Integrating Constraint Logic Programming and Operations Research Techniques for the Crew Rostering Problem

A. Caprara[1], F. Focacci[1], E. Lamma[1], P. Mello[2], M. Milano[1], P. Toth[1], D. Vigo[1]

(1) DEIS, Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
Tel: +39 51 6443033, Fax: +39 51 6443073
`filippo@deis33.deis.unibo.it`
`{acaprara,elamma,mmilano,ptoth,dvigo}@deis.unibo.it`

(2) Dipartimento di Ingegneria, Università di Ferrara
Via Saragat, Ferrara, Italy
`pmello@ing.unife.it`

Abstract

In this paper, we investigate the possibility of integrating Artificial Intelligence (AI) and Operations Research (OR) techniques for solving the Crew Rostering Problem (CRP). CRP calls for the optimal sequencing of a given set of duties into rosters satisfying a set of constraints. The optimality criterion requires the minimization of the number of crews needed to cover the duties. This kind of problem has been traditionally solved with OR techniques. In recent years, a new programming paradigm based on Logic Programming, named Constraint Logic Programming (CLP), has been successfully used for solving hard combinatorial optimization problems. CLP maintains all the advantages of Logic Programming such as declarativeness, non-determinism and an incremental style of programming, while overcoming its limitations, mainly due to the inefficiency in exploring the search space. CLP achieves good results on hard combinatorial optimization problems which, however, are not comparable with those achieved by OR approaches. Therefore, we integrate both techniques in order to design an effective heuristic algorithm for CRP which fully exploits the advantages of the two methodologies: on the one hand, we maintain the declarativeness of CLP, its ease of representing knowledge and its rapid prototyping; on the other hand, we inherit from OR some efficient procedures based on a mathematical approach to the problem. Finally, we compare the results we achieved by means of the integration with those obtained by a pure OR approach, showing that AI and OR techniques for hard combinatorial optimization problems can be effectively integrated.

Keywords: *Constraint Logic Programming, Operations Research, Crew Rostering Problem, Combinatorial Optimization Problems, Artificial Intelligence*

Contents

1 Introduction

Many real-life applications such as planning, scheduling, timetabling can be classified as hard combinatorial optimization problems, characterized by:

- a search space that grows exponentially with the size of the problem;
- many complex constraints;
- a high number of feasible solutions among which we have to choose the best one, following some domain-dependent optimization criteria.

The Operations Research (OR) approach for solving hard combinatorial optimization problems has been the only one for a long time. It is based on a mathematical representation of the problems, which are typically modeled as integer linear programs. Bounding procedures are derived by relaxing some constraints in the formulation, and the information obtained through the bound computation is used to effectively drive exact and heuristic algorithms, see, e.g., [23]. In recent years a new programming paradigm, Constraint Logic Programming (CLP), has been successfully used for solving hard combinatorial optimization problems such as scheduling, planning, sequencing and assignment problems, see [6, 7, 14]. CLP [16] is a powerful programming paradigm combining the advantages of Logic Programming [20] and the efficiency of constraint solving. CLP gave encouraging results that, however, are still not comparable with those achieved by OR approaches on large instances of hard problems.

In this paper we present a heuristic method for a particular hard combinatorial optimization problem: the *Crew Rostering Problem* (CRP). The CRP concerns the construction of *rosters* by sequencing *duties*, e.g., daily activities. Duties are characterized by starting and ending time and other attributes which determine their difficulty. Duties should be sequenced in accordance with a set of constraints. Typical constraints concern the difficulty of the whole roster and the time distance between two duties.

This application arose as a part of a wider problem of the Italian Railway Company (Ferrovie dello Stato SpA), whose goal concerns the optimization of the daily assignment of trips to employees. Characteristics of this problem are the very large number of trips to be handled (up to 5000), the large number of constraints, and the fact that trips are activities to be considered every day. Such a problem is usually solved in two phases. The first phase, called *crew scheduling*, is a short-term scheduling problem where trips are grouped into daily activities called *duties*. An example of a duty can be the sequence of trips Bologna → Parma, Parma → Milano, Milano → Bologna; it represents a single activity that has to be executed every day. In the second phase, namely the *crew rostering*, duties have to be grouped into rosters: a roster is a cyclic sequence of duties and represents a (monthly) working plan for a set of crews. The construction of rosters has to satisfy the constraints imposed by company regulations and labour laws.

CRP[1] has been widely investigated in the field of OR, [4, 1, 26, 10, 11, 3]. In this paper, we present a Crew Rostering application solved by integrating the CLP and OR approaches in order to fully exploit the advantages of both: on one hand, we maintain the declarativeness of CLP, its ease in representing knowledge and its rapid prototyping; on the other hand, we inherit from OR

[1]In 1994 the Italian Railway Company, jointly with AIRO (the Italian Operations Research Society), organized two competitions among Italian Universities, in order to promote the design of effective heuristic codes for the crew scheduling and rostering phases. On the second competition, called FARO (Ferrovie Airo Rostering Optimization), two prizes were to be assigned to the codes giving the best and the second best solution on three instances with up to 1,000 duties. The OR group of DEIS, University of Bologna, took part in the FARO competition and won the first prize giving the best solution for all the instances, following a pure OR approach, see [3].

some efficient procedures based on a mathematical approach to the problem, in particular a lower bound procedure taken from [3].

