw, 64
relaxation heuristic, 25
relaxation(LNS), 26

schedule early, 60
search heuristic, 19
search space, 18
search tree, 18
sequence variable, 27
SetTimes, 58
solution set CSP, 11
start, 3
start variable, 34
start-window, 3
support
    domain, 12
    interval, 13

task, 1

variable, 9

wipe-out (domain), 9

# References


[1] J Christopher Beck, Patrick Prosser, and Richard J Wallace. "Trying again to fail-first". In: *Recent Advances in Constraints: Joint ERCIM/CoLogNet International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2004, Lausanne, Switzerland, June 23-25, 2004, Revised Selected and Invited Papers*. Springer. 2005, pp. 41–55.

[2] Craig Boutilier et al. "Constraint-based optimization with the minimax decision criterion". In: *Principles and Practice of Constraint Programming–CP 2003: 9th International Conference, CP 2003, Kinsale, Ireland, September 29–October 3, 2003. Proceedings 9*. Springer. 2003, pp. 168–182.

[3] Peter Brucker, Yu N Sotskov, and Frank Werner. "Complexity of shop-scheduling problems with fixed number of jobs: a survey". In: *Mathematical Methods of Operations Research* 65 (2007), pp. 461–481.

[4] Emilie Danna and Laurent Perron. "Structured vs. Unstructured Large Neighborhood Search: A Case Study on Job-Shop Scheduling Problems with Earliness and Tardiness Costs". In: *Principles and Practice of Constraint Programming – CP 2003*. Ed. by Francesca Rossi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 817–821. ISBN: 978-3-540-45193-8.

[5] Augustin Delecluse, Pierre Schaus, and Pascal Van Hentenryck. "Sequence Variables for Routing Problems". In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.

[6] M. Dell'Amico and M. Trubian. "Applying tabu search to the job-shop scheduling problem". In: *Annals of Operations Research* 41 (1993), 231–252. URL: https://doi.org/10.1007/BF02023076.

[7] Henry Fisher. "Probabilistic learning combinations of local job-shop scheduling rules". In: *Industrial scheduling* (1963), pp. 225–251.

[8] M. R. Garey, D. S. Johnson, and Ravi Sethi. “The Complexity of Flowshop and Jobshop Scheduling”. In: *Mathematics of Operations Research* 1.2 (1976), pp. 117–129. ISSN: 0364765X, 15265471. URL: http://www.jstor.org/stable/3689278 (visited on 05/16/2023).

[9] Steven Gay et al. “Conflict ordering search for scheduling problems”. In: *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*. Springer. 2015, pp. 140–148.

[10] Daniel Godard, Philippe Laborie, and Wim Nuijten. “Randomized Large Neighborhood Search for Cumulative Scheduling.” In: Jan. 2005, pp. 81–89.

[11] Joaquim A.S. Gromicho et al. “Solving the job-shop scheduling problem optimally by dynamic programming”. In: *Computers & Operations Research* 39.12 (2012), pp. 2968–2977. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2012.02.024. URL: https://www.sciencedirect.com/science/article/pii/S0305054812000500.

[12] Karla L Hoffman, Manfred Padberg, Giovanni Rinaldi, et al. “Traveling salesman problem”. In: *Encyclopedia of operations research and management science* 1 (2013), pp. 1573–1578.

[13] A.S. Jain and S. Meeran. “Deterministic job-shop scheduling: Past, present and future”. In: *European Journal of Operational Research* 113.2 (1999), pp. 390–434. ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(98)00113-1. URL: https://www.sciencedirect.com/science/article/pii/S0377221798001131.

[14] James E. Kelley and Morgan R. Walker. “Critical-Path Planning and Scheduling”. In: *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM ’59 (Eastern). Boston, Massachusetts: Association for Computing Machinery, 1959, 160–173. ISBN: 9781450378680. DOI: 10.1145/1460299.1460318. URL: https://doi.org/10.1145/1460299.1460318.

[15] Philip Kilby and Andrew Verden. “Flexible routing combing constraint programming, large neighbourhood search, and feature-based insertion”. In: *Proceedings 2nd Workshop on Artificial Intelligence and Logistics (AILOG’11)*. 2011, pp. 43–49.

[16] S. Lawrence. “Resource Constrained Project Scheduling. An Experimental Investigation of Heuristic Scheduling Techniques (Supplement)”. In: (1984).

[17] Alejandro López-Ortiz et al. “A fast and simple algorithm for bounds consistency of the alldifferent constraint”. In: *IJCAI*. Vol. 3. 2003, pp. 245–250.

[18] L. Michel, P. Schaus, and P. Van Hentenryck. "MiniCP: a lightweight solver for constraint programming". In: *Mathematical Programming Computation* 13.1 (2021), pp. 133–184. DOI: 10.1007/s12532-020-00190-7. URL: https://doi.org/10.1007/s12532-020-00190-7.

[19] *MiniCP Sequences - Anonymous GitHub*. 2021. URL: https://anonymous.4open.science/r/minicp-sequences-5EE3/README.md.

[20] V Nageshwara Rao and Vipin Kumar. "Parallel depth first search. part i. implementation". In: *International Journal of Parallel Programming* 16.6 (1987), pp. 479–499.

[21] Yves Rochat and Éric D Taillard. "Probabilistic diversification and intensification in local search for vehicle routing". In: *Journal of heuristics* 1 (1995), pp. 147–167.

[22] Vianney le Clément de Saint-Marcq et al. "Sparse-sets for domain implementation". In: *In CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*. 2013, 1–10.

[23] Paul Shaw. "A new local search algorithm providing high quality solutions to vehicle routing problems". In: 1997.

[24] Paul Shaw. "Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems". In: *Principles and Practice of Constraint Programming — CP98*. Ed. by Michael Maher and Jean-Francois Puget. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 417–431. ISBN: 978-3-540-49481-2.

[25] Stephen F Smith and Cheng-Chung Cheng. "Slack-based heuristics for constraint satisfaction scheduling". In: *AAAI*. 1993, pp. 139–144.

[26] Charles Thomas. "Advanced modelling and search techniques for routing and scheduling problems". PhD thesis. 2023.

[27] Charles Thomas, Roger Kameugne, and Pierre Schaus. "Insertion sequence variables for hybrid routing and scheduling problems". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21–24, 2020, Proceedings 17*. Springer. 2020, pp. 457–474.

**UNIVERSITÉ CATHOLIQUE DE LOUVAIN**
**École polytechnique de Louvain**
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**