UCLouvain

epl

École polytechnique de Louvain

Optimization of production planning with resource allocation

Author: **Florian KNOP**
Supervisors: **Pierre SCHAUS, Charles THOMAS**
Reader: **Vincent BRANDERS**
Academic year 2018–2019
Master [120] in Computer Science

Abstract

This thesis presents an optimization problem with resource allocation. We need to assign workers and several resources to jobs which may require various skills while taking into account multiple constraints such as availability. We present two models to solve this problem: Constraint Programming and Mixed Integer Programming. Comparing their performances shows that Constraint Programming outperforms Mixed Integer Programming in most cases. However, combining the two models offers the best results as we can take advantage of Constraint Programming to find a good feasible solution quickly and Mixed Integer Programming to further optimize this solution.

Acknowledgements

First, I would like to thank Charles Thomas for his guidance and advice throughout the year and for his useful explanations on various subjects such as performance profiles. I also thank him for reading this thesis in its first drafts and helping me improve it.

I express my gratitude to Pierre Schaus for his valuable remarks on multi-objective optimization using Constraint Programming.

Finally, I thank Stéphanie Bogaert at N-Side for taking the time to talk about the Village n°1 problem.

Florian

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Staff scheduling is an important problem encountered by many organizations, such as industry or hospitals. One of the most studied staff scheduling problem is the nurse scheduling problem (NSP). Staff scheduling problems such as the NSP become hard to solve as the problem grows in size due to their NP-hard complexity.

This thesis presents an optimization problem for production planning with resource allocation based on the needs of a Belgian company called Village n°1. Village n°1 employs people with disabilities. They offer services to companies and private individuals such as industrial jobs. They have to assign their workforce to jobs, as well as various resources. However, manually assigning resources can be time consuming and error-prone. The goal is to automate the resource allocation process with little to no manual intervention.

Due to the important nature of staff scheduling, a lot of techniques exist to solve these problems. The aim of this thesis is to present two approaches that solve the problem described in Chapter 2. These techniques are Constraint Programming and Mixed Integer Programming. We then analyze, compare, and discuss the performances of these techniques.

This thesis is organized as follows:

Chapter 2 introduces the resource allocation problem derived from the needs of Village n°1.

Chapter 3 describes the state-of-the-art in the domains of Mixed Integer Programming and Constraint Programming.

Chapter 4 gives formal definitions of the Mixed Integer Programming and the Constraint Programming models.

Chapter 5 describes the implementation of the models presented in Chapter 4.

Chapter 6 presents the carried experiments, their results, and our analysis of these results.

Chapter 7 concludes this thesis by giving some takeaway and suggests some improvements that could be made to the solvers.

Chapter 2

The resource allocation problem

This chapter presents the resource allocation problem. We first introduce the general problem and its constraints, the formal models are then described in Chapter 4.

The resource allocation problem described in this thesis is a staff scheduling problem based on the needs of the Village n°1 company. This company employs people with disabilities, they offer services to companies and private individuals such as industrial jobs. Working with disabled people means that there are special needs concerning the work that each worker can perform.

Our resource allocation problem consists of:

- A planning period (T) with each period ($t \in T$) equal in duration.
- A list of clients ($c \in C$).
- A list of demands ($d \in D$).
- A list of workers ($w \in W$).
- A list of skills ($s \in S$).
- A list of locations ($l \in L$).
- A list of machines ($m \in M$).
- A list of incompatibilities between workers ($\langle w_1, w_2 \rangle \in I_w \mid w_1, w_2 \in W$). Two incompatible workers cannot work with each other.
- A list of incompatibilities between workers and clients ($\langle w, c \rangle \in I_{wc}$). A worker incompatible with a client cannot work for that client.

Each demand has:

- A client ($c_d \in C$).
- A given set of time periods ($T_d \subseteq T$).
- A required number of workers per period ($n_d \in \mathbb{N}$). This number translates into n_d positions in the demand. $P(d)$ describes the set of positions from 0 through $n_d - 1$.
- Some skill requirements to be fulfilled by different workers ($S_d \subseteq S$). It imposes that some workers have the required capacities to work at a given position (e.g. package lifter). Each position $p \in P(d)$ requires the skill $s_{d,p} \in S_d$. Each skill in the set S_d may represent multiple real-life skills. For example, if position p needs to have someone who is a *lifter* and a *supervisor* at the same time, $s_{d,p} = lifter \wedge supervisor$.
- Additional skill requirements to be fulfilled by any workers assigned to that demand (e.g. driver license) ($S_d^+ \subseteq S$).
- A list of machines to perform the work ($M_d \subseteq M$). Vehicles used to drive the workers to a work location are considered as machines.
- An eventual list of possible locations where the demand can be performed ($L_d \subseteq L$).

Each worker has:

- A list of skills (e.g. package lifter, supervisor, etc.) ($S_w \subseteq S$). A worker w can only work for a demand d at position $p \in P(d)$ if $s_{d,p} \in S_w$.
- A list of availabilities at which the worker can work ($T_w \subseteq T$).
- A list of working requirements ($r \in R$). A requirement r states that a worker r_w needs to work for a minimum of r_{min} and maximum of r_{max} times in the problem planning period.

This type of problem can be seen as a variant of the well known nurse scheduling problem (`NSP`) [2]. These problems often contain hard constraints to state restrictions and soft constraints to state preferences and to minimize violations.

Our goal is to assign workers to multi-skill positions as well as machines and locations to a list of demands over the planning period. Each resource (i.e. worker, machine, location) must satisfy all the constraints described in Section 2.1.

2.1 Constraints

To describe the problem constraints, we define three types of assignments:

- A_w: set of worker assignments. We represent each assignment with a 4-tuple $\langle t, d, p, w \rangle$ where the worker w is assigned to position p of demand d at time t.
- A_m: set of machine assignments. We represent each assignment with a 2-tuple $\langle d, m \rangle$ where the machine m is assigned to the demand d for the entire duration of that demand.
- A_l: set of location assignments. We represent each assignment with a 2-tuple $\langle d, l \rangle$ where the location l is assigned to the demand d for the entire duration of that demand.

2.1.1 Hard constraints

A worker can only work when available

Each worker has a defined set of availabilities and cannot be assigned to a demand when unavailable.

$$t \in T_w, \ \forall \ \langle t, d, p, w \rangle \in A_w$$

No worker should be assigned to a demand which is not occurring

A demand has a set of time periods in which it occurs, no workers should be assigned to that demand if the demand is not occurring.

$$t \in T_d, \ \forall \ \langle t, d, p, w \rangle \in A_w$$

No worker can be assigned twice for the same period

A worker cannot do the work of two different workers at the same time. Hence, a worker can only work at most once per time period. For every pair of worker assignments, there cannot be a worker assigned twice for the same time period.

$$w_1 \neq w_2, \ \forall \ \langle t, d_1, p_1, w_1 \rangle \in A_w, \langle t, d_2, p_2, w_2 \rangle \in A_w$$

Each demand has a required number of workers

Each demand needs a number of workers to be satisfied. For each time period in which a demand is occurring, it should have the required number of workers n_d assigned to it.

$$|\{\langle t, d, p, w\rangle \mid \langle t, d, p, w\rangle \in A_w\}| = n_d,\ \forall\ d \in D, t \in T_d$$

Each assignment must respect skill restrictions

Each position of a demand may require skills to be satisfied. To be assigned to that position, a worker must have the required skills. A worker can also have more skills than the required skills by the position.

$$s_{d,p} \in S_w,\ \forall\ \langle t, d, p, w\rangle \in A_w$$

Worker-worker incompatibilities

Workers might be incompatible with each other. Such workers cannot be assigned together (i.e. in the same demand) at the same time period.

$$d_1 \neq d_2,\ \forall\ \langle w_1, w_2\rangle \in I_w, \langle t, d_1, p_1, w_1\rangle \in A_w, \langle t, d_2, p_2, w_2\rangle \in A_w$$

Worker-client incompatibilities

A worker and a client might be incompatible with each other. If this is the case, the worker must not be assigned at a demand for such client.

$$c_d \neq c,\ \forall\ \langle w, c\rangle \in I_{wc}, \langle t, d, p, w\rangle \in A_w$$

The required machines must always be assigned

A demand has machine needs. Such machines should always be assigned for a demand to be satisfied.

$$\exists\ \langle d, m\rangle \in A_m,\ \forall\ d \in D, m \in M_d$$

No machines should be assigned twice for the same period

A machine is assigned for the entirety of a demand. Moreover, it cannot be assigned twice for the same period. In other words, it cannot be assigned for two overlapping demands in time. We define the set of overlapping demands with $O(d) = \{d_o \mid T_{d_o} \cap T_d \neq \varnothing, \forall\ d_o \in D, d_o \neq d\}$

$$m_1 \neq m_2,\ \forall\ d \in D, d_o \in O(d), \langle d, m_1 \rangle \in A_m, \langle d_o, m_2 \rangle \in A_m$$

The location assigned must be in the set of possible locations

A demand has a set of possible locations. Only one of those locations can be assigned to that demand.

$$l \in L_d,\ \forall\ \langle d, l \rangle \in A_l$$

No location should be assigned twice for the same period

As with machines, locations must be assigned only once per time period.

$$l_1 \neq l_2,\ \forall\ d \in D, d_o \in O(d), \langle d, l_1 \rangle \in A_l, \langle d_o, l_2 \rangle \in A_l$$

2.1.2 Soft constraints

Satisfy the most assignments possible

A demand might not have enough workers or have impossible constraints. We need to be able to have a final result in which the worker assignments are all fulfilled. For this to happen, we can assign a dummy worker to positions and minimize the number of times this dummy worker is assigned. This dummy worker can work at every time period, has every skills, and can work as many times as possible even in the same time period.

Contiguous shifts

A demand has multiple positions and these positions should keep the same worker over time to avoid the hassle for the workers of changing positions. In practice, this is an unreal requirement as not every worker will be available for the entire duration of the demand. We instead minimize the number of different workers over time for each position.

Working requirements

Workers can have minimum and maximum working requirements. We want to make sure that these requirements are satisfied. However, as this is not always possible to solve, we state this as a soft constraint and minimize the number of times a requirement is not met.

Chapter 3

State of the art

3.1 Nurse scheduling problem

The nurse scheduling problem (NSP) is a well-known combinatorial problem. It involves assigning nursing staff to shifts. It takes into account both hard and soft constraints. The objective maximizes the preferences of the nursing staff while minimizing the violations of the soft constraints. The problem is known to be NP-hard [3]. The NSP can be transformed to many types of staff scheduling problems. Due to this, the literature contains a lot of different methods to solve it.

Our problem described in Chapter 2 is a staff scheduling problem with resource allocation. It can be seen as a variant of the nurse scheduling problem. For example, assigning nurses to shifts can be transformed into assigning workers to demand positions and assigning patients to rooms can be seen as assigning locations to demands.

3.2 Mixed Integer Programming

Linear Programming (LP) is a mathematical optimization technique which is used to minimize or maximize an objective subject to constraints represented by linear equations. If all variables are required to be integers, it is called Integer Programming (IP). Integer Programming, in contrast to LP which can be solved efficiently, is often NP-complete. Mixed Integer Programming (MIP) takes LP and IP together to form a problem where only some variables are required to be integers. MIP problems are also generally NP-complete.

The most common MIP problems are of the form:

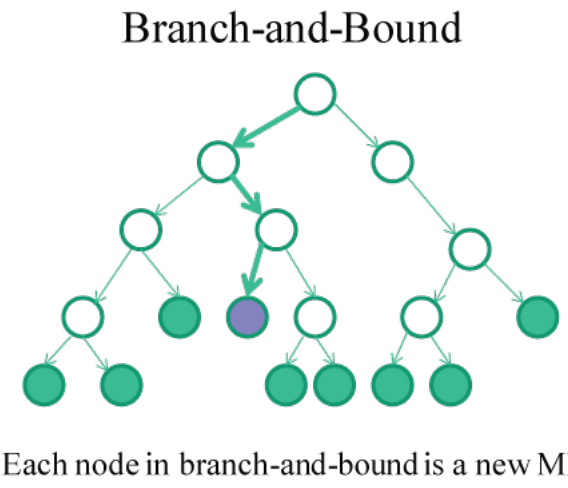

Figure 3.1: `MIP` Branch and Bound search tree [1]

$$\begin{aligned} \min \quad & \boldsymbol{c}^T\boldsymbol{x} && (3.1)\\ \text{s.t} \quad & A\boldsymbol{x} = \boldsymbol{b} && (3.2)\\ & \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u} && (3.3)\\ & \text{Some or all } x_i \text{ must take integer values} && (3.4) \end{aligned}$$

(3.1) is the problem objective. $\boldsymbol{c}^T$ is the vector of coefficients, $\boldsymbol{x}$ is the vector of variables. (3.2) are the linear constraints. $\boldsymbol{b}$ is a vector of bounds while A is a matrix of coefficients for the constraints. (3.3) are the bound constraints. Each x_i can only take values between l_i and u_i. Finally, (3.4) states the integrality constraints over some or all variables.

`MIP` problems are usually solved using a branch-and-bound algorithm [1]. The process is as follows: we start with the `MIP` formulation and remove all integrality constraints to create a resulting linear programming relaxation to the original problem. The relaxation can be solved easily compared to the original problem. The result might satisfy all integrality constraints and be a solution to the original problem. But more often than not, a variable has a fractional value. We can then solve two relaxations by imposing two additional constraints. For example, if x takes value 5.5, we add the following linear constraints: $x \leq 5.0$ and $x \geq 6.0$. This process is repeated throughout the search tree (Figure 3.1) until a valid solution is found. More techniques are used to find solutions more efficiently. Each solver (e.g Gurobi Optimizer [1]) uses its own algorithm.

3.2.1 Gurobi Optimizer

The *Gurobi Optimizer* [4] is a state-of-the-art commercial solver for mathematical programming. Gurobi includes multiple solvers, among those: (i) Linear Programming (`LP`); (ii) Mixed Integer Linear Programming (`MILP`), abbreviated as `MIP`.

