UCLouvain

École polytechnique de Louvain

Optimization of production planning with resource allocation

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

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

Chapter 1

Introduction

Village n°1 is a Belgian company employing persons with disabilities. They offer services to companies and private individuals such as industrial jobs. They are currently in the process of automating the way they schedule these jobs.

The aim of this thesis is to solve their resource allocation problem automatically using two different techniques: Constraint Programming and Mixed Integer Programming. We then analyze and compare the performance of both models.

TODO
TODO
TODO
TODO

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 a formal definition of both MIP and CP models.

Chapter 5 describes the implementation of the models.

Chapter 6 presents the carried experiments and performance results of both models.

Chapter 7 TODO

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. The fact that they are working with people with handicap means that they have special needs concerning the work that each worker can do.

Our resource allocation problem consists of:

- A planning period (T) with each period ($t \in T$) equal in time.
- 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 working requirements associated with workers ($r \in R$). A requirement r states that a worker r_w needs to work a minimum of r_{min} and maximum of r_{max} times in the problem time window.
- A list of incompatibilities between workers ($(w_1, w_2) \in I_{ww}$). Two incompatible workers cannot work with each other.

– A list of incompatibilities between workers and clients ($(w, c) \in I_{wc}$) A worker incompatible with a client cannot work for that client.

Each demand has:

– A client ($d^c \in C$).

– A given set of time periods ($d^T \subseteq T$).

– A required number of workers per period ($d^w \in \mathbb{N}$).

– Some skills requirements to be fulfilled by different workers ($d^S \subseteq S$). It imposes that some workers have the needed capacities to work at a given position (e.g. package lifter).

– Additional skills requirements to be fulfilled by any workers assigned to that demand (e.g. driver license) ($d^{S^+} \subseteq S$).

– A list of machines to perform the work ($d^M \subseteq M$).

– An eventual list of possible locations where the demand can be performed ($d^L \subseteq L$). Vehicles used to drive the workers to the work location are considered as machines.

Each worker has:

– A list of skills (e.g. package lifter, supervisor, etc.) ($w^S \subseteq S$).

– A list of availabilities at which the worker can work ($w^T \subseteq T$).

This type of staff scheduling problem can be seen as a variant of the well known *Nurses Scheduling Problem* (NSP) [**?**]. This type of problem often contains hard constraints to state restrictions and soft constraints to state preferences.

The goal is to assign workers to multi-skill positions as well as machines and locations to a list of demands over the set of all time slots. Each resource can only be assigned once per time period and needs to satisfy all the constraints stated by the demand.

2.1 Constraints

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.

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.

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.

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 assigned to it.

Each assignment must respect skill restrictions

Each position of a demand might 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.

Worker-worker incompatibilities

Workers might be incompatible with each other. Such workers cannot be assigned together at the same time period.

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.

The required machines must always be assigned

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

No machines should be assigned twice for the same period

A machine is assigned for the entirety of a demand. It can be used for other demands that do not overlap in time with the first one. But it can never be assigned twice for the same time period.

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.

No location should be assigned twice for the same period

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

2.1.2 Soft Constraints

Satisfy the most assignments possible

Each demand need a required number of workers. However, we can assign a fictitious worker to demands and minimize the number of fictitious workers. This is done in the case where there is not enough workers or no worker that satisfy a particular skill. From a modeling point of view, the hard constraint which states that the required number of workers must be satisfied is still satisfied with a fictitious worker.

Contiguous shifts

A demand consists of multiple positions over a period of time. For each position, a worker should keep working at that position for the longest time possible. We want to avoid the hassle of changing shift every time. As this constraint is harder to solve, we express it as a soft constraint and minimize the number of violations.

Working requirements

Workers can have minimum and maximum working periods. 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.

Chapter 3

State of the art

3.1 Nurse Scheduling problem

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 integer. MIP problems are also generally NP-complete.

The most common MIP problems are of the form:

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

(3.1) is the problem objective. $\boldsymbol{c}^T$ is the vector of coefficient, $\boldsymbol{x}$ 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. And finally, (3.4) states the integrality constraints over some or all variables.

MIP problems are usually solved using a branch-and-bound algorithm [**?**]. The process is as follow: we start with the MIP formulation and remove all integrality

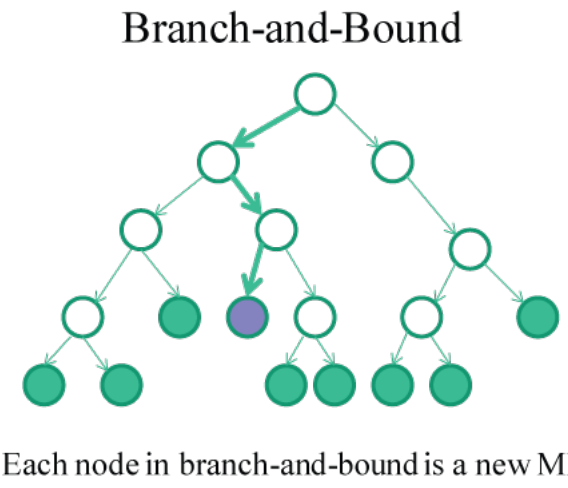

Figure 3.1: MIP Branch & Bound search tree [**?**]

constraints to create a resulting linear-programming (LP) 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) a valid solution is found. More techniques are used to find solution more efficiently. Each solver uses its own algorithm (e.g Gurobi Optimizer [**?**]).