While the crew scheduling problem has been previously solved with Constraint Logic Programming (CLP) techniques (see [13]), to our knowledge, the crew rostering has never been solved either with CLP techniques or with a mixture of CLP and OR techniques. In this paper, we propose the integration of these techniques and compare the results obtained with a pure OR approach described in [3]. A pure CLP approach has not been fully investigated, but usually suffers from inefficiency when the dimensions of the problem increase. However, we will investigate a pure CLP approach by using a language that provides the so called global constraints, see [22]. Global constraints are relations involving a number of variables with specialized propagation techniques, which are very efficient for solving real life applications.

The paper is organized as follows. Section 2 presents a description of the problem, the constraints involved, the optimization criteria to be followed and a simple example. Section 3 outlines some OR approaches to combinatorial optimization problems. Section 4 gives a brief overview of Constraint Logic Programming, of possible applications solved and of tools available in this programming paradigm. Section 5 describes the CLP modeling of the problem in terms of variables, data structures and constraints. Section 6 presents the lower bounding procedure and the heuristic algorithms used. Section 7 presents the post-optimization procedure. Experimental results are shown and discussed in Section 8. Conclusions and a mention of future work follow.

2 The problem domain

The Crew Rostering Problem (CRP) is described as follows. Given a set of N duties to be performed each day of a given period, the problem is to create a set of rosters such that all the duties are covered. A *roster* is a set of n weeks; a *week* is a group of 6 consecutive days, the sixth of which is always a rest day[2]. Each duty has to be assigned to one and only one roster; to assign a duty means to state in which day of which week of the roster the duty will be executed. Rosters are preferably composed of 5 weeks (= 30 days); since rosters are considered cyclic groups, the last week precedes the first on the same roster. Once we have obtained the set of rosters, we can provide a working plan to each crew. In fact, since each duty is assigned once for every roster, and a duty should be executed every day, we need, for each roster, a number of crews equal to the number of days in the roster. For example, suppose a crew starts a roster whose first day is January the 1st, 1997. There must be another crew starting the same roster whose first day is January the 2nd, 1997 and so on. Therefore, we need a number of crews equal to the number of days in the roster, as depicted in Figure 1.

Each duty, identified by a number i, has input attributes which define its difficulty. These attributes are: *start time* s_i and *end time* f_i; *working time* w_i and *paid time* p_i (the paid time can be greater than the working time); all attributes representing time are expressed in minutes starting at midnight. Since a duty is never longer then 24 hours, s_i, f_i, w_i, and p_i are all integer numbers between 0 and 1440 (= 24 hours). Note that if a duty spans two days, since its duration is never longer than 24 hours, s_i is greater than f_i. For example, Duty 36 in the example of Section 2.3 starts at 12.15 p.m. (735 min.) and finishes at 9.08 a.m. (548 min.) the day after. Other attributes of a duty are *external rest*, *overnight*, *heavy overnight* and *long duty*, which are binary parameters defining the heaviness of a duty. Therefore, we can describe each duty as a tuple:

$$duty(i, s_i, f_i, w_i, p_i, Ext\_rest_i, Long_i, Overnight_i, Heavy\_over_i).$$

[2]Note that a week can start on whatever day and spans 6 days.

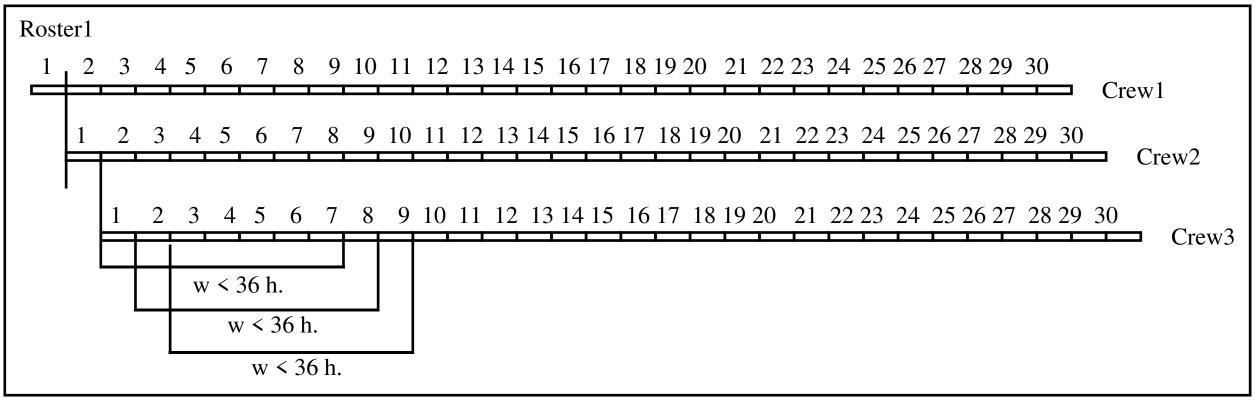

Figure 1: Cyclic feature of rosters

For example, the duty characterized by the following tuple:

$$duty(5, 245, 468, 223, 223, 0, 0, 1, 0).$$

is a (non *heavy*) overnight duty starting at 4:05 a.m., finishing at 7:48 a.m. with working time and paid time of 3 hours and 43 minutes.