The Gurobi Optimizer is used by more than 2100 companies in over 40 industries at this time. It allows describing business problems as mathematical models. It also supports a lot of programming interfaces in a variety of programming languages like `C++`, `Java`, `Python`, and `C#`.

3.3 Constraint Programming

Constraint Programming is a technique used for solving hard combinatorial problems. It is a programming paradigm where relations between variables are stated as constraints to create a Constraint Satisfaction Problem.

A *Constraint Satisfaction Problem* (`CSP`) consists of a set of n variable, $\{x_1, \ldots, x_n\}$; a domain $D(x_i)$ of possible values for each variable x_i, $1 \leq i \leq n$; and a collection of m constraints $\{C_1, \ldots, C_m\}$. Each constraint C_j, $1 \leq j \leq m$, is a constraint over some set of variables called the scheme of the constraint. The size of this set is known as the arity of the constraint. The number of constraints associated with a variable is called the degree of the variable. A solution to a `CSP` is an assignment of values $a_i \in D_i$ to x_i, that satisfies all the constraints. [5]

Problems are sometimes over-constrained and hard to solve with a `CSP`. In those cases, we can transform our `CSP` to a Constraint Optimization Problem (`COP`) where we try to optimize one or multiple objectives coming from the transformation of hard constraints into soft constraints.

3.3.1 Global constraints

We now describe the principle of global constraints and how they are useful for our resource allocation problem described in Chapter 2.

As described in more depth in [6]:

> [...] a constraint C is often called "global" when "processing" C as a whole gives better results than "processing" any conjunction of constraints that is "semantically equivalent" to C.

The author also defines three types of constraint globality, we are mostly interested in what he refers to *operational globality*. Those constraints can be decomposed

into multiple simpler constraints but the filtering quality of the decomposition is often worse than its global counterpart.

There also exists soft variants [7] of global constraints where a constraint is associated with a number of violations. This is particularly useful for over-constrained problems which cannot be solved by a `CSP`. Instead, the `CSP` is transformed into a `COP` where the number of violations is minimized.

Alldifferent constraint

In our resource allocation problem (Chapter 2), we need to assign different workers to demands during the same time period. This is usually done in Constraint Programming by using the `alldifferent` constraint.

The `alldifferent` constraint [8] is one of the most famous global constraint used in Constraint Programming. This constraint is defined over a subset of variables for which values must be different. More formally:

$$\texttt{alldifferent}(x_1, \ldots, x_n) = \{(d_1, \ldots, d_n) \mid d_i \in D(x_i), d_i \neq d_j \forall i \neq j\}$$

This constraint can be decomposed into multiple binary inequalities. This makes `alldifferent` an operational global constraint. It can be proven that the filtering of the global constraint cannot be achieved with a decomposition. As an example, let us define three variables x_1, x_2 and x_3 respectively taking domains $\{1, 2\}$, $\{1, 2\}$, $\{1, 2, 3, 4\}$. The global constraint would be able to successfully filter 1 and 2 from the domain of x_3 because the values are always taken by x_1 and x_2. However, the decomposition is not able to filter those values.

Global cardinality constraint

As described in Chapter 2, our resource allocation problem needs to take into account working requirements, i.e. a minimum and a maximum number of times that a worker can work. For this, we need to count the occurrences of a worker being assigned to a position and limit those occurrences to a minimum and maximum.

The global cardinality constraint (`gcc`) [9] is a generalization of the `alldifferent` constraint. It does not enforce, although it can, the uniqueness of values of its variables but enforces that the cardinality of each value d_i for all its variables in its scope lies between a lower bound and an upper bound, respectively l_i and u_i.

$$\texttt{gcc}(X, l, u) = \{(d_1, \ldots, d_n) \mid d_i \in D(x_i), l_d \leq |\{d_i \mid d_i = d\}| \leq u_d, \forall d \in D(X)\}$$

As stated above, we can express the `alldifferent` constraint with this definition:

$$\texttt{gcc}(\{x_1, \ldots, x_n\}, [1, \ldots, 1], [1, \ldots, 1]) = \texttt{alldifferent}(x_1, \ldots, x_n)$$

We are also interested in a soft variant of `gcc` called `softgcc` [10]. The violation associated with this constraint is the sum of excess or shortage [11] for each value.

$$\texttt{softgcc}(X, l, u, Z) = \{(d_1, \ldots, d_n) \mid d_i \in D(x_i), d_z \in D(Z), viol(d_1, \ldots, d_n) \leq d_z\}$$
$$\text{with} \quad viol(d_1, \ldots, d_n) = \sum_{d \in D(X)} \max(0, |\{d_i \mid d_i = d\}| - u_d, l_d - |\{d_i \mid d_i = d\}|)$$

3.3.2 Large Neighborhood Search

A Constraint Programming solver can often get stuck in a search tree that does not lead to good solutions. We want instead to explore as much of the search space as possible.

Large Neighborhood Search (`LNS`) is a technique that makes use of the principles of local search. `LNS` uses Constraint Programming as a tool to find solutions and local search to expand the exploration of the search space. The `LNS` framework often goes as follows:

1. Use Constraint Programming to find a solution.
2. Relax last best solution: we fix some variables to the last value in the best solutions. This is the step that can change the most for different types of problems. Most of the time, a simple random relaxation is used (i.e. randomly fix a percentage of variables).
3. Restart.

The entire search can be limited with a time limit, number of solutions or number of restarts. Each independent search is often limited with a number of backtracks or a time limit.

3.3.3 Variable Objective Search

A multi-objective problem is often modeled by having a weighted sum of sub-objectives to form a single objective.

$$\begin{aligned} \min \quad & obj = \sum_i w_i o_i \\ \text{s.t.} \quad & constraints \end{aligned}$$

Our resource allocation problem (Chapter 2) uses such an objective. One issue with this objective modeling is how to prioritize sub-objectives (e.g. is it more important to minimize contiguous workers or requirements?). This is usually solved by assigning more weight to more important objectives. However, the pruning of such method alone is inefficient.

Variable Objective Large Neighborhood Search (`VO-LNS`) [12] is an extension of `LNS` for multi-objective problems which offers (i) prioritization of sub-objectives; (ii) better pruning. Each objective in the `VO-LNS` framework has three types of filtering available:

1. *No-Filtering*: The objective has no impact.
2. *Weak-Filtering*: When a solution is found, it has to be better or equal to the bound of the objective.
3. *Strong-Filtering*: When a solution is found, it has to be strictly improving the bound of the objective.

The `VO-LNS` formulation is expressed as follows:

$$\begin{aligned} \min \quad & obj = (obj_1, \ldots, obj_n, obj_{n+1}) \\ \text{s.t.} \quad & constraints \end{aligned}$$

$obj_1, \ldots, obj_n$ are the sub-objectives while obj_{n+1} is the original weighted sum. We keep obj_{n+1} in *Strong-Filtering* during the search such that the formulation is at least as strong as a sum of sub-objectives. We can change the filtering dynamically during the search before each restart depending on the problem and prioritization of sub-objectives.

3.3.4 Heuristics

The backtracking algorithm uses two heuristics for its search. One heuristic chooses the variable while the other chooses the value for the previously selected variable. Good heuristics can drive the search quickly to good solutions. We present in Section 4.1 a value heuristic created for the needs of the problem.

One of the biggest principle used for variable ordering is the first-fail principle. This principle states that the search should first select the variable that most likely leads to an inconsistency. Multiple heuristics follow this principle, the most simple being the smallest domain ordering.

3.3.5 OscaR

OscaR [13] is a Scala toolkit for solving Operations Research problems. OscaR has multiple optimization techniques available: (i) Constraint Programming; (ii) Constraint Based Local Search; (iii) Derivative Free Optimization; (iv) Visualization.

The project is mainly developed by UCLouvain and the research group of Pierre Schaus. But some companies like *N-Side* and *CETIC* collaborate on its development.

The library of OscaR that this project uses is the Constraint Programming library. It offers a lot of existing constraints and abstractions. Some black-box searches are also implemented with the possibility to implement custom heuristics to drive the search forward.

Chapter 4

Models for the resource allocation problem

In this chapter, we present two models that attempt to solve the problem described in Chapter 2. We first present a Mixed Integer Programming model followed by a Constraint Programming model. We compare and discuss both model performances in Chapter 6. The complete notations used by these models can be found in Appendix A.

4.1 Mixed Integer Programming model

We first start by presenting the mathematical model, we describe the variables needed to model our problem and the constraints associated to them.

4.1.1 Variables

To represent our problem in `MIP`, we need three types of variables, one per resource and an additional type of variable for the dummy workers.

$$w_{ijkl} = \begin{cases} 1 & \text{if worker } i \text{ is working at time } j \text{ for demand } k \text{ at position } l \\ 0 & \text{otherwise} \end{cases}$$

$$d_{jkl} = \begin{cases} 1 & \text{if no worker is assigned at time } j \text{ for demand } k \text{ at position } l \text{ (dummy)} \\ 0 & \text{otherwise} \end{cases}$$

$$m_{ij} = \begin{cases} 1 & \text{if machine } i \text{ is used for demand } j \\ 0 & \text{otherwise} \end{cases}$$

$$l_{ij} = \begin{cases} 1 & \text{if location } i \text{ is used for demand } j \\ 0 & \text{otherwise} \end{cases}$$

This is, in fact, a binary integer programming model as every variable is a $\{0, 1\}$ integer.

As solutions can be partial, we need to introduce a way to allow the absence of worker for a given position. In `MIP`, we model this by having the variables d_{jkl}, d for *dummy*. This variable is one, if and only if all the corresponding worker variables $(w_{ijkl}, \forall_i)$ are equal to zero. The goal is to minimize the number of dummy variables assigned to one.

4.1.2 Constraints

All positions must be assigned with one worker

All positions must have one worker assigned to it. We achieve this by summing all the worker variables for each position. We also add the dummy variable associated to this position to allow partial solutions.

$$\sum_{i \in W} w_{ijkl} + d_{jkl} = 1, \qquad \forall k \in D, j \in T_{d_k}, l \in P(d_k) \tag{4.1}$$

Figure 4.1 shows a visualization of constraints (4.1) and (4.2) for one time period. However, to simplify, we suppose that demands only need one worker, thus ignoring positions.

All workers assigned in a time period must be different

One worker can only work at one position per time period. For each worker, we add the sum of all its variables over each time period. This sum must be less than or equal to one to make sure a worker works at most once in the period.

$$\sum_{k \in D} \sum_{l \in P(d_k)} w_{ijkl} \leq 1, \qquad \forall i \in W, j \in T \tag{4.2}$$

Exclude impossible values

Some resources (workers, locations, demands) cannot be assigned in some time periods. We need to set the variables to 0 if this is the case.

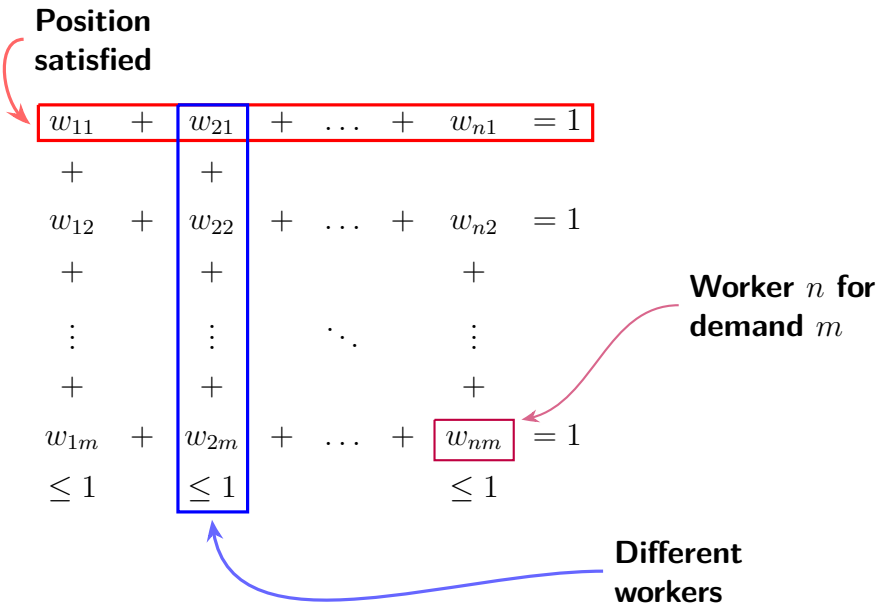

Figure 4.1: Visualization of (4.1) and (4.2).

$$t_j \notin T_{d_k} \implies \forall i, l \; w_{ijkl} = 0, \qquad \forall j \in T, k \in D \tag{4.3}$$
$$t_j \notin T_{w_i} \implies \forall k, l \; w_{ijkl} = 0, \qquad \forall j \in T, i \in W \tag{4.4}$$
$$t_j \notin T_{d_k} \implies \forall l \; d_{jkl} = 0, \qquad \forall j \in T, k \in D \tag{4.5}$$
$$l_i \notin L_{d_j} \implies l_{ij} = 0, \qquad \forall i \in L, j \in D \tag{4.6}$$
$$m_i \notin M_{d_j} \implies m_{ij} = 0, \qquad \forall i \in M, j \in D \tag{4.7}$$

Incompatibilities between workers and clients

For each incompatibility $\langle i, c \rangle \in I_{wc}$, we set every worker variables of worker i, if the client of the demand is c, to zero.

$$c_{d_k} = c \implies \forall l \; w_{ijkl} = 0, \qquad \forall \langle i, c \rangle \in I_{wc}, j \in T, k \in D \tag{4.8}$$

Incompatibilities between workers

For each incompatibility and each demand, two incompatible workers cannot have both their variables assigned to one. We define that the sum over all the positions of the two incompatible workers must be lower than two.