3.2.1 Gurobi Optimizer

The *Gurobi Optimizer* [**?**] 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, 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. A solution to a CSP is an assignment of values $a_i \in D_i$ to x_i, that satisfies all of the constraints. [**?**]

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 [**?**]:

> [...] 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 *operatinal 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 [**?**] 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 *Constraint Optimization Problem* (COP) where we minimize the number of violations.

AllDifferent Constraint

In our resource allocation problem (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 [**?**] is one of 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. It 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 need 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 that a worker is assigned to a position and limit those occurrences to a minimum and maximum.

The global cardinality constraint (`gcc`) [**?**] is a generalization of the `alldifferent` constraint. It does not enforces (although it can) the uniqueness of values of its variables but instead enforces that the cardinality of each value d_i for all its variables in its scope lies between a lowerbound and a upperbound, 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` [**?**]. The violation associated with this constraint is the sum of excess or shortage [**?**] 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 do 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 follow:

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. fix a percentage of variable).
3. Restart

The entire search might 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 (2) uses such 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) [**?**] is an extension of LNS for multi-objective problems which offers (i) Prioritization of sub-objectives; (ii) Better pruning. VO-LNS consists of three types of filtering for each objective

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 sum of all sub-objectives. 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 a good result. We will present in Section 4.2 a value heuristic created for the needs of our staff scheduling 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 will most likely lead to an inconsistency. Multiple heuristics follow this principle, the most simple being the smallest domain ordering.

3.3.5 OscaR

OscaR [**?**] is a Scala toolkit for solving Operations Research problems. OscaR has multiple optimization techniques available: (i) Constraint Programming; (ii) Constraint Based Local Search (CBLS); (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* allocate resources to improve it.

The library of OscaR in which this project is interested in is the Constraint Programming library. It offers a lot of existing constraints and abstractions. Some

black-box searches are also implemented but we can bring our own heuristics to drive the search forward.

Chapter 4

Models for the resource allocation problem

In this chapter, we first start by presenting formal notations used by both models. We then present two models to solve the Village n°1 problem: a Mixed Integer Programming model and a Constraint Programming model. We compare and discuss both model performances in a future chapter.

4.1 Notations

Set of periods

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

Set of workers

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

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 demands

$D = \{d_0, \ldots, d_n \mid n \in \mathbb{N}\}$	Set of demands
$d^w \in \mathbb{N}$	Required number of workers
$d^T \subseteq T$	Periods in which demand occurs
$d^c \in C$	Client of demand
$d^S \subseteq S$	List of required skills
$d^{s_i} \in d^S$	The ith skill of d^S
$d^{S^+} \subseteq S$	List of additional skills
$d^{S_i^+} \in d^{S^+}$	The ith skill of d^{S^+}
$d^P \in \{0, \ldots, d^w - 1\}$	List of positions

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

$$I_{ww} = \{(i, j) \in \mathbb{N} \times \mathbb{N} \mid w_i, w_j \in W, w_i \neq w_j\}$$

Set of worker - client incompatibilities

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

4.2 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.

4.2.1 Variables

To represent our problem in MIP, we will need three types of variables, one per resource.

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

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

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

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

This is in fact a binary Integer Programming model as every variables is a $\{0, 1\}$ integer.

As solution 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 f_{jkl}, f for *fictitious*. This variable is one, if and only if all the corresponding worker variables ($w_{ijkl}, \forall_i$) are equal to zero. The goal will be to minimize the number of fictitious variables assigned to one.

4.2.2 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 fictitious variable associated to this position to allow partial solutions.

$$\sum_{i \in W} w_{ijkl} + f_{jkl} = 1, \qquad \forall k \in D, j \in d_k^T, l \in d_k^P \tag{4.1}$$

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

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

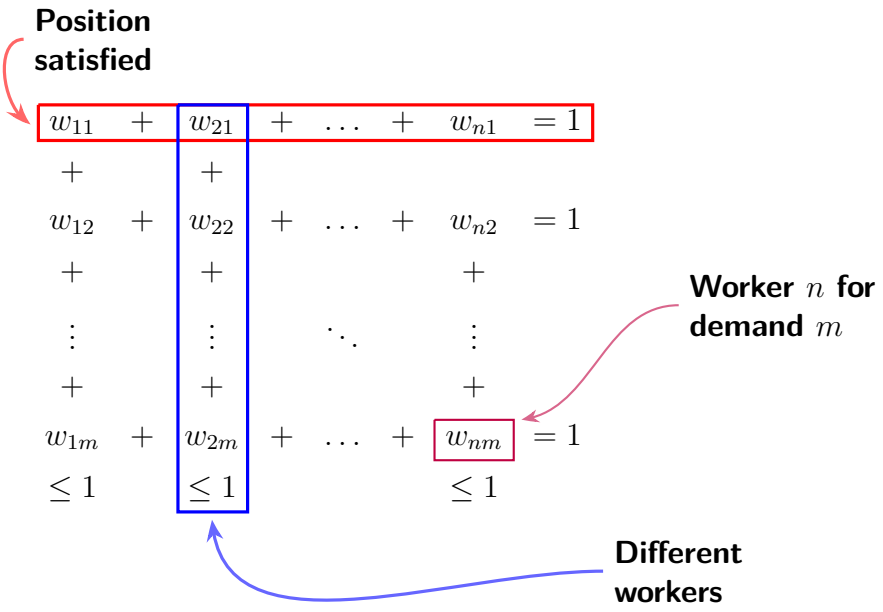

Figure 4.1: Visualization of (4.1) and (4.2).

Exclude impossible values

Some workers (demands) are not available (occurring) at one time period. We need to set the variables to 0 if this is the case.

$$t_j \notin d_k^T \implies \forall i, l \; w_{ijkl} = 0, \qquad \forall j \in T, k \in D \tag{4.3}$$

$$t_j \notin w_i^T \implies \forall k, l \; w_{ijkl} = 0, \qquad \forall j \in T, i \in W \tag{4.4}$$

$$t_j \notin d_k^T \implies \forall l \; s_{jkl} = 0, \qquad \forall j \in T, k \in D \tag{4.5}$$

Incompatibilities between workers and clients

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

$$d_k^c = c \implies \forall l \; w_{ijkl} = 0, \qquad \forall (i, c) \in I_{wc}, j \in T, k \in D \tag{4.6}$$

Incompatibilities between workers

For each incompatibility, at each demand, two workers cannot have both their variable assigned to one. We define that the sum over all the positions of the two

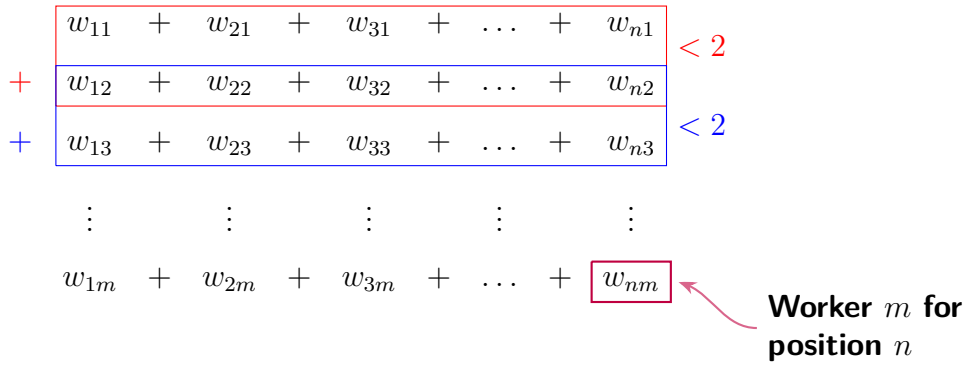

Figure 4.2: Example of (4.7) if (w_1, w_2) and (w_2, w_3) are incompatible pairs).

incompatible workers must be less than two.

$$\sum_{l \in d_k^P} w_{ajkl} + w_{bjkl} < 2, \qquad \forall (a,b) \in I_{ww}, j \in T, k \in D \tag{4.7}$$

Restrict skilled positions to skilled workers

These constraints ensure that no worker is working for a position at which he is not qualified to work. $W_{d_k^{s_l}}$ define the workers that satisfy skill(s) s_l of demand k.

$$w_{ijkl} = 0, \qquad \forall j \in T, k \in D, l \in d_k^P, i \in W \setminus W_{d_k^{s_l}} \tag{4.8}$$

Additional skills must be satisfied

$$\sum_{l \in d_k^P} w_{ijkl} \geq 1, \qquad \forall j \in T, k \in D, s \in d_k^{S^+}, i \in W_{d_k^{S^+}} \tag{4.9}$$

Binary constraints

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

$$w_{ijkl} \in \{0,1\}, \qquad \forall i \in W, j \in T, k \in D, l \in d_k^P \tag{4.10}$$
$$s_{jkl} \in \{0,1\}, \qquad \forall j \in T, k \in D, l \in d_k^P \tag{4.11}$$
$$m_{ij} \in \{0,1\}, \qquad \forall i \in M, j \in D \tag{4.12}$$
$$l_{ij} \in \{0,1\}, \qquad \forall i \in L, j \in D \tag{4.13}$$

4.2.3 Objective

$$\min \quad \delta_0 \sum_{k \in D} \sum_{l \in d_k^P} \sum_{i \in W} min(\sum_{j \in T} w_{ijkl}, 1) \tag{4.14a}$$

$$+ \delta_1 \sum_{r \in R} (max(r_{min} - occ_{r_w}, 0) + max(r_{max} - occ_{r_w}, 0)) \tag{4.14b}$$

$$+ \delta_2 \sum_{j \in T} \sum_{k \in D} \sum_{l \in d_k^P} f_{jkl} \tag{4.14c}$$

$$\text{with} \quad occ_i = \sum_{j \in T} \sum_{k \in D} \sum_{l \in d_j^P} w_{ijkl}, \quad \forall i \in W$$

$$\boldsymbol{\delta} = (\delta_0, \delta_1, \delta_2) = (1, 15, 100)$$

The objective function is stated in (4.14), it is a weighted sum split in multiple parts, it minimizes:

1. The number of different workers for every position between periods of that demand (4.14a), $min(\sum_{j \in T} w_{ijkl}, 1)$ is one if the worker i is working for that position at that time, 0 otherwise. Hence, the sum of that value for all worker will be equal to the number of worker for that shift.

2. The number of violations of working requirements (4.14b).

3. The number of *fictitious* worker assigned to demands (4.14c).

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

4.3 Constraint Programming Model

We now present our Constraint Programming model. This model contains some differences with the mathematical model described in Section 4.2. For example, a Constraint Programming model usually does not contain binary variables to refer to multiples 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 $w \in \{0, \ldots, n\}$.

4.3.1 Variables

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

$$w_{ijk} \in W \tag{4.15}$$

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

$$m_{ij} \in M \tag{4.16}$$
$$z_i \in Z \tag{4.17}$$

(4.16) is the j^{th} machine used for demand i while (4.17) is the zone used for demand i

As explained in the problem description and in the mathematical model section, we need to allow partial solutions where we have a fictitious worker that can work at any time. We will 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 will define $\sigma = -1$.