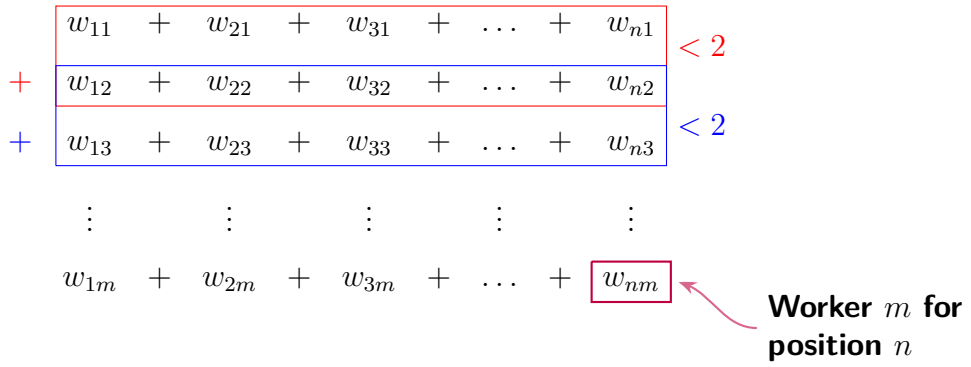

Figure 4.2: Example of (4.9) if (w_1, w_2) and (w_2, w_3) are incompatible pairs.

$$\sum_{l \in P(d_k)} w_{ajkl} + w_{bjkl} < 2, \qquad \forall \langle a, b \rangle \in I_w, j \in T, k \in D \tag{4.9}$$

Restrict skilled positions to skilled workers

These constraints ensure that no worker is working for a position at which they are not qualified to work. $W_{s_{d_k,l}}$ defines the set of workers that satisfy the skill of demand k at position l.

$$w_{ijkl} = 0, \qquad \forall j \in T, k \in D, l \in P(d_k), i \in W \setminus W_{s_{d_k,l}} \tag{4.10}$$

Additional skills must be satisfied

An additional skill must be satisfied by at least one worker in a team. For each additional skill, we sum over all the workers that satisfy the skill to check if at least one is working in the team.

$$\sum_{i \in W_s} \sum_{l \in P(d_k)} w_{ijkl} \geq 1, \qquad \forall j \in T, k \in D, s \in S^+_{d_k} \tag{4.11}$$

Location assignments should be satisfied

For each demand that requires a location, we sum over all the location variables for this demand and check if only one is assigned.

$$|L_{d_j}| > 0 \implies \sum_{i \in L} l_{ij} = 1, \qquad \forall j \in D \tag{4.12}$$

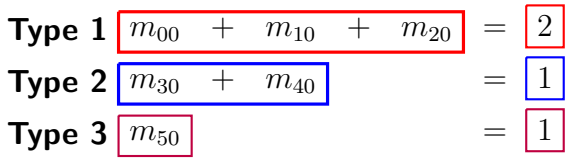

Figure 4.3: Visualization example for (4.14).

No location should be assigned to overlapping demands

For each demand and its overlapping demands, we check that a location is not assigned to both demands. We do this with a sum lower than or equal to one.

$$l_{ij} + l_{ik} \leq 1, \qquad \forall j \in D, k \in O(d_j), i \in L \tag{4.13}$$

Machine assignments should be satisfied

For each demand, we check that each required machine is assigned to that demand. In the set of machines M, we can have multiple machines equal to the same value v. We define $M_v \subseteq M$ the set of all machines in M equal to v.

$$\sum_{i \in M_v} m_{ij} = |\{m \mid m \in M_{d_j}, m = v\}|, \qquad j \in D, v \in M_{d_j} \tag{4.14}$$

As an example, let us take a demand d_0 with $M_{d_0} = \{1, 1, 2, 3\}$. This demand needs two machines of type 1, one machine of type 2 and one machine of type 3. Let us now imagine that we have in total three machines of type 1, two machines of type 2 and one machine of type 3. We have six variables associated with this demand: $m_{00}, \ldots, m_{50}$ with m_{00} through m_{20} the variables associated with the machines of type 1, m_{30} and m_{40} the variables associated with the machines of type 2 and m_{50} the variable associated with the machine of type 3. Figure 4.3 shows a visualization of this example.

No machine should be assigned to overlapping demands

Following the same principle of (4.13), we check that two machines are not assigned to two overlapping demands.

$$m_{ij} + m_{ik} \leq 1, \qquad \forall j \in D, k \in O(d_j), i \in M \tag{4.15}$$

Binary constraints

These constraints simply state that each variable must be a binary $\{0, 1\}$ variable.

$$w_{ijkl} \in \{0,1\}, \qquad \forall i \in W, j \in T, k \in D, l \in P(d_k) \tag{4.16}$$
$$d_{jkl} \in \{0,1\}, \qquad \forall j \in T, k \in D, l \in P(d_k) \tag{4.17}$$
$$m_{ij} \in \{0,1\}, \qquad \forall i \in M, j \in D \tag{4.18}$$
$$l_{ij} \in \{0,1\}, \qquad \forall i \in L, j \in D \tag{4.19}$$

4.1.3 Objective

$$\min \quad \delta_0 \sum_{k \in D} \sum_{l \in P(d_k)} \sum_{i \in W} min(\sum_{j \in T} w_{ijkl}, 1) \tag{4.20a}$$
$$+ \delta_1 \sum_{r \in R} (max(r_{min} - occ_{r_w}, 0) + max(r_{max} - occ_{r_w}, 0)) \tag{4.20b}$$
$$+ \delta_2 \sum_{j \in T} \sum_{k \in D} \sum_{l \in P(d_k)} d_{jkl} \tag{4.20c}$$
$$\text{with} \quad occ_i = \sum_{j \in T} \sum_{k \in D} \sum_{l \in P(d_k)} w_{ijkl}, \quad \forall i \in W$$
$$\boldsymbol{\delta} = \langle \delta_0, \delta_1, \delta_2 \rangle$$

The objective function is stated in (4.20), it is a weighted sum split into multiple parts, it minimizes:

1. The number of different workers for every position between periods of that demand (4.20a), $min(\sum_{j \in T} w_{ijkl}, 1)$ is one if the worker i is working for that position at any time, 0 otherwise. Hence, the sum of those values for all workers is equal to the number of workers for that shift.
2. The number of violations of working requirements (4.20b).
3. The number of dummy workers assigned to demands (4.20c).

occ_i represents the occurrences of worker i while the penalties $\boldsymbol{\delta}$ are representative of the importance of each sub-objective.

4.2 Constraint Programming model

We now present our Constraint Programming (`CP`) model. This model contains some differences with the mathematical model described in Section 4.1. For example, a

CP model rarely contains binary variables to refer to multiple values of a domain. Instead, it uses integer variables having the entire domain. Typically, binary variables $w_0, \ldots, w_i, \ldots, w_n$ where $i \in W$ and $w = i$ if $w_i = 1$ are equivalent to one variable $w \in \{0, \ldots, n\}$.

4.2.1 Variables

First, we need to express the set of workers for each demand at each time period at which that demand occurs.

$$w_{ijk} \in W \tag{4.21}$$

(4.21) is the worker working at time i for demand j at the kth position with $t_i \in T$, $d_i \in D$, $t_i \in T_{d_j}$ and $k \in P(d_j)$. The same reasoning is used for locations and machines:

$$m_{ij} \in M \tag{4.22}$$
$$l_i \in L \tag{4.23}$$

(4.22) is the jth machine used for demand i while (4.23) is the location used for demand i.

As explained in the problem description and in the mathematical model section, we need to allow partial solutions in which we have a dummy worker that can work at any time. We add this value to every worker variable domain but ignore it during the constraint propagation. We define this worker by $\sigma \notin W$. The actual value of this worker does not matter as long as it does not belong to W. For simplicity, we define $\sigma = -1$.

This modeling already satisfies some constraints, such as the number of required resources (i.e. worker, location, machine) per demand. We also satisfy the required skills and availabilities for each position by only initializing variables with the possible workers. Let $W_{s_{d_j,k}} \subseteq W$ be the subset of workers that satisfy the kth skill of demand d_j.

$$w_{ijk} \in W_{s_{d_j,k}} \cap \{w \mid t_i \in T_w\} \cup \{\sigma\}, \forall j \in D, i \in T_{d_j}, k \in P(d_j) \tag{4.24}$$

Note that initializing the variables with a reduced set of values is semantically equivalent to adding a `not_equal` constraint for each impossible value.

We also follow the same principle for the locations and machines by reducing the domain at initialization instead of adding constraints. For each required machine in a demand, we take all possible machines of this type in M.

$$m_{ij} \in \{m \mid m \in M, m = M_{d_i,j}\}, \forall i \in D, j \in 1..|M_{d_i}| \tag{4.25}$$

For each location of a demand, the domain is the possible locations of that demand.

$$l_i \in L_{d_i}, \forall i \in D \tag{4.26}$$

4.2.2 Constraints

All workers for one period must be different

All the worker variables for a given time period must be different. The `alldifferent` (Section 3.3.1) constraint is well suited to express this. However, as our model has the dummy worker σ in the domain of all worker variables and this value can appear as many times as possible, we need a slight variant of the `alldifferent` called `alldifferent_except`. This constraint has the same behavior as the original except that we can specify values that are ignored from the constraint propagation.

$$\begin{aligned}\texttt{alldifferent\_except}(X, v) = \{(d_1, \ldots, d_n) \mid d_i \in D(x_i),\\ d_i \notin v \ \wedge \ d_j \ \notin v \implies d_i \neq d_j \ \forall i \neq j\}\end{aligned}$$

We use this constraint to ignore the σ value from the propagation. For each period, we define:

$$\texttt{alldifferent\_except}(\{w_{ijk} \mid j \in D, k \in P(d_j)\}, \{\sigma\}), \forall i \in T \tag{4.27}$$

Incompatibilities between workers and clients

A worker might have an incompatibility with a client or a set of clients. Clients are characteristics of the demands, we can solve this constraint by adding a series of `not_equal` constraints for each incompatible worker-client pair.

$$\texttt{not\_equal}(w_{ijk}, w), \forall \langle w, c\rangle \in I_{wc}, i \in T, j \in \{d \mid d \in D \wedge c_d = c\}, k \in P(d_j) \tag{4.28}$$

Incompatibilities between workers

A worker might have an incompatibility with another worker or a set of workers. Unlike the worker-client incompatibilities, we cannot solve this constraint with a series of `not_equal` constraints. We instead use a constraint called `negative_table`.

This constraint is a type of *Table Constraints* [14] which in general can express either the allowed or forbidden combinations of values. In this case, `negative_table` expresses the forbidden combinations of values. These combinations are expressed by the table I_w. We add a `negative_table` constraint for each pair of workers for a demand at one given time. Let $P_{ij} = \{(w_{ijk}, w_{ijl}) \mid k, l \in P(d_j), k \neq l\}$ be the permutations of worker variables for demand j at period i:

$$\texttt{negative\_table}(x, y, I_w), \forall (x, y) \in P_{ij} \tag{4.29}$$

In practice, we can slightly reduce the number of constraints by checking first if each pair of workers are indeed available at these positions.

Additional skills must be satisfied

A demand can have what we call additional skills. Those skills can be satisfied by any workers in the demand. Unlike required skills by different workers, we cannot pre-assign possible values to the domains of variables. Any number of workers in the demand can have an additional skill. We use the `gcc` constraint coupled with a `sum` constraint. The `gcc` acts as a counter of occurrences for the workers that satisfy the skills, the sum states that at least one worker needs to be assigned.

Let us define o_{ijs} the occurrences of workers ($\in W_s$) at time i for demand j.

$$\begin{aligned} & \texttt{gcc}(\{w_{ijk} \mid k \in P(d_j)\}, o_{ijs}) && (4.30)\\ & \texttt{sum}(o_{ijs}) \geq 1 && (4.31)\\ \text{with} \quad & o_{ijs} \in \{0, 1\} && (4.32)\\ & \forall j \in D, s \in S^+_{d_j}, i \in T_{d_j} && (4.33) \end{aligned}$$

This is a different syntax for `gcc` from what we introduced before. This variant takes variables and assigns the occurrences of values to them. In this case, the `gcc` assigns occurrences of $w \in W_s$ to o_{ijs} and the `sum` constraint ensures that these occurrences sum to at least one.

Machines can only be assigned once per period

As machines can only be assigned once per period and are assigned for the whole duration of the demand. We need to ensure that a machine is not assigned to two overlapping demands in time. For each pair of overlapping demands, we add a `alldifferent` constraint with all the machine variables associated to the two demands.

$$\texttt{alldifferent}(\{m_{ij} \mid j \in M_{d_i}\} \cup \{m_{kj} \mid j \in M_{d_k}\}), \forall i \in D, k \in O(d_i) \tag{4.34}$$

Locations can only be assigned once per period

As with machines, locations can only be assigned once per period and are assigned for the whole duration of the demand. We need to ensure that a location is not assigned to two overlapping demands in time. For each pair of overlapping demands, we add a `not_equal` constraint with the two associated location variables.

$$\texttt{not\_equal}(l_i, l_k), \forall i \in D, k \in O(d_i) \tag{4.35}$$

Minimizing violations of working requirements

Workers might have working requirements. They have to work for a minimum (maximum) number of times, hence the total occurrences of these workers must be above (below) or equal the requirement. As a solution cannot always be achieved with these requirements, we use a soft constraint and minimize the number of violations. In this case, we use the `softgcc` constraint introduced in Section 3.3.1. Let X be the entire set of variables and v_r the total number of violations.

$$\texttt{softgcc}(X, [r_{1_{min}}, \ldots, r_{n_{min}}], [r_{1_{max}}, \ldots, r_{n_{max}}], v_r) \tag{4.36}$$

Note that from a model point of view, if a worker does not have any requirement, r_{min} is 0 and r_{max} is $|T_{r_w}|$ (i.e. the number of availabilities of that worker).

Minimizing the number of dummy workers

A solution might not always be possible, leading to a partial solution containing dummy workers. We defined this dummy worker by the value σ. This is again a case of soft constraint where we use a `softgcc`. Let v_σ be the total number of violations and X be, again, the entire set of variables.

$$\texttt{softgcc}(X, \sigma \rightarrow \sigma, [0], [0], v_\sigma) \tag{4.37}$$

This syntax is a little bit different than what was introduced before. We specify $\sigma \rightarrow \sigma$ to check only the occurrences of values in that range, hence only σ in our case.

Objective function

We already defined violations v_r (4.36) and v_σ (4.37) as our working requirements and dummy worker violations respectively. We also need to define a final part of our objective function which is not a violation per se. Let N_{jk} be the number of different workers working for demand j at position k throughout the periods T_{d_j}. We use a constraint called `at_least_nvalue` [15] to count this number. Let

$W_{jk} = \{w_{ijk} \mid i \in T_{d_j}\}$ be the set of worker variables for demand j at position k across all time periods for that demand:

$$\texttt{at\_least\_nvalue}(W_{jk}, N_{jk}) \ \forall j \in D, k \in P(d_j) \tag{4.38}$$

We now have the number of different workers for each shift and we need to minimize the sum of all N_{jk} to avoid perturbations (change of worker).

The final objective is a weighted-sum of all sub-objectives in the model. However, not all objectives are equal in priority, some objectives need bigger penalties when violated. This is the case for v_r and v_σ. We define three penalties δ_0, δ_1 and δ_2 which are associated with our three sub-objectives.

$$\min \quad \delta_0(\sum_{j \in D} \sum_{k \in P(d_j)} N_{jk}) + \delta_1 v_r + \delta_2 v_\sigma \tag{4.39}$$

$$\boldsymbol{\delta} = \langle \delta_0, \delta_1, \delta_2 \rangle \tag{4.40}$$

4.2.3 Search

We now define the variable and value heuristics used in our model. The variable heuristic is a variant of the max degree heuristic while the value heuristic is a custom heuristic that selects the most available worker for a position.

Variable heuristic

The variable heuristic used for the search is a custom max degree heuristic. The degree of a variable is the number of constraints assigned to it. However, as skills are not stated with constraints, we also add the number of required skills for a position k to the existing degree of the variable. As an example, if a variable X has a degree of 3 and the position defined by this variable requires 2 different skills, the final degree will be 5. We also make sure that if a variable is of size two, it is always selected first to avoid having the dummy value σ assigned to it by propagation.

Most available value heuristic

We define a value heuristic that we call the *most available heuristic*. This heuristic consists of two value orderings.

1. The first ordering orders the workers from most available to least available throughout the duration of the demand. This allows the search to select workers that are more likely to work for that demand throughout all periods.

2. If workers have the same number of availabilities for a demand, they are ordered with respect to their remaining availabilities in other demands. This second ordering is important for smaller demands as the search chooses workers that are less likely to be needed in other demands.

The heuristic also never considers the dummy worker σ for most available worker. This value heuristic will in practice find solutions much quicker than a traditional *min* value heuristic (see Subsection 6.2.1).

Let us take an example to show how this heuristic works in practice, let us define w_1, w_2 and w_3, three possible workers for two demands d_1 and d_2 that only need one worker each. The availabilities are defined as $T_{w_1} = \{0, 1, 2\}$, $T_{w_2} = \{0, 2\}$, $T_{w_3} = \{0, 1, 2, 3, 4\}$ and the demand occurrences as $T_{d_1} = \{0, 1, 2\}$, $T_{d_2} = \{0, 1, 2, 3, 4\}$. Intuitively, we can see that worker w_3 should be assigned to d_2 and w_1 should be assigned to d_1. This is what the heuristic tries to achieve, the ordering for each demand is as follows:

$$\texttt{mostavailable}(d_1) = [w_1 = \langle 3, 0\rangle, w_3 = \langle 3, 2\rangle, w_2 = \langle 2, 0\rangle]$$
$$\texttt{mostavailable}(d_2) = [w_3 = \langle 5, 0\rangle, w_3 = \langle 3, 0\rangle, w_2 = \langle 2, 0\rangle]$$

First, we can see that w_2 is never considered in this case as it is not available enough. For d_1, both w_1 and w_3 have the 3 availabilities. However, w_3 has two remaining availabilities. This heuristic guesses that those two remaining availabilities could be used elsewhere. In this case, it is used on d_2 where w_3 has all his 5 availabilities. The search always considers w_1 first for d_1 and w_3 first for d_2.

Dynamic value heuristic

The *most available* heuristic works fairly well in practice as seen in Chapter 6. We can, however, improve it by making it adapt dynamically to the search. Instead of one static ordering at the start of the search, we can reorder values at each value selection to select the best worker in the current search tree.

To achieve this, we need to store some state during the search that will backtrack automatically.

1. $occ_{wdp} \in \mathbb{N}$: the number of times worker w already works for position p of demand d.

2. $occ_w \in \mathbb{N}$: the number of times worker w already works in any demands.

3. $a_w \subseteq T_w$: the set of remaining availabilities of the worker w.

We now have three levels of ordering in the heuristic:

1. The first ordering is now the occurrences occ_{wpd} from greatest to smallest values. We prioritize workers that already work for this demand for the longest time.

2. The second ordering is the same as the first static ordering except we now take into account the remaining availabilities of the worker a_w to select the most available worker.

3. The third ordering is the second static ordering except we also take into account the number of times the worker is already assigned occ_w.

Working requirements extension

One pitfall with our aforementioned heuristic is that a worker might have working requirements but not enough availabilities for this worker to be selected by the search. We can slightly improve our most available heuristic by adding a small extension to take into account the working requirements. This extension adds two orderings in front of all other orderings:

1. If r_{max} is equal to occ_w for worker w, we consider this worker last.

2. $\max(r_{min} - occ_w, 0)$ orders the workers by their remaining requirements. We take the workers that have the most requirements first. r_{min} is the minimum requirement for worker w. We take the subtraction of this value with the current occurrences of worker w and we max this result with 0 to avoid having negative values which are not relevant in our case.

For the remaining of this thesis, when we talk about the most available heuristic, we also take into account this extension.

Breaking symmetries

It is fairly easy to see that our problem contains a lot of symmetries between different positions within the same demand. Two positions might require no skill and thus have the same possible workers. We want to avoid as much as possible to consider every permutation of those workers. For this, we use the `lexleq` constraint [16]. This constraint takes two vectors of variables X and Y. It ensures that $x_i \leq y_i \; \forall i$. As we already have an `alldifferent` constraint applied, it ensures $x_i < y_i \; \forall i$. Let $x_i \in X$ be a variable symmetric to $y_i \in Y$ where x_i and y_i are two variables from the same demand occurring at the same time period.

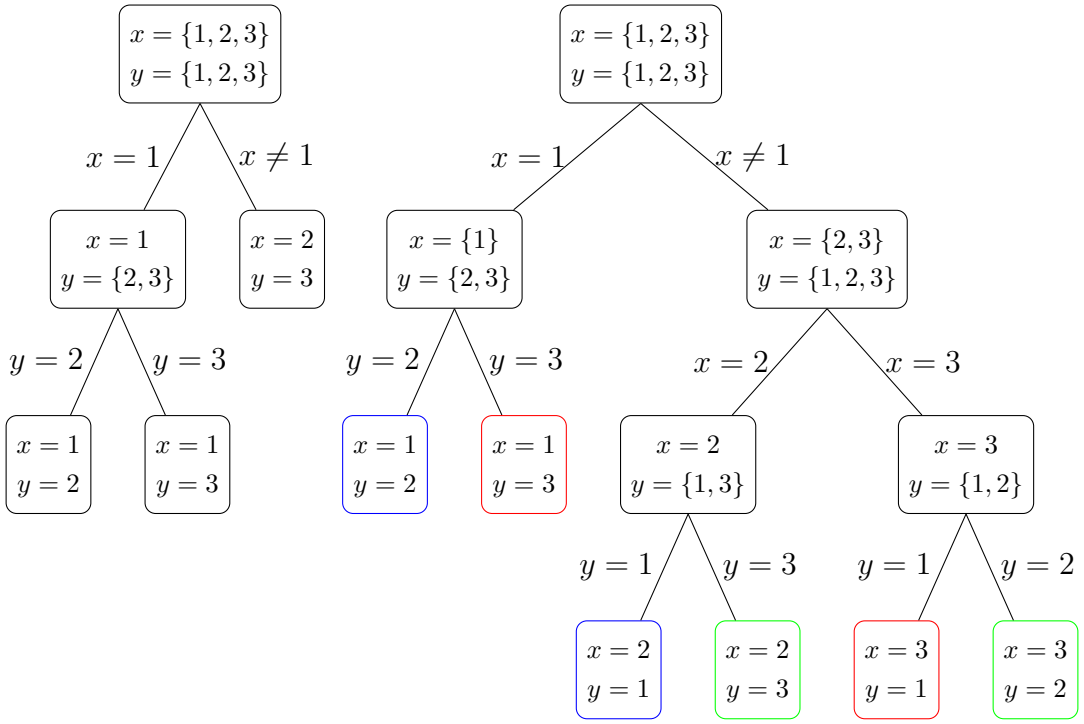

Figure 4.4: Search tree with symmetry breaking (left) and without (right)

$$\texttt{lexleq}(X, Y) \tag{4.41}$$

As a simple example, let us define $x_1 = x_2 = \{1, 2, 3\}$ with $x_1 < x_2$ for symmetry breaking. If x_1 is assigned the value 2, we ignore the permutation $x_1 = 2, x_2 = 1$ because it is symmetric to $x_1 = 1, x_2 = 2$. x_2 is instead directly assigned to the value 3 and thus reducing the search space. Figure 4.4 shows the search trees with and without symmetry breaking. We can see that the search tree with the `lexleq` constraint is reduced.

Large neighborhood search

We use `LNS` to ensure that we explore as much of the search space as possible. We used both random relaxation and propagation guided relaxation [17] to relax our best solution. We discuss and compare these relaxations in Chapter 6.

Variable objective LNS

Our problem uses a multi-objective (4.39) model. Let us define $o_1 = \min v_\sigma$, $o_2 = \min v_r$, $o_3 = \min \sum_{j \in D} \sum_{k \in P(d_j)} N_{jk}$ and o_4 is the original weighted sum

described in (4.39).

We wish to optimize sub-objectives o_1 and o_2 first to avoid partial solutions and unmet requirements respectively.

1. First set o_1 to *Strong-Filtering* while others are set to *No-Filtering*.
2. Once optimized or if it met a threshold, set o_2 to *Strong-Filtering*, o_1 to *Weak-Filtering* and others to *No-Filtering*.
3. Once o_2 is optimized or met a threshold, keep it in *Weak-Filtering* for the rest of the search and switch o_3 to *Strong-Filtering*.

o_4 is also kept in *Strong-Filtering* mode for the entire duration of the search to avoid having a weaker model than the original weighted sum.

Chapter 5

Development and implementation

In this chapter, we describe our implementation for the models presented in Chapter 4. We talk about the difficulties encountered while trying to transform the theoretical model to code. We also discuss some differences between the models and the implementation and some trade-offs that were taken in order to have the most performant solver.

The implementation was done in Scala using *OscaR* (3.3.5) for the `CP` solver and *Gurobi Optimizer* (3.2.1) for the `MIP` solver. The general implementation tries to keep the same `API` (Application Programming Interface) for the `CP` and `MIP` solvers with the only change being optional options that can be passed to it.

5.1 Input and output format

For consistency, both solvers take the same input format and return the same output format. We created a `JSON` (JavaScript Object Notation) Schema [18] to formulate our problems and solution assignments. Those schemas allow us to create a typed data structure for `JSON` objects. All the typing validation is handled by the `JSON` Schema library [19]. A small example of `JSON` schema can be found in Listing 5.1. This example defines a client structure which takes a required string property called *name*.

```
1  "client": {
2    "type": "object",
3    "properties": {
4      "name": {
5        "type": "string"
6      }
7    },
```

```
    "required": ["name"]
  }
```

Listing 5.1: JSON Schema example

The data is then parsed into an immutable data structure in Scala which looks like this:

```
case class Problem(
  T: Int,
  demands: Array[Demand],
  workers: Array[Worker],
  clients: Array[Client],
  locations: Array[Location] = Array(),
  machines: Array[Machine] = Array(),
  workerWorkerIncompatibilities: Array[Array[Int]] = Array(),
  workerClientIncompatibilities: Array[Array[Int]] = Array(),
  workingRequirements: Array[WorkingRequirement] = Array(),
  initialSolution: Option[Solution] = None
)
```

Listing 5.2: Problem structure in Scala

5.2 Common solver API

Both solvers implement a `solve` function which takes the same set of parameters. This function can take generic options implemented by the subclasses (i.e. specific `MIP` or `CP` options). The solve function returns a data structure containing a solution and the time spent in the solver and an optional solution.

```
case class SearchResult(solution: Option[Solution], time: Long)

trait SearchOptions

trait Search[T <: SearchOptions] {
  def solve(timeLimit: Int, solutionLimit: Int, silent: Boolean,
      options: Option[T] = None): SearchResult
}
```

Listing 5.3: Solver `API`

5.3 Mixed Integer Programming solver

We used the Java `API` [20] of the Gurobi Optimizer in Scala to create our implementation. The implementation did not change from the theoretical model presented in Section 4.1 as `MIP` solvers are less flexible in their modeling abilities than `CP` solvers as we discuss later. The Gurobi solver comes with default parameters [21], it is advised to keep default parameters as changing them does not give much gain. Multiple different parameters were tested but as advised from the Gurobi website, no change was noticed.

5.4 Constraint Programming solver

The Constraint Programming implementation differs in some parts from the theoretical model presented in Section 4.2. In Constraint Programming, the constraint propagation takes the most time in the solving algorithm. Propagations may be unnecessarily too strong for a model. This is what happened with our model and the use of `softgcc` constraints.