Some constraints are already satisfied by the modeling of the variables, like 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 $W_{d_j^{s_k}} \subseteq W$ be the subset of workers that satisfy the kth skill (set of skills) of demand d_j.

$$w_{ijk} \in W_{d_j^{s_k}} \cap \{w \mid t_i \in w^T\} \cap \{\sigma\}, \forall j \in D, i \in d_j^T, k \in d_j^P \tag{4.18}$$

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

4.3.2 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 fictitious worker σ in the domain of all worker variables and this value can appear as many times as possible, we will need a slight variant of the `alldifferent` called `alldifferent_except`. This constraint is the same as the original except that we can specify values that will be ignored from the constraint.

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

We will use this constraint to ignore the σ value from the propagation. Let $X_i = \{w_{ijk} \mid j \in D, k \in d_j^P\}$ be the set of all worker variables for period i. For each period, we define:

$$\texttt{alldifferent\_except}(X_i, \{\sigma\}), \forall i \in T \tag{4.19}$$

Incompatibilities between workers and clients

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

$$\texttt{not\_equal}(w_{ijk}, w), \forall (w, c) \in I_{wc}, \forall \ i, j, k \tag{4.20}$$

Incompatibilities between workers

A worker might have an incompatibility with a worker or a set of workers. This constraint cannot be solved with a series of `not_equal` like the worker - client incompatibilities. We will use a constraint called `negative_table`. This constraint is a type of *Table Constraints* [**?**] which in general can express either the allowed or forbidden combinations of values. In this case, `negative_table` expresses the forbidden combinations of values. The forbidden combinations of values is expressed by the table I_{ww}. We will add a `negative_table` constraint for each pair of workers for a demand at one given time. Let $P_{ij} = \{(w_{ijk}, w_{ijl}) \mid k \in d_j^P, l \in d_j^P, k \neq l\}$ be the permutations of worker variables for demand j at period i:

$$\texttt{negative\_table}(x, y, I_{ww}), \forall (x, y) \in P_{ij} \tag{4.21}$$

Additional skills must be satisfied

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

Let us define o_{ijs} the occurrences of workers at time i for demand j in W_s (the set of workers that satisfy skill s).

$$\texttt{gcc}(\{w_{ijk} \mid k \in d_j^P\}, o_{ijs}) \tag{4.22}$$
$$\texttt{sum}(o_{ijs}) \geq 1 \tag{4.23}$$
$$\text{with} \quad o_{ijs} \in \{0, 1\} \tag{4.24}$$
$$\forall j \in D, s \in d^{S^+}, i \in d_j^T \tag{4.25}$$

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

Minimizing violations of working requirements

A worker might have working requirements. He has to work a minimum (maximum) number of times, hence the total occurrences of this worker 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.

$$\texttt{softgcc}(X, [r_{1_{min}}, \ldots, r_{n_{min}}], [r_{1_{max}}, \ldots, r_{n_{max}}], v_r) \tag{4.26}$$

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

Minimizing the number of fictitious worker

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

$$\texttt{softgcc}(X, \sigma \rightarrow \sigma, [0], [0], v_\sigma) \tag{4.27}$$

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 (4.26) and v_σ (4.27) as our working requirements and fictitious 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 d_j^T. We use a constraint called `at_least_nvalue` to count this number. Let $W_{jk} = \{w_{ijk} \mid i \in d_j^T\}$ be the set of worker variables for demand j at position k across all time periods of that demand:

$$\texttt{at\_least\_nvalue}(W_{jk}, N_{jk}) \; \forall j \in D, k \in d_j^P \tag{4.28}$$

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.

The final objective will be a weighted-sum of all sub-objectives in the model. However, not all objectives are equal in values, 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. We define those penalties in (4.30).

$$\min \quad \delta_0(\sum_{j \in D} \sum_{k \in D_j^P} N_{jk}) + \delta_1 v_r + \delta_2 v_\sigma \tag{4.29}$$

$$\boldsymbol{\delta} = (\delta_0, \delta_1, \delta_2) = (1, 15, 100) \tag{4.30}$$

4.3.3 Search

We define a heuristic that allows: (i) the fictitious worker to never be selected if there is another value available in the domain of the variable; (ii) the worker chosen for a variable is the most available for that demand but is also the less available for other demands.

Variable Heuristic

The variable heuristic used for the search is a first-fail heuristic. In other words, the heuristic will chose the variable will the smallest domain. This allows variable with only one value alongside the fictitious value σ to always be selected first.

Most Available Value Heuristic

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

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 availabilities for a demand, they are ordering in respect to their remaining availabilities in other demands. This second ordering is important for smaller demands, the search will choose workers that are less likely to be needed in other demands.

It also never considers the fictitious worker σ for the worker value as it is not even considered for most available worker. This value heuristic will in practice find solutions much quicker than a traditional *min* value heuristic.

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 $w_1^T = \{0, 1, 2\}$, $w_2^T = \{0, 2\}$, $w_3^T = \{0, 1, 2, 3, 4\}$ and the demand occurrences as $d_1^T = \{0, 1, 2\}$, $d_2^T = \{0, 1, 2, 3, 4\}$. 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 will be as follow:

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

First we can see that w_2 will never be 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 guess 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 will always consider 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 more dynamic 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}: the number of times worker w already works for position p of demand d.

2. occ_w: the number of times worker w already works in any demands.

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

We now have three level 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.