The minimization of dummy workers described in (4.37) used a `softgcc` in the model. However, this constraint is slow to propagate in practice due to the high number of variables. Using a `CPU` profiler, we noticed the `softgcc` constraint took up to 20% of the solver runtime. OscaR proposes a variant of the `gcc` constraint which simply counts the number of occurrences of values.

```
gcc(x: Array[CPIntVar], o: Array[(Int, CPIntVar)])
```

Listing 5.4: Variant of `gcc` implemented in OscaR

This definition offers a weaker propagation for the variables but is enough for our model. This definition is used as follows:

```
// workerVariables: Array[CPIntVar]
// dummyViolations: CPIntVar
// Constants.DummyWorker: Int = -1
add(
  gcc(workerVariables, Array(
      (Constants.DummyWorker, dummyViolations)
    )
  )
)
```

Listing 5.5: Usage of `gcc` to count dummy workers

We followed the same idea for the working requirements minimization (4.36). `softgcc` also turned out to be too strong. We used a weaker model with the `gcc` described above and computed our own violations, similar to the `softgcc` definition, from the occurrences given by the `gcc`.

```
case class WorkingRequirement(worker: Int, min: Option[Int], max:
    Option[Int])

// ...

val violations: Array[CPIntVar] = Array.fill(requirements.length)(null)

val occurrences = requirements
  .map(_.worker)
  .map(w => (w, CPIntVar(0, workers(w).availabilities.size)))

add(gcc(workerVariables, occurrences))

// For each requirement
for (i <- requirements.indices) {
  val r = requirements(i)
  violations(i) = maximum(Array(
      occurrences(i)._2 -
          r.max.getOrElse(workers(r.worker).availabilities.size),
      -occurrences(i)._2 + r.min.getOrElse(0),
      CPIntVar(Set(0))
    )
  )
}

// workingRequirementsViolations: CPIntVar
add(sum(violations, workingRequirementsViolations))
```

Listing 5.6: Usage of `gcc` to count working requirements violations

5.5 Instances generation

Randomized instances were needed to be able to test our solvers. Unfortunately, we were not able to have real testing data to base our generation on.

A generator was implemented with a series of options to create different types of instances. The options are the following:

```scala
case class InstanceOptions(
  t: Int,                                  // Number of periods
  clients: Int,                            // Number of clients
  demands: Int,                            // Number of demands
  workers: Int,                            // Number of workers
  skills: Int,                             // Number of skills
  locations: Int = 0,                      // Number of locations
  machines: Int = 0,                       // Number of machines
  probabilities: Map[String, Double] = Map(
    "assignSkill" -> 0.2,                  // Assign skill to demand
    "assignWorkerSkill" -> 0.2,            // Assign skill to worker
    "assignPeriod" -> 0.6,                 // Assign period to demand
    "assignLocation" -> 0.5,               // Assign location to demand
    "assignMachines" -> 0.3,               // Assign machines to demand
    "takeMachine" -> 0.2,                  // Assign a machine to demand
    "assignWorkingRequirements" -> 0.2, // Assign requirements to worker
    "assignWWI" -> 0.05,                   // Assign worker-worker
                                           // incompatibility for each
                                              worker
    "assignWCI" -> 0.05                    // Assign worker-client
                                           // incompatibility for each
                                              worker
  )
)
```

Listing 5.7: Instance options

This represents almost all the parameters that an instance can have. We created a map of probabilities to generate easier or harder instances. For example, the `assignSkill` value is responsible for the probability of a position to be assigned a skill. We can increase this value if we want more skilled positions and vice-versa.

We also needed to be able to reproduce instances, the `InstanceGenerator API` can take a seed parameter which defaults to 0.

```scala
class InstanceGenerator(val seed: Long = 0L) {
  def generate(options: InstanceOptions): Problem
}
```

Listing 5.8: Instance generator `API`

The generator makes sure that a solution is always possible by assigning periods to workers that have been assigned to demands. It also adds the demand periods

to k random workers where k is the number of required workers for that demand. This allows to have more variety in the possible solutions.

The generator tries as much as possible to create meaningful instances but it is of course not able to replicate the importance of real data properly.

5.6 Benchmark runner

A benchmark `API` is created on top of the instances generation `API`. The `API` takes an options structure (Listing 5.9). The arrays `T`, `D` and `W` determine the sizes of the instances. The benchmark runner will create instances with a combination of those parameters. For example, with the values in Listing 5.9, the benchmark runner creates six instances with the sizes: $\langle 5, 30, 100 \rangle$, $\langle 5, 50, 100 \rangle$, $\langle 5, 30, 200 \rangle$, $(5, 50, 200\rangle$, $\langle 5, 30, 300 \rangle$ and $\langle 5, 50, 300 \rangle$ with the three values being $\langle T, D, W \rangle$.

```
1  trait BenchmarkOptions {
2    val solutionLimit: Int = Int.MaxValue
3    val timeLimit: Int = 20
4    val repeat: Int = 1
5    val dryRun: Int = 1
6    val T: Array[Int] = Array(5)
7    val D: Array[Int] = Array(30, 50)
8    val W: Array[Int] = Array(100, 200, 300)
9    val probabilities: Map[String, Double] = Map()
10   val seed: Long = -1L
11 }
```

Listing 5.9: Benchmark options

We can specify a solution limit as well as a time limit. We can also repeat our benchmark and takes the average of the results. We can also have dry runs to warm up the `JVM` (Java Virtual Machine) and a seed to have reproducible benchmarks. Finally, we can specify the probabilities for our instances as explained in Section 5.5.

We then pass these options to our benchmark runner in addition to our solver functions. The runner then returns multiple results such as the time taken by the solvers, the final objective and the objective over time.

Chapter 6

Experiments

In this chapter, we experiment with our solvers and discuss the results.

6.1 Benchmark process

We used our benchmark runner described in Section 5.6 to run our experiments. Once all the tested solvers have finished their run, we extract the results and create a `JSON` file with the final results (objective, time, etc.). We can then create plots that make it easier to analyze and discuss the results. Two of those plots are *timelines* and *performance profiles*.

6.1.1 Timelines

A timeline is a decreasing function of the objective over time. It represents the average evolution of the objective of all instances over time. However, as the scale of the objective can be significantly different from one instance to another, all objectives are normalized to a $[0, 1]$ range. We call this the relative distance to the best objective. The normalized objective (no) for instance i at each timestamp t can be computed with the following formula:

$$no_i(t) = \frac{objective_i(t) - best_i}{objective_i(0) - best_i}$$

A value of 1 is always the initial solution for each solver, even though each solver might have a different objective value for its initial solution. A value of 0 is the best solution found at the end of the search.

For each time t, we take the average of all normalized objectives over all instances:

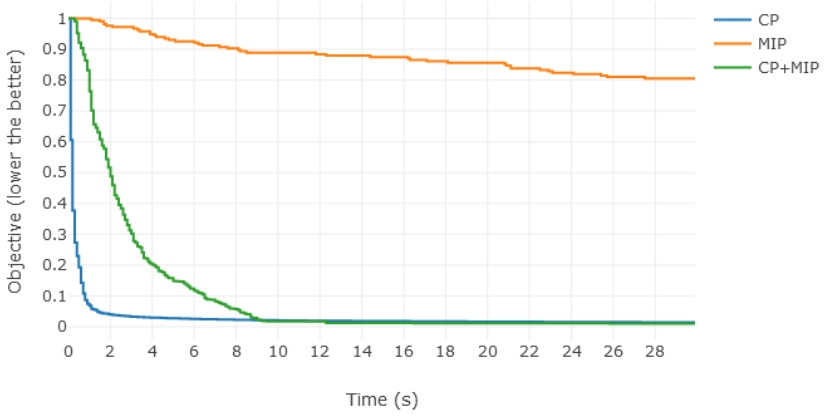

Figure 6.1: Example of a timeline.

$$no(t) = \frac{\sum_{i \in \mathcal{I}} no_i(t)}{|\mathcal{I}|}$$

Figure 6.1 shows an example of a timeline with three solvers. The x axis is the time in seconds and the y axis is the normalized objective value. In practice, we do not compute the normalized objective for every second but for every 1/10th second for more accurate results.

6.1.2 Performance profiles

A performance profile [22] is a cumulative distribution function of a performance metric (e.g. objective, time, etc.). In our case, we are mostly interested in the objective metric after a fixed amount of time. This is how a performance profile is computed:

For each problem p and solver s, we define two performance metrics:

$$t_{p,s} = \text{time required to solve } p \text{ by solver } s$$
$$o_{p,s} = \text{objective of } p \text{ obtained by solver } s$$

From these results, we can compute a performance ratio $r_{p,s}$ for a performance metric. We use $m_{p,s}$ in the following formulas which refers to a performance metric such as $t_{p,s}$ or $o_{p,s}$.

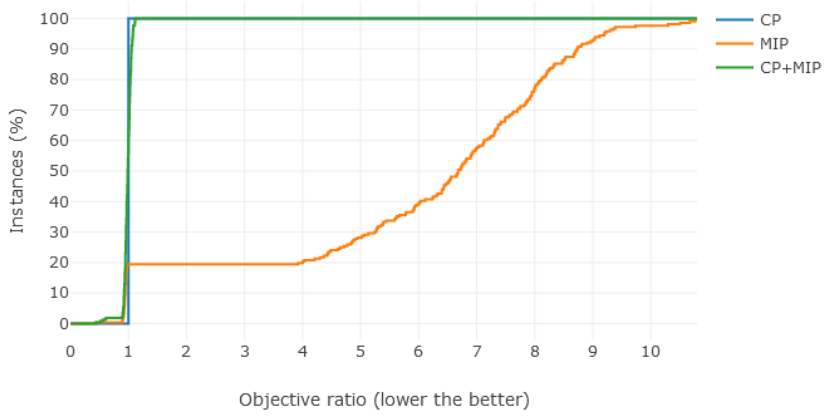

Figure 6.2: Example of performance profiles.

$$r_{p,s} = \frac{m_{p,s}}{min\{m_{p,s} : s \in \mathcal{S}\}} \tag{6.1}$$

This ratio is the comparison between the performance of a solver s with the best solver for p. This means that the best solver is used as baseline. However, it can be interesting to change the baseline to change the comparison. For this, we change the formula [23] to:

$$r_{p,s} = \frac{m_{p,s}}{min\{m_{p,b} : b \in \mathcal{B}\}} \tag{6.2}$$

where $\mathcal{B} \subseteq \mathcal{S}$ is the set of baselines. The performance profile of a solver is then given by:

$$F_s(\tau) = \frac{1}{|\mathcal{P}|} |\{p \in \mathcal{P} : r_{p,s} \leq \tau\}| \tag{6.3}$$

with $\tau \in \mathbb{R}$ being a performance factor. In other words, $F_s(\tau)$ is the cumulative probability of having a performance ratio within a factor τ of the best possible ratio.

We can plot multiple performance profiles of different solvers, with different baselines, to compare them. Figure 6.2 shows an example of performance profiles with the objective as performance metric. These profiles represent the objective after a fixed amount of time with three solvers `MIP`, `CP` and `CP+MIP` with `CP` as baseline.

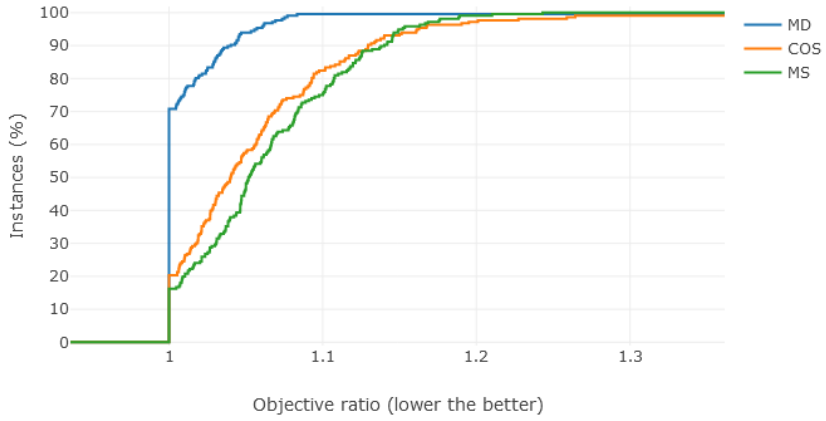

Figure 6.3: Example of performance profiles with all solvers as baselines.

We can see on the y axis the percentage of instances solved by the solvers and on the x axis the objective expressed as a ratio of the objective obtained by the baseline. We can see in this example that for around 60% of instances, the objective is better or equal for the `CP+MIP` solver with an improvement of up to 10% compared to the baseline. In the remaining 40% of instances, we can see that the `CP+MIP` solver is worse also up to around 10%.

We can also have multiple solvers as baseline as we can see in Figure 6.3 where all the solvers are used as baselines. In this type of profile, the objective used in the objective ratio is the best objective between all solvers.

6.1.3 Hardware

All the experiments were performed on a UCLouvain server [24] from the INGI department.

- Intel(R) Xeon(R) CPU E5-2687W v3 `@` 3.10GHz
  - 20 cores available.
  - 40 threads.
- 128 Go RAM

The tests were run mostly during the night due to the long running times and to avoid having perturbations from other users. However, as this machine has a

high number of cores, running tests during the day did not impact the results. The server load was monitored through the INGI website [25] and the server did not show any sign of overload.

6.1.4 Benchmark instances

For our experiments, we generated 216 instances of different sizes. The problem sizes were based on the needs of the Village nº1 company. Table 6.1 shows the size proportions for the 216 instances used in the experiments. T represents the number of time period, D the number of demands and W the number of workers. The generated instances also have varying probabilities as presented in Listing 5.7. These probabilities are referenced in Table 6.2 and follow a uniform distribution where $P_\mu = \frac{P_{min}+P_{max}}{2}$.

Table 6.1: Proportions of instance sizes for 216 instances.

Size			Prop.	
T	D	W	n	%
5	30	150	8	3.7
		225	8	3.7
		300	8	3.7
	40	150	8	3.7
		225	8	3.7
		300	8	3.7
	50	150	8	3.7
		225	8	3.7
		300	8	3.7

Size			Prop.	
T	D	W	n	%
10	30	150	8	3.7
		225	8	3.7
		300	8	3.7
	40	150	8	3.7
		225	8	3.7
		300	8	3.7
	50	150	8	3.7
		225	8	3.7
		300	8	3.7

Size			Prop.	
T	D	W	n	%
15	30	150	8	3.7
		225	8	3.7
		300	8	3.7
	40	150	8	3.7
		225	8	3.7
		300	8	3.7
	50	150	8	3.7
		225	8	3.7
		300	8	3.7

Table 6.2: Probability range for generated instances.

P_{name}	P_{min}	P_{max}
assignSkill	0.1	0.3
assignWorkerSkill	0.1	0.3
assignPeriod	0.4	0.8
assignLocation	0.3	0.7
assignMachines	0.1	0.5
takeMachine	0.1	0.3
assignWorkingRequirements	0.1	0.3
assignWWI	0	0.1
assignWCI	0	0.1

Locations and machines do not affect the results as much as the number of workers, demands, and time periods. This is due to the fact that locations and machines have a low number of constraints and only need to be assigned once and are neither optimized nor relaxed (in `CP`). Because of this, we simply add a fixed number of locations and machines (i.e. 30 each) throughout the entire set of instances.

6.1.5 Model parameters

We presented in Equation 4.20 and Equation 4.40 the penalties of our sub-objectives. We define for our experiments the following penalty vector:

$$\boldsymbol{\delta} = \langle \delta_0, \delta_1, \delta_2 \rangle = \langle 1, 15, 100 \rangle$$

6.2 Constraint Programming

6.2.1 Comparison between heuristics

We talked in Section 4.2 about our custom heuristic called the *most available* heuristic. We now compare the performance of this heuristic with standard heuristics such as the *max value* heuristic. We also compare the variable heuristics used in addition to our aforementioned heuristic.

Value heuristic

We compare multiple value heuristics:

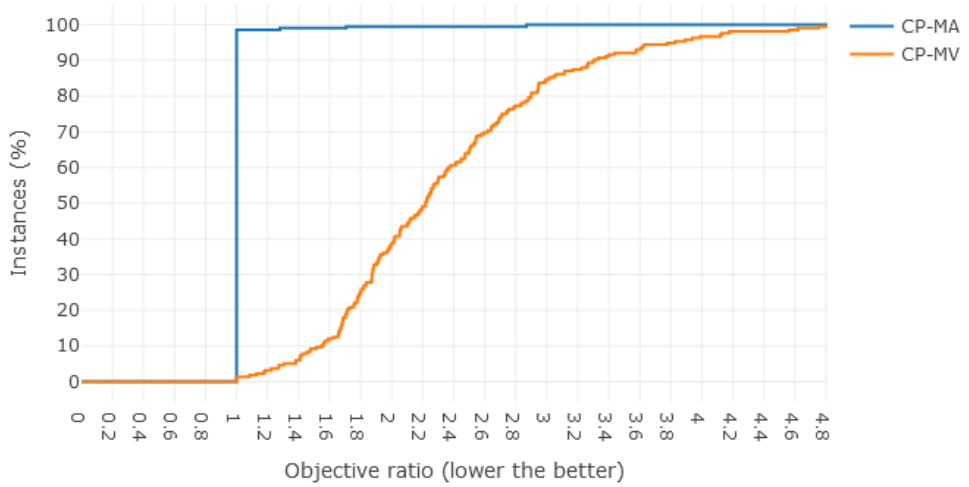

Figure 6.4: Performance profiles of *most available* and *max value* heuristics [216 instances/30s].

- The *max value* heuristic (`CP-MV`). This heuristic simply takes the maximum value in the domain of the variable. We take the maximum instead of the minimum because of the dummy value being equal to -1 in every domain.
- The *most available* heuristic discussed in Section 4.2 (`CP-MA`).
- The *dynamic most available* heuristic also discussed in Section 4.2 (`CP-MA-D`).

We first start by comparing the *max value* and the static *most available* heuristics together.

Figure 6.4 shows performance profiles of the *max value* heuristic compared to the *most available* heuristic as baseline. We observe that our custom heuristic outperforms the *max value* heuristic in almost every instance with up to a 4.8 times improvement after 30 seconds of search.

We can also see from Figure 6.5 that, as the search moves forward, our custom heuristic manages to find better solutions faster.

Heuristics such as the *max value* are ones of the simpler heuristics to implement but often offer bad performances due to the lack of knowledge of the problem.

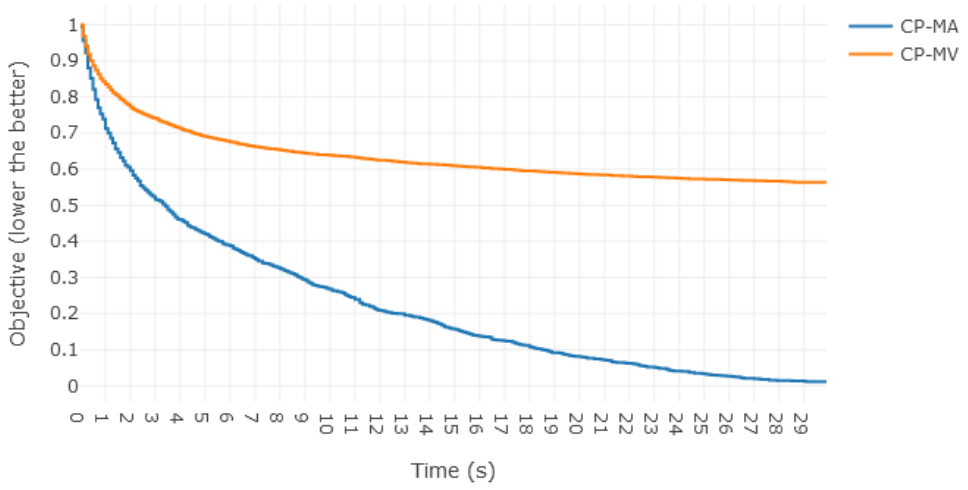

Figure 6.5: Timeline of *most available* and *max value* heuristics [216 instances/30s].

We now compare our static and dynamic *most available* heuristics together. Figure 6.6 shows the performance profiles of the dynamic and static *most available* heuristics. We can see that the dynamic version of the heuristic outperforms the static one in 90% of instances after 30 seconds of search.

Even though the dynamic version needs to process more during the search, we can see from Figure 6.7 that the time lost by this processing is gained back during the search. The first solutions are almost of the same quality but as the search moves forward, better solutions are found with the dynamic version. This is expected from the implementation of the dynamic version which finds the best worker to assign at the current state of the search.

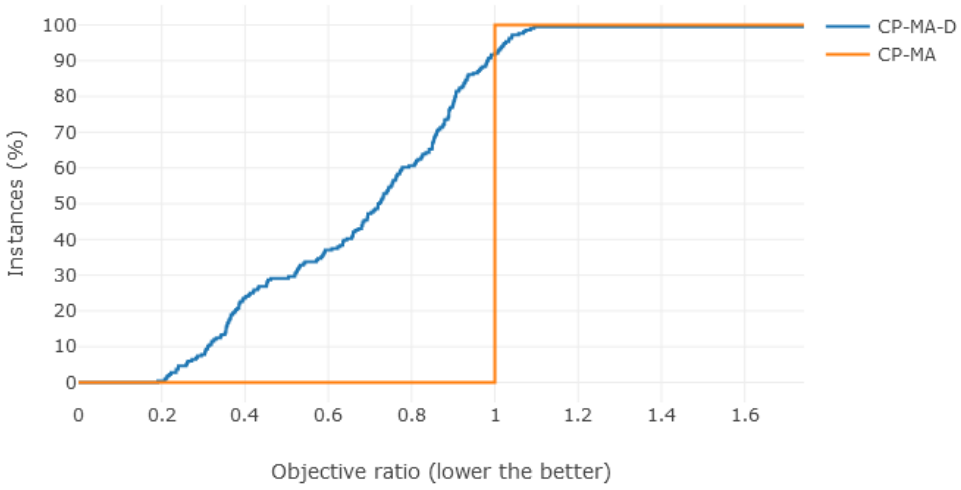

Figure 6.6: Performance profiles of static and dynamic *most available* heuristics with `CP-MA` as baseline [216 instances/30s].

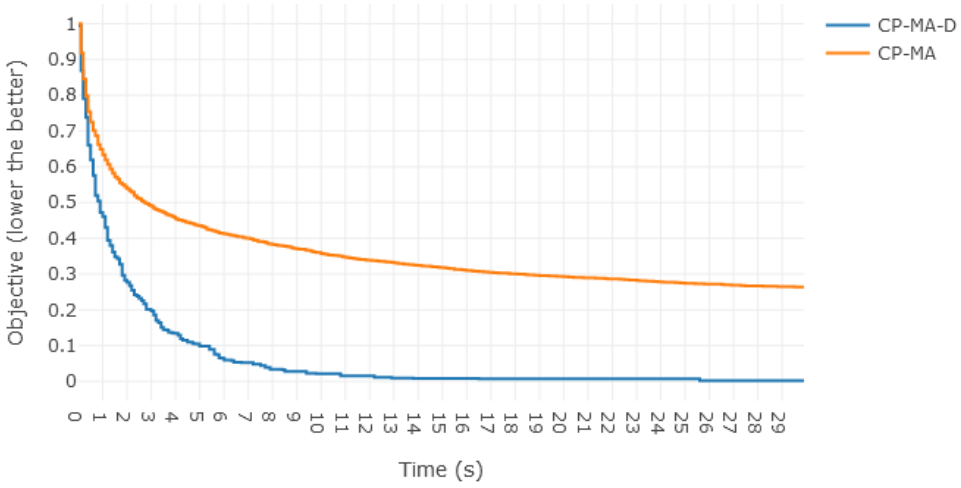

Figure 6.7: Timeline of static and dynamic *most available* heuristics [216 instances/30s].

As our dynamic value heuristic outperforms other tested value heuristics, we assume for the rest of this chapter that the value heuristic for the Constraint Programming solver is the dynamic *most available*.

Variable heuristic

In addition to our most available heuristic, we use a variation of the *max degree* heuristic. This heuristic is a first-fail variable heuristic that selects the most constrained unbound variable. However, as stated in our model in Section 4.2, skills are not represented with constraints but instead, values are removed from the domain at initialization. To express skills as part of the degree, we simply add the number of skills required by a variable to the degree of that variable.

We compare multiple variable heuristics:

- The custom *max degree* (`MD`) heuristic
- The *min size* (`MS`) heuristic. This heuristic simply selects the variable with minimum domain size.
- The *conflict ordering search* (`COS`) heuristic [26]

Conflict ordering search is a variable ordering heuristic that reorders variables based on the number of conflicts that happen during the search. It is a variant to the *last conflict* heuristic which selects the variable which caused the last conflict first. `COS` was shown in [26] to be the most performant on scheduling problems. We now see how it performs in our problem in comparison with the aforementioned heuristics.

Figure 6.8 shows the performance profiles of the objective after 30 seconds for each solver. We can see that the *conflict ordering search* and *min size* heuristics are underperforming in comparison to the *max degree* heuristic.

However, as we can see from the first five seconds of search in Figure 6.9, `MD` performs slightly worse than `COS` and `MS` for the first second of search. If a solver only runs for one second, using `COS` or `MS` would be preferable.

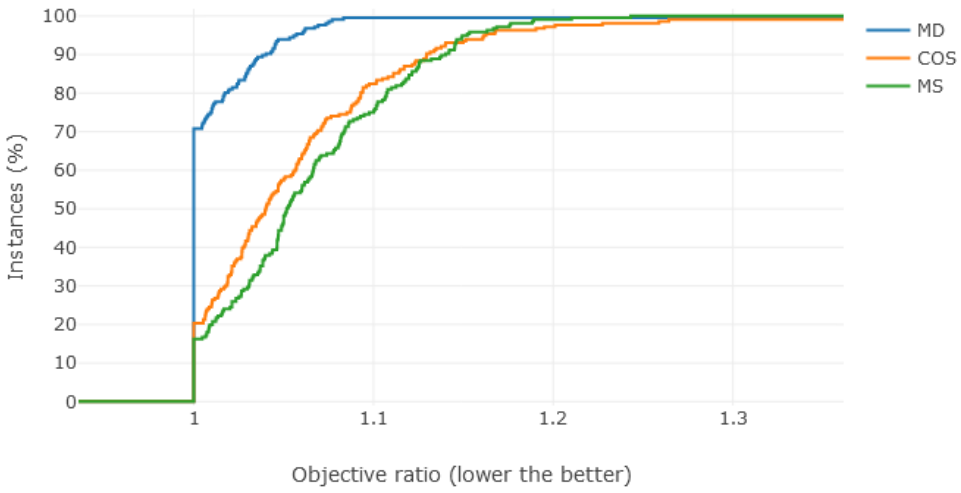

Figure 6.8: Performance profiles of `COS`, `MD` and `MS` heuristics [216 instances/30s].

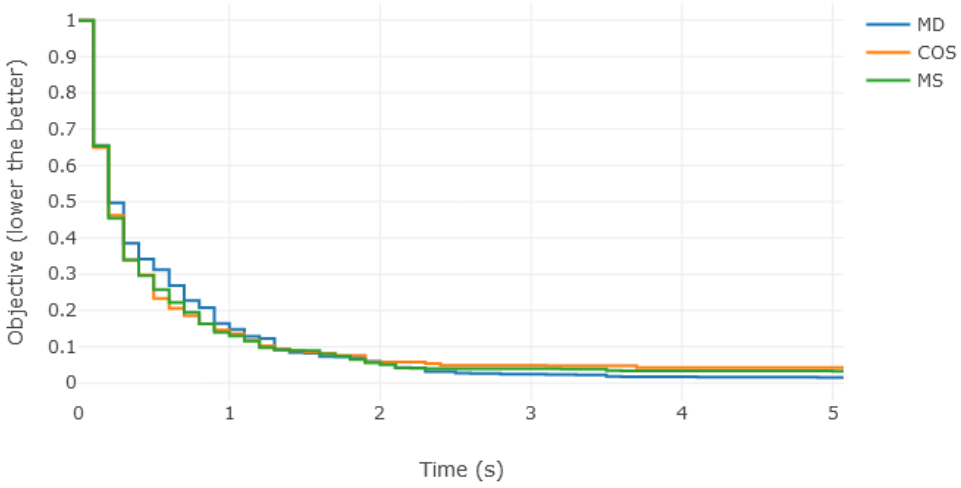

Figure 6.9: Timeline of `COS`, `MD` and `MS` heuristics for the first five seconds of search [216 instances/5s].