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 permutations of those workers. For this, we use the `lexleq` constraint [**?**]. This constraints 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 occuring at the same time period.

$$\texttt{lexleq}(X, Y) \tag{4.31}$$

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 will ignore the permutation $x_1 = 2, x_2 = 1$ because it is symmetric to $x_1 = 1, x_2 = 2$. x_2 will be instead directly assigned to the value 3 and thus reducing the search space. Figure 4.3 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 use the Propagation Guided Relaxation [**?**] to relax our best solutions. We discuss and compare more relaxations options in a future chapter.

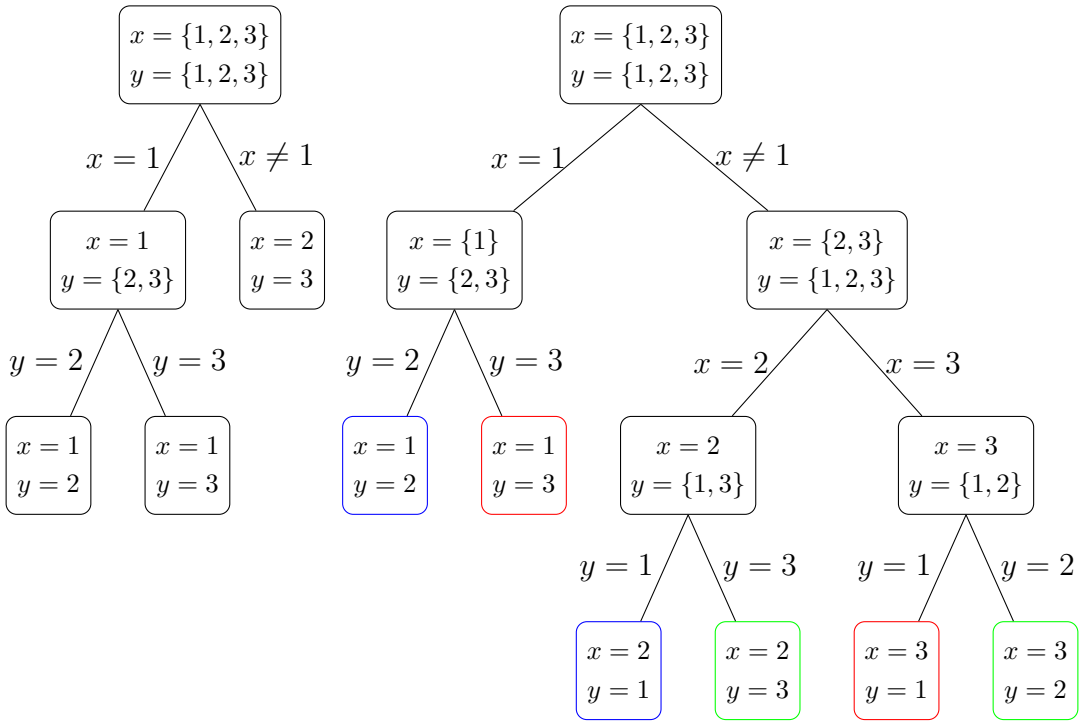

Figure 4.3: Search tree with symmetry breaking (left) and without (right)

Variable Objective LNS

Our problem uses a multi-objective (4.29) model. Let us define $o_1 = \min v_\sigma$, $o_2 = \min v_r$, $o_3 = \min \sum_{j \in D} \sum_{k \in D_j^P} N_{jk}$ and o_4 is the original weighted sum described in (4.29).

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, set o_2 to *Strong-Filtering*, o_1 to *Weak-Filtering* and others to *No-Filtering*.
3. Once o_2 is optimized, 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 will 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) as CP solver and *Gurobi Optimizer* (3.2.1) as MIP solver. The general implementation tries to keep the same API (Application Programming Interface) for the CP and MIP solvers with the only changes 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 returns the same output format. We created a JSON (JavaScript Object Notation) Schema [**?**] 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. 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    },
8    "required": ["name"]
```

```
  }
```

Listing 5.1: JSON Schema example

The data is 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).

```
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 [?] of the Gurobi Optimizer in Scala to create our implementation. The implementation did not change from the theoretical model presented in Section 4.2 as MIP solvers are less flexible in their modeling abilities than CP

solvers as we discuss later. The Gurobi solver comes with default parameters [?], it is advised to keep default parameters as changing them do not give much gain. Multiple different parameters were tested but as advised from the Gurobi website, no change were noticed.

5.4 Constraint Programming solver

The Constraint Programming implementation differ in some parts from the theoretical model presented in Section 4.3. In Constraint Programming, the constraint propagation takes the most time in the solving algorithm. Propagations might be unnecessary too strong for a model. This is what happened with our model and the use of `softgcc` constraints.

The minimization of fictitious workers described in (4.27) 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 count 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]
// sentinelViolations: CPIntVar
// Constants.SentinelWorker: Int = -1
add(
  gcc(workerVariables, Array(
      (Constants.SentinelWorker, sentinelViolations)
    )
  )
)
```

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

We followed the same idea for the working requirements minimization (4.26). `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])
```

```
2
3  // ...
4
5  val violations: Array[CPIntVar] = Array.fill(requirements.length)(null)
6
7  val occurrences = requirements
8    .map(_.worker)
9    .map(w => (w, CPIntVar(0, workers(w).availabilities.size)))
10
11 add(gcc(workerVariables, occurrences))
12
13 // For each requirement
14 for (i <- requirements.indices) {
15   val r = requirements(i)
16   violations(i) = maximum(Array(
17       occurrences(i)._2 -
           r.max.getOrElse(workers(r.worker).availabilities.size),
18       -occurrences(i)._2 + r.min.getOrElse(0),
19       CPIntVar(Set(0))
20     )
21   )
22 }
23
24 // workingRequirementsViolations: CPIntVar
25 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:

```
1  case class InstanceOptions(
2    t: Int,                              // Number of periods
3    clients: Int,                        // Number of clients
4    demands: Int,                        // Number of demands
5    workers: Int,                        // Number of workers
6    skills: Int,                         // Number of skills
7    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 represent 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 which default at 0.

```
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 add the demand periods to k random workers where k is the number of required workers for that demand. This allow having more solutions variety.

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.

Chapter 6

Experiments

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

6.1 Benchmark process

Our benchmark API takes an options structure (Listing 6.1). The arrays `T`, `D` and `W` determines the sizes of the instances. The benchmark runner will create instances with a combinations of those parameters. For example, with the values in Listing 6.1, the benchmark runner will create 6 instances with the sizes: $(5, 30, 100)$, $(5, 50, 100)$, $(5, 30, 200)$, $(5, 50, 200)$, $(5, 30, 300)$ and $(5, 50, 300)$ with the three values being (T, D, W).

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

Listing 6.1: 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 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.

Our benchmark runner has a function `run` that takes a name (e.g. solver name, serie name, etc), and a solving function which takes a generic model and returns a pair with the time spent in milliseconds and the objective value respectively.