6.2.2 Comparison between searches

Large neighborhood search

As stated in Chapter 3 and Section 4.2, `LNS` is used to expand the exploration of the search tree. We now compare our solver with and without the use of `LNS`. We also compare multiple relaxation techniques.

Figure 6.10 shows the performance profiles of the final objective after 30 seconds of search between a standard search and `LNS`. As expected, we can see that `LNS` outperforms the standard search in almost every instance.

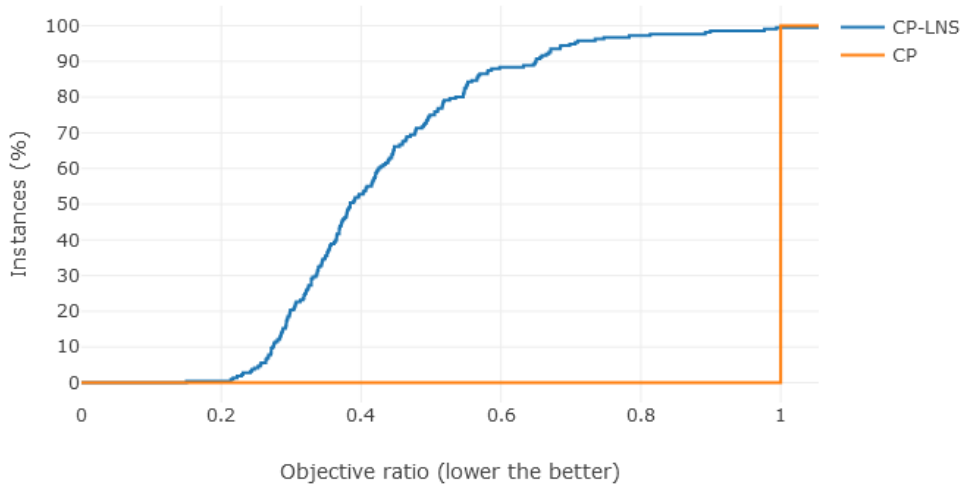

Figure 6.10: Performance profiles of `CP` and `CP-LNS` with `CP` as baseline [216 instances/30s].

Figure 6.11 shows that the standard search quickly gets stuck while the `LNS` manages to find better solutions quickly after relaxing the initial solution.

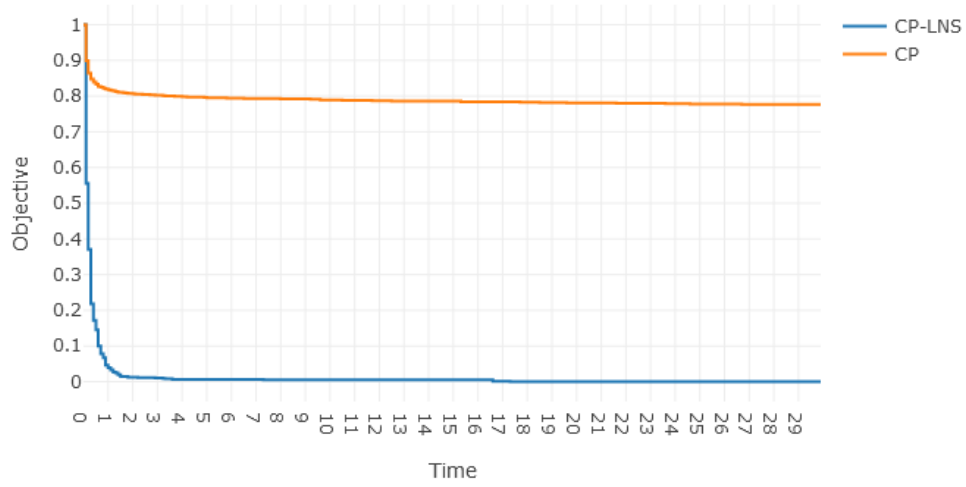

Figure 6.11: Timeline of `CP` and `CP-LNS` [216 instances/30s].

The trouble of the standard search to find good solutions is also due to the lack of `VO-LNS` (Section 4.2.3) as this method is an extension of the `LNS` framework. The standard search can only minimize the weighted sum of sub-objectives which causes issues when trying to minimize more important (more weighted) sub-objectives first.

Relaxations

We also experiment with multiple relaxations within the `LNS` framework. We compare:

- Random relaxation:
  - 10% through 90% relaxation (`CP-Random-{P}%`).
- Propagation based relaxation (`CP-Prop`).

Figure 6.12 first shows performance profiles of all random relaxations from 10% through 90% relaxation. We observe a trend on this graph that the more relaxation we have, the worst the objective is. Out of all the parameters, the 10% and 20% relaxations offer the best performances after 30 seconds of search. However, as we

can see from Figure 6.13, the 40% relaxation offers a slightly better objective at the start and the 10% and 20% catch up after 9 seconds of search.

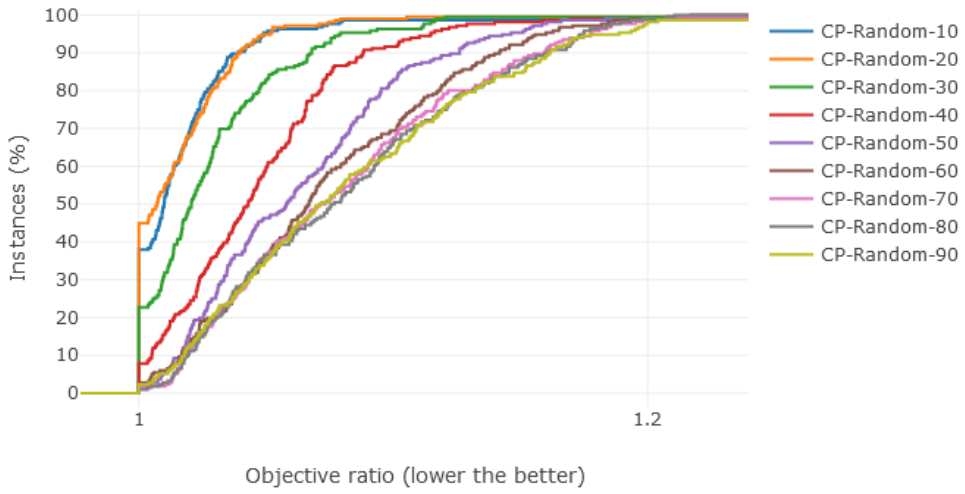

Figure 6.12: Performance profiles of multiple random relaxations [216 instances/30s].

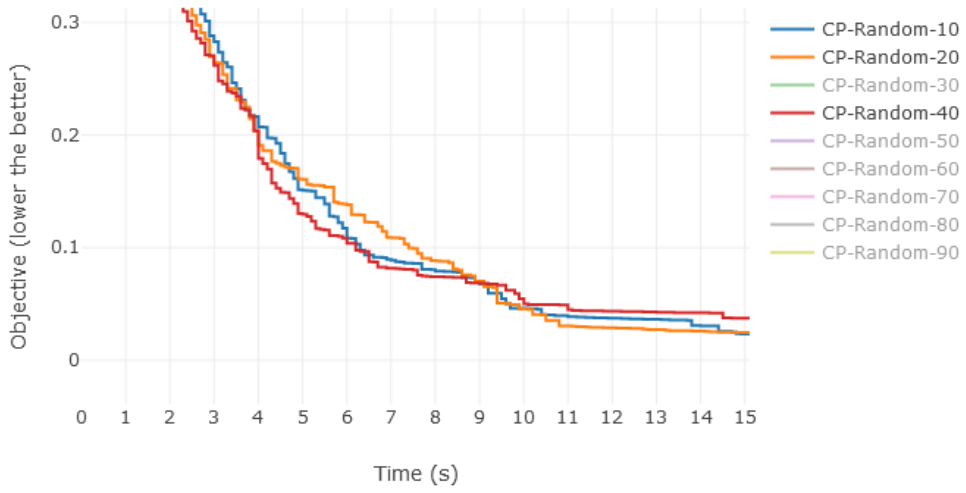

Figure 6.13: Timeline of multiple random relaxations [216 instances/15s].

We now compare our best random relaxation with a propagation based relaxation. We choose `CP-Random-20` to compare our propagation based relaxation with, as the performances of the two best random relaxations are almost the same. The propagation guided relaxation takes a neighborhood size parameter, we compare five values for this parameter:

- `CP-Prop-1/3`: the size of the neighborhood to attain is one third of the size of variables.
- `CP-Prop-1/2`: the size of the neighborhood to attain is half the size of variables.
- `CP-Prop-1`: the size of the neighborhood to attain is the size of variables.
- `CP-Prop-2`: the size of the neighborhood to attain is twice the size of variables.
- `CP-Prop-3`: the size of the neighborhood to attain is thrice the size of variables.

Figure 6.14 shows the propagation based relaxations in comparison to `CP-Random-20` as baseline. We can see that all parameters of the propagation based relaxations almost all have the same performances and are overall worse than the best random

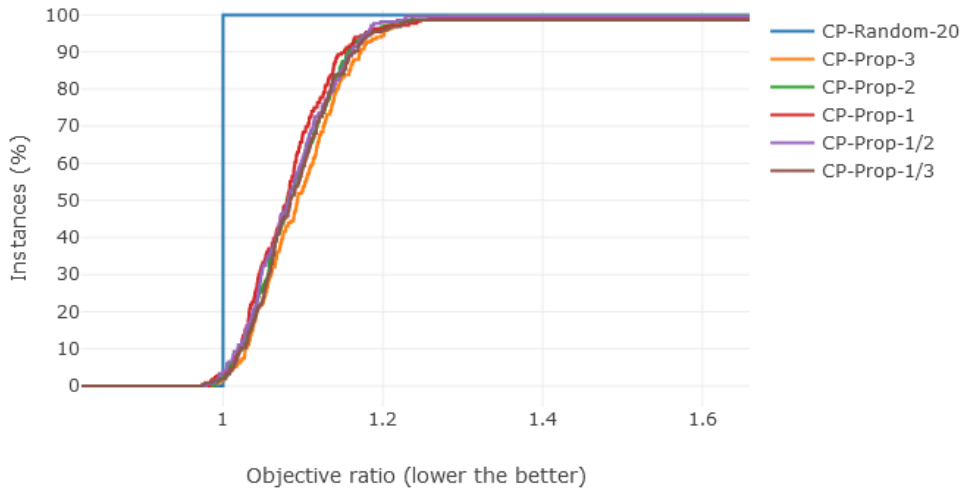

Figure 6.14: Performance profiles of multiple propagation based relaxations in comparison with the best random relaxation [216 instances/30s].

relaxation. Approximately only 5% of instances are best solved by a propagation based relaxation. This makes the random relaxation with 20% relaxed variables the best out of all tested relaxations.

6.3 Comparison between solvers

We now start by comparing different solvers together. We experiment with three solvers:

- The Constraint Programming (`CP`) solver.
- The Mixed Integer Programming (`MIP`) solver.
- A combination of `CP` and `MIP` (`CP+MIP`) solvers. We take an initial solution from the `CP` solver and give it to the `MIP` solver as start solution.

Figure 6.15 shows performance profiles of `CP`, `MIP` and `CP+MIP` solvers. We can clearly see that the `MIP` solver underperforms for most instances. However, it slightly outperforms the `CP` solver for approximately 20% of instances.

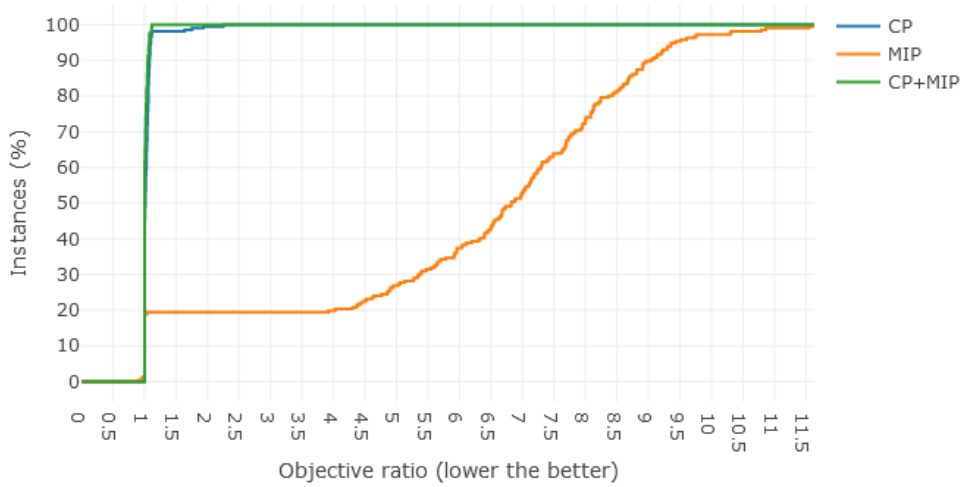

Figure 6.15: Performance profiles of `CP`, `MIP` and `CP+MIP` solvers [216 instances/30s].

From Table 6.3, we can observe that only small instances are best solved by the `MIP` solver. This highlights the fact that `MIP` has trouble finding good objectives as the problem size grows. The problem size directly correlates with the number of binary variables of the `MIP` model. As an example, if the problem contains 50 demands over 15 periods with 300 workers and we have 3 workers per demand on average, we obtain 675.000 binary variables which is more than ten times the number of variables for small instances shown in Table 6.3. `MIP` is `NP`-complete and this high number of variables makes it hard for the solver to find good solutions.