```
class BenchmarkRunner(val options: BenchmarkOptions) {
  def run (
    name: String,
    solve: VillageOneModel => (Long, Int)
  ): (BenchmarkSerie, BenchmarkSerie)
}
```

Listing 6.2: Benchmark run function

This functions returns a pair of benchmark series: one serie for the time values and one serie for the objective values. The `BenchmarkSerie` class is simply a class that takes a name and a list of benchmark measurements (i.e. mean, standard deviation, min and max).

This implementation allows us to create a variety of benchmark by simply changing the `solve` function.

6.2 Constraint Programming

6.2.1 Comparison between heuristics

We talked in Section 4.3 about our custom heuristic called Most Available Heuristic. We now compare the performance of this heuristic with standard heuristic like the `max-value` heuristic. We also compare the variable heuristics used in addition to our aforementioned heuristic.

`mostavailable` & `max-value`

The `max-value` heuristic is a value heuristic that takes the maximum value in the domain of a selected variable during the search. It's one of the simplest value heuristic that can be implemented but unfortunately does not offer great performances.

Figure 6.1 shows the objective ratio between the implemented custom *Most Available* Heuristic and a standard *First Fail* variable heuristic with the `max-value` value heuristic after the first solution. The two heuristics were tested on 72 instances of various sizes from small to big instances. The performance profile shows a clear gain of about 2 to 3.4 for our custom heuristic.

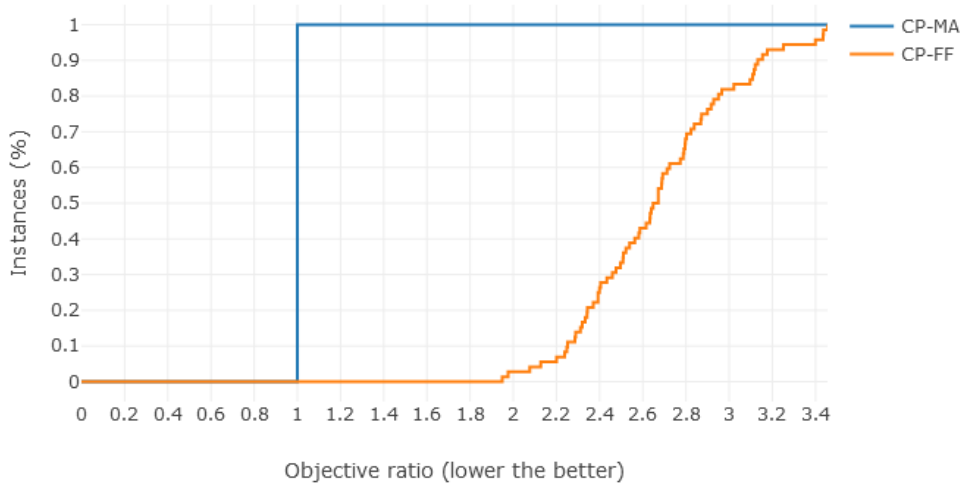

Figure 6.1: Most Available and First Fail heuristics [72 instances/first solution].

`mostavailable` & dynamic `mostavailable`

We also discussed in Section 4.3 an improvement based on the `mostavailable` static ordering where we ordered dynamically the most available worker at each value selection.

Figure 6.2 shows the performance profile of the dynamic & static most available heuristics. This benchmark was performed on 486 instances for a duration of 15 seconds each.

TODO: performance analysis

As our dynamic value heuristic outperforms every other tested heuristic, we will assume for the rest of this chapter that the value heuristic for the Constraint Programming solver is the dynamic `mostavailable`.

Max Degree & Min Size