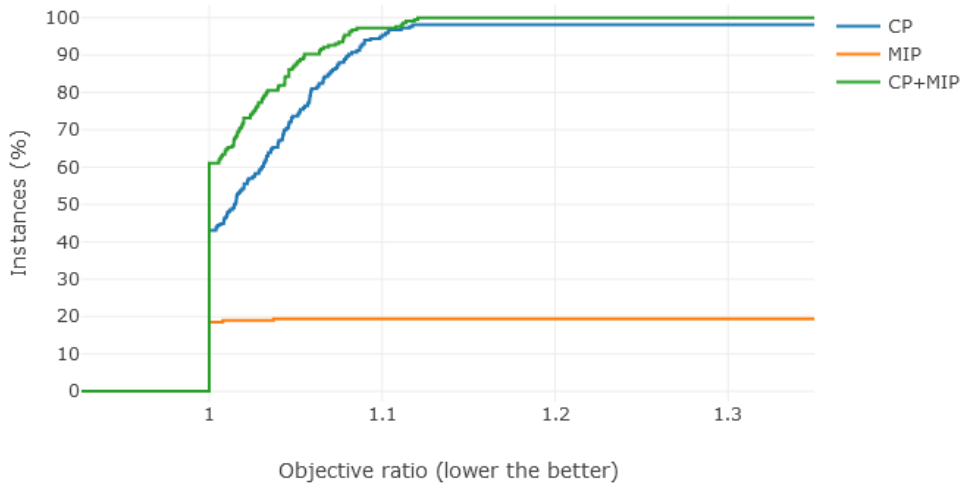

Figure 6.16: Performance profiles of `CP`, `MIP`, and `CP+MIP` solvers within a smaller objective range than Figure 6.15 [216 instances/30s].

Table 6.3: Instance sizes best solved by the `MIP` solver over the 216 tested instances.

Size			Prop.
T	D	W	n
5	30	150	8
		225	4
		300	1
	40	150	8
		225	4
	50	150	8
		225	7
Total			40

We can also observe from these performance profiles, and in Figure 6.16, that the `CP+MIP` solver gives the best objective in 60% of instances. For the remaining instances, the objective is only worse by a maximum of 10%. The `MIP` solver performs a lot better when given an initial solution to work with.

Figure 6.17 gives the objective per time for the three solvers. As expected from our previous results, the `MIP` solver does not manage to find a good solution after

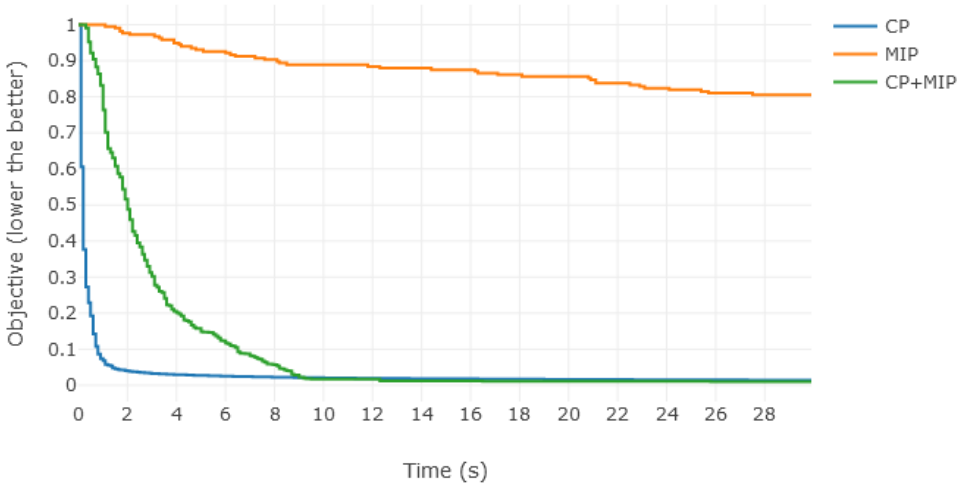

Figure 6.17: Timeline of `CP`, `MIP`, and `CP+MIP` solvers [216 instances/30s].

the elapsed time. However, the `CP` solver manages to find good solutions in less than one second. While the `CP` solvers finds a good solution quickly, the `CP+MIP` solver finds a better solution but takes on average ten seconds to catch up with `CP` solver.

Chapter 7

Conclusion

In this thesis, we presented an optimization problem for production planning with resource allocation.

We introduced two models that solve this problem: a Mixed Integer Programming and a Constraint Programming model. The Constraint Programming model required the most work with custom heuristics that take into account knowledge of the problem.

The models were implemented in Scala using OscaR and Gurobi for the Constraint Programming and Mixed Integer Programming models respectively.

We tested our models on generated test instances of different sizes. We also showed and discussed the results of these tests. We saw from these results that our Constraint Programming model greatly outperforms its Mixed Integer Programming counterpart. However, the Mixed Integer Programming model performs better in most cases when given an initial solution to work with.

Mixed Integer Programming is a great technique to solve combinatorial problems. However, most of our instances have too many variables for Mixed Integer Programming to handle. On the other hand, our Constraint Programming model contains more knowledge of the problem during the search. It allows us to implement custom heuristics such as the one presented in Section 4.2. Hence, it manages to find better solutions faster.

From the work presented in this thesis, we can conclude that Mixed Integer Programming and Constraint Programming both have their advantages and disadvantages. `MIP` has the advantage of having its search algorithms already implemented in the solver (e.g. Gurobi). There is no need to implement additional heuristics which may be time consuming. However, this lack of knowledge about the problem makes it harder to find solutions on bigger instances. Implementing complex constraints also complicates the model making it harder to understand.

The `CP` model offers a better expressiveness which makes it easier to understand. We can take advantage of the local search framework the explore more of the search

space. However, finding optimal solutions with LNS becomes harder because of the nature of the search (i.e. restarts). CP with LNS is instead more efficient at finding good feasible solutions in a short amount of time.

By taking the fast good feasible solution property from CP and the good optimization property from MIP, we can combine these two search techniques and gain from the advantages of both.

Finally, we discuss in 7.1 some improvements that could be made to our models and solvers.

7.1 Further Work

Contiguous shifts for the CP model : Our implementation of the contiguous shifts constraint only takes into account the number of different workers working for a position throughout the demand planning period. We tested the implementation with real contiguous shifts with a decomposition of equality constraints. However, the results were disappointing as the decomposition slowed down the search. While our implementation gives us really good results for contiguous shifts, we could replace our constraint by another soft global constraint that would be more efficient than a decomposition of equality constraints.

Generic resource allocation : While our location and machine resources could be replaced with any resources with a one to one and one to many relationship respectively, our Scala implementation only allows to describe locations and machines by name. Our model preprocessing could be more complete to allow the identification of a resource with any properties.

Self-Adaptive LNS : Our solver only uses one relaxation method during the search. Even though it already performs well, we could make use of the self-adaptive large neighborhood search [27] to dynamically and automatically change the relaxation method to the best available one during the search.

Multi-objective search : Our CP solver uses the VO-LNS framework to optimize sub-objectives separately. We could explore other approaches such as a *Pareto* optimization.

Bibliography

[1] "Gurobi - MIP Basics." `http://www.gurobi.com/resources/getting-started/mip-basics`. Accessed: 2019-03-13.

[2] E. K. Burke, P. De Causmaecker, G. V. Berghe, and H. Van Landeghem, "The state of the art of nurse rostering," *Journal of Scheduling*, vol. 7, pp. 441–499, Nov 2004.

[3] T. Osogami and H. Imai, "Classification of various neighborhood operations for the nurse scheduling problem," in *ISAAC*, 2000.

[4] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2018.

[5] P. van Beek and X. Chen, "Cplan: A constraint programming approach to planning," in *AAAI/IAAI*, 1999.

[6] C. Bessière and P. Van Hentenryck, "To be or not to be ... a global constraint," vol. 2833, pp. 789–794, 11 2003.

[7] J.-C. Régin, T. Petit, C. Bessiere, and J.-F. Puget, "An original constraint based approach for solving over constrained problems.," pp. 543–548, 01 2000.

[8] J.-C. Régin, "A filtering algorithm for constraints of difference in csps," in *AAAI*, 1994.

[9] J.-C. Régin, "Generalized arc consistency for global cardinality constraint," in *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*, AAAI'96, pp. 209–215, AAAI Press, 1996.

[10] W.-J. Van Hoeve, G. Pesant, and L.-M. Rousseau, "On global warming: Flow-based soft global constraints," *Journal of Heuristics*, vol. 12, pp. 347–373, Sep 2006.

[11] P. Schaus, P. Van Hentenryck, and A. Zanarini, "Revisiting the soft global cardinality constraint," vol. 6140, pp. 307–312, 06 2010.

[12] P. Schaus, "Variable objective large neighborhood search: A practical approach to solve over-constrained problems," pp. 971–978, 11 2013.

[13] OscaR Team, "OscaR: Scala in OR," 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[14] J.-B. Mairy, P. Van Hentenryck, and Y. Deville, "An optimal filtering algorithm for table constraints," in *Principles and Practice of Constraint Programming* (M. Milano, ed.), (Berlin, Heidelberg), pp. 496–511, Springer Berlin Heidelberg, 2012.

[15] C. Bessiere, G. Katsirelos, N. Narodytska, C.-G. Quimper, and T. Walsh, "Decomposition of the nvalue constraint," in *Principles and Practice of Constraint Programming – CP 2010* (D. Cohen, ed.), (Berlin, Heidelberg), pp. 114–128, Springer Berlin Heidelberg, 2010.

[16] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh, "Global constraints for lexicographic orderings," in *Principles and Practice of Constraint Programming - CP 2002* (P. Van Hentenryck, ed.), (Berlin, Heidelberg), pp. 93–108, Springer Berlin Heidelberg, 2002.

[17] L. Perron, P. Shaw, and V. Furnon, "Propagation guided large neighborhood search," in *Principles and Practice of Constraint Programming – CP 2004* (M. Wallace, ed.), (Berlin, Heidelberg), pp. 468–481, Springer Berlin Heidelberg, 2004.

[18] "JSON Schema Specification." `https://json-schema.org/`. Accessed: 2019-04-05.

[19] "Play json schema validator." `https://github.com/eclipsesource/play-json-schema-validator`. Accessed: 2019-04-22.

[20] "Gurobi - Java API." `http://www.gurobi.com/documentation/8.1/refman/java_api_overview.html`. Accessed: 2019-04-05.

[21] "Gurobi - Java API." `https://www.gurobi.com/documentation/8.1/refman/parameters.html`. Accessed: 2019-04-05.

[22] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical Programming*, vol. 91, pp. 201–213, Jan 2002.

[23] S. V. Cauwelaert, M. Lombardi, and P. Schaus, "A visual web tool to perform what-if analysis of optimization approaches," *CoRR*, vol. abs/1703.06042, 2017.

[24] "Jabba server." https://wiki.student.info.ucl.ac.be/Mat%C3%A9riel/Jabba. Accessed: 2019-04-14.

[25] "Ingi servers monitoring." https://wiki.student.info.ucl.ac.be/Mat%c3%a9riel/Monitoring. Accessed: 2019-04-14.

[26] S. Gay, R. Hartert, C. Lecoutre, and P. Schaus, "Conflict ordering search for scheduling problems," vol. 9255, pp. 140–148, 08 2015.

[27] C. Thomas and P. Schaus, *Revisiting the Self-adaptive Large Neighborhood Search*, pp. 557–566. 06 2018.

Appendices

Appendix A

Notations

Set of periods

$$T = \{0, \ldots, n \mid n \in \mathbb{N}\}$$

Set of workers

$$W = \{w_0, \ldots, w_n \mid n \in \mathbb{N}\} \quad \text{Set of workers}$$
$$T_w \subseteq T \quad \text{Availabilities of a worker}$$
$$W_s \subseteq W \quad \text{Workers that satisfy skill } s$$

Set of skills

$$S = \{s_0, \ldots, s_n \mid n \in \mathbb{N}\}$$

Set of clients

$$C = \{c_0, \ldots, c_n \mid n \in \mathbb{N}\}$$

Set of locations

$$L = \{l_0, \ldots, l_n \mid n \in \mathbb{N}\}$$

Set of machines

$$M = \{m_0, \ldots, m_n \mid n \in \mathbb{N}, m_n \in \mathbb{N}\}$$

Set of demands

$D = \{d_0, \ldots, d_n \mid n \in \mathbb{N}\}$	Set of demands
$n_d \in \mathbb{N}$	Required number of workers
$T_d \subseteq T$	Periods in which demand occurs
$c_d \in C$	Client of demand
$S_d \subseteq S$	List of required skills
$s_{d,i} \in S_d$	The ith skill of S_d
$S_d^+ \subseteq S$	List of additional skills
$s_{d,i}^+ \in S_d^+$	The ith skill of S_d^+
$M_d \subseteq M$	List of required machines
$L_d \subseteq L$	List of possible locations
$P(d) = \{0, \ldots, n_d - 1\}$	List of positions
$O(d) = \{d_o \mid T_{d_o} \cap T_d \neq \varnothing, \forall\ d_o \in D, d_o \neq d\}$	List of overlapping demands

Working requirements

$R = \{r_0, \ldots, r_n \mid n \in \mathbb{N}\}$	Set of requirements
r_w	The worker concerned with this requirement
r_{min}	Minimum number of times the worker has to work
r_{max}	Maximum number of times the worker has to work

Set of worker - worker incompatibilities

$$I_w = \{\langle i, j\rangle \in \mathbb{N} \times \mathbb{N} \mid w_i, w_j \in W, w_i \neq w_j\}$$

Set of worker - client incompatibilities

$$I_{wc} = \{\langle i, j\rangle \in \mathbb{N} \times \mathbb{N} \mid w_i \in W, c_j \in C\}$$

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