In addition to our Most Available heuristic, we use a variation of the `maxdegree` heuristic. This heuristic is a first-fail variable heuristic that selects the most constrained unbound variable. However, as stated in our model in Section 4.3, skills are not represented with constraints but instead values are removed from the domain at initialization. To express skills as part of the max degree, we simply add the number of skills required by a variable to the degree of that variable. The

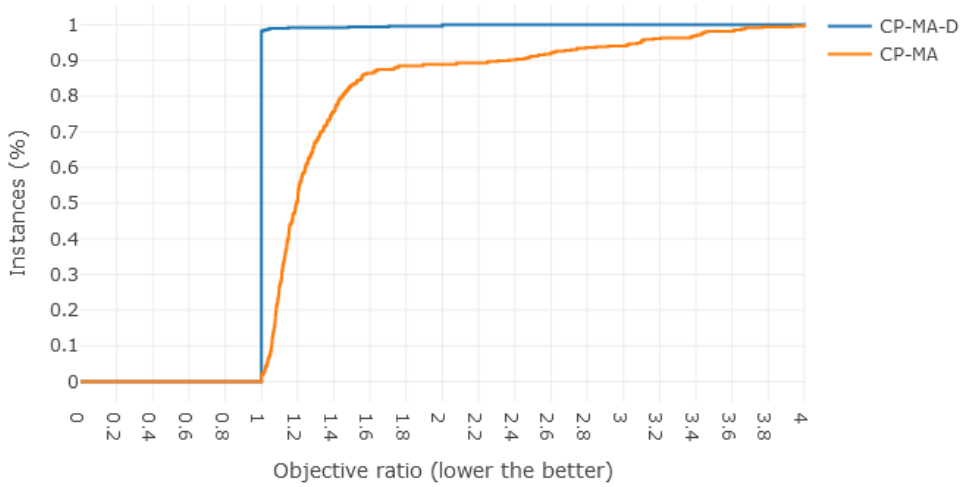

Figure 6.2: Most Available: static and dynamic [486 instances/15s].

`minsize` heuristic is also a first-fail variable heuristic which selects the variable with minimum domain size.

TODO FIGURE + PERF ANALYSIS

Max Degree & Conflict Ordering Search

Conflict Ordering Search (COS) [**?**] 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 that selects the variable which caused the last conflict first. COS was shown to be the most performant on scheduling problems.

TODO: FIGURE + PERF ANALYSIS

6.2.2 Comparison between searches

As stated in Chapter 3 and Section 4.3, LNS is used to expand the exploration of the search tree. We now compare our solver with the use of LNS and without. We also compare multiple relaxations method.

Standard Search & LNS

TODO: FIGURE + PERF ANALYSIS

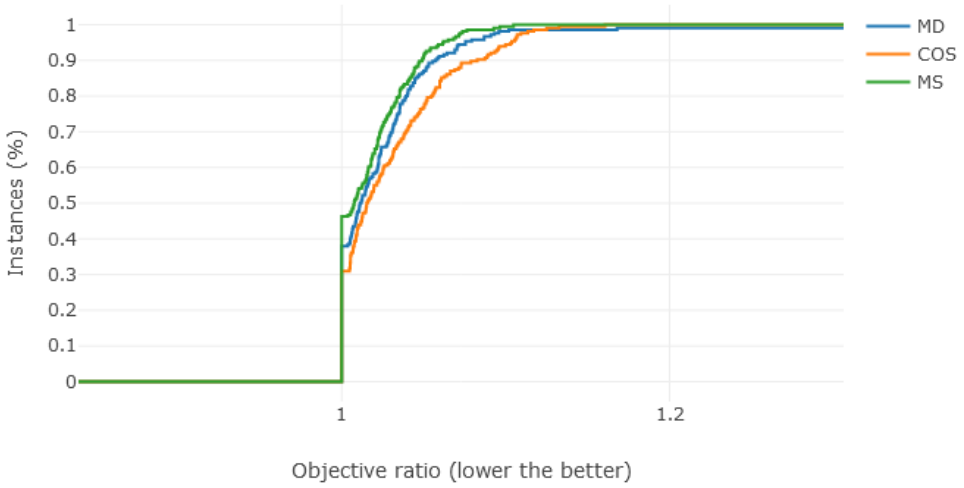

Figure 6.3: COS, Max Degree and Min Size heuristics [216 instances/30s].

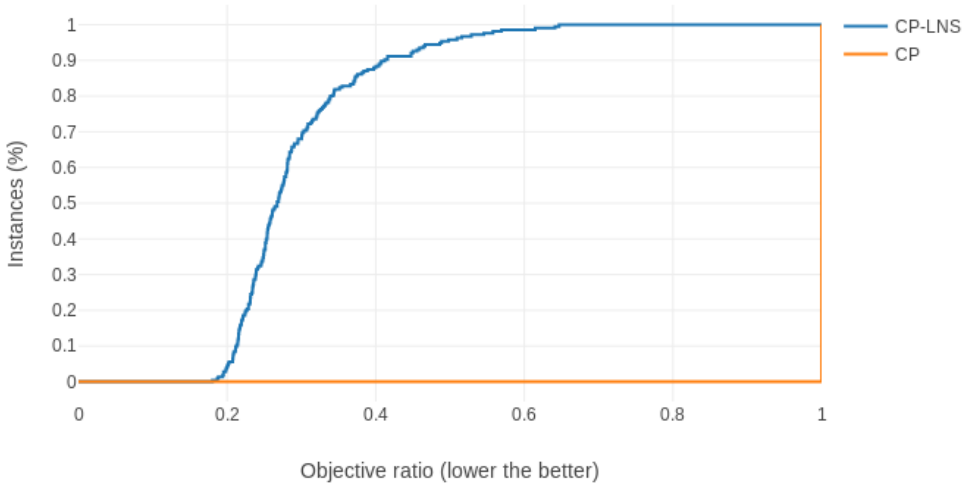

Figure 6.4: CP with and without Large Neighborhood Search [216 instances/30s].

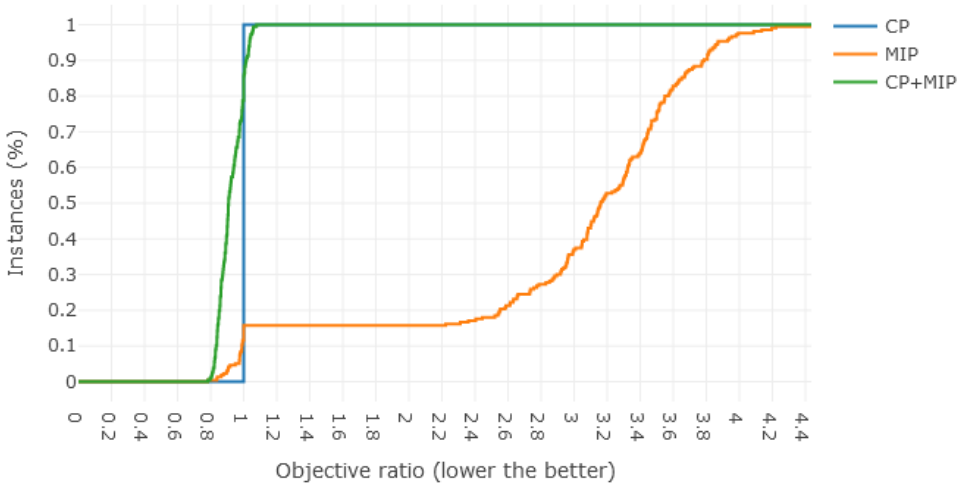

Figure 6.5: CP, MIP and CP+MIP solvers [216 instances/30s].

Random Relaxation & Propagation Based Relaxation

TODO: FIGURE + PERF ANALYSIS

6.3 Comparison between solvers

We now start by comparing different solvers together. Figure 6.5 shows a performance profile generated from 216 instances of various sizes. This benchmark was set to a time limit of 30s per instance. The baseline of this profile is the CP solver. We observe that the CP solver performs better than MIP in more than 80% of instances. However, we also tested the MIP solver by giving it a first solution obtained from CP, we can see that it slightly outperforms CP and MIP in 80% of instances.

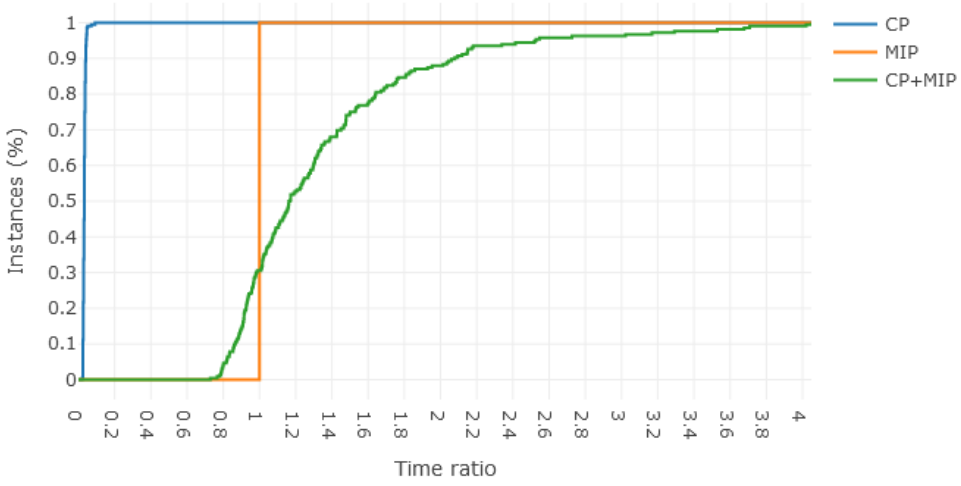

Figure 6.6: Time on first solution [216 instances/First solution].

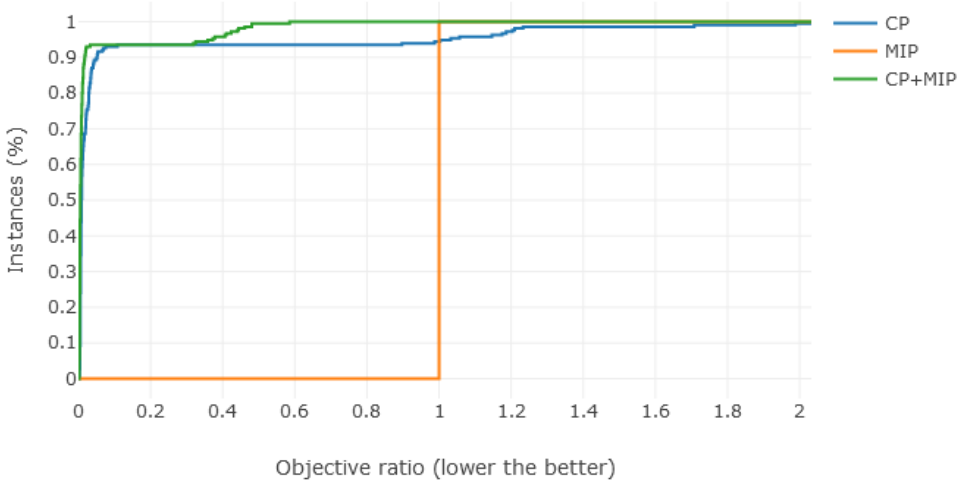

Figure 6.7: Objective on first solution [216 instances/First solution].

Chapter 7

Conclusion

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