A *weekly rest* is a time interval always containing the sixth day of a week, and lasting at least for 48 hours; if it contains at least two complete days (a complete day is a time interval 24 hours long, starting at midnight) it is called *double weekly rest*; otherwise it is called *single weekly rest*. A *technical interval* is a rest day between two duties in the same week.

In the following, we present the constraints involved in the problem, and the optimization criteria followed in searching for a solution.

2.1 Constraints

One of the characteristics of this problem is the large number of constraints to be considered. Constraints are mainly imposed by the Railway Company regulation and labour laws. We can identify four kinds of constraints: sequencing rules that concern the distance between two duties according to their difficulty; week and roster constraints that impose restrictions on the composition of weeks and rosters; finally, the last kind of constraints concerns the fact that rosters can be considered cyclic groups of days. We will further discuss the cyclic features of rosters in Section 6 by taking into account symmetries of the problem.

Sequencing rules

- The minimum rest between two duties is 18 hours, unless both duties are overnight, in which case the minimum rest is 22 hours.
- Each weekly rest must be at least 48 hours long. Note that this does not imply that it contains two complete days. For example, a weekly rest starting at 5 p.m. on Wednesday and finishing at 5 p.m. on Friday, is a single weekly rest because it does not contain two complete days starting and finishing at midnight.
- A heavy overnight duty cannot be followed by another heavy overnight duty.

- A pair of consecutive overnight duties cannot be followed by another duty starting earlier than 22 hours from the end of the second duty of the pair.
- If an overnight duty precedes a single weekly rest, then the first duty of the next week cannot start earlier than 6.30 a.m., unless the weekly rest includes two complete days.
- For each duty starting before 6 a.m. following a double weekly rest, the weekly rest must span at least three days.

Week constraints
For each week, there can be no more than two duties with external rest, no more than one long duty and no more than two overnight duties.

Roster constraints
For each roster, the average weekly rest time must be at least equal to 58 hours; the number of double weekly rests must be at least equal to 40% of the total number of weekly rests.

Cyclic constraints
For each cyclic group of 30 days within a roster there can be no more than 7 duties with external rest, and the total paid time of all the included duties cannot exceed 170 hours. If we build rosters of exactly 30 days, this constraint can be simplified: for each roster there can be no more than 7 duties with external rest, and the total paid time of all the included duties cannot exceed 170 hours. Another cyclic constraint is the following: for each cyclic group of 7 consecutive days within a roster, the total working time (w) of the included duties cannot exceed 36 hours, see Figure 1.

2.2 Problem objective

The objective is to find a feasible (i.e., one that satisfies all the constraints) set of rosters covering all the duties and minimizing the total number of weeks. In this way, we minimize the number of crews used, and therefore, the number of employees. Indeed, we need a number of crews equal to the overall number of days in the rosters in order to cover all the duties every day. Consider the example of Figure 1, where we depicted three instances of the same roster. We have a first crew that executes the roster starting in its first day, a second crew that executes the same roster starting the day after and so on. For these reason every week saved corresponds to a saving of six crews.

2.3 Example

In this section, we present an example of a problem instance and its solution. Suppose we have the following set of duties which are represented by tuples as explained in Section 2:

```
duty(4, 245, 468, 223, 223, 0, 0, 1, 0).
duty(5, 255, 594, 339, 339, 0, 0, 1, 0).
duty(8, 286, 683, 397, 397, 0, 0, 1, 0).
duty(10, 305, 751, 446, 446, 0, 0,0, 0).
duty(11, 316, 638, 322, 322, 0, 0, 0, 0).
duty(12, 331, 713, 382, 382, 0, 0, 0, 0).
duty(14, 358, 749, 391, 391, 0, 0, 0, 0).
duty(17, 401, 925, 524, 524, 0, 1, 0, 0).
duty(27, 580, 1056, 476, 476, 0, 0, 0, 0).
duty(34, 705, 1215, 510, 510, 0, 1, 0, 0).
duty(36, 735, 548, 648, 1253, 1, 0, 1, 0).
duty(38, 755, 518, 714, 1203, 1, 0, 1, 0).
duty(43, 825, 419, 559, 1034, 1, 0, 1, 0).
duty(56, 1020, 406, 370, 826, 1, 0, 1, 0).
duty(65, 1161, 1381, 220, 220, 0, 0, 0, 0).
duty(67, 1161, 8, 287, 287, 0, 0, 1, 0).
duty(69, 1302, 286, 424, 424, 0, 0, 1, 1).
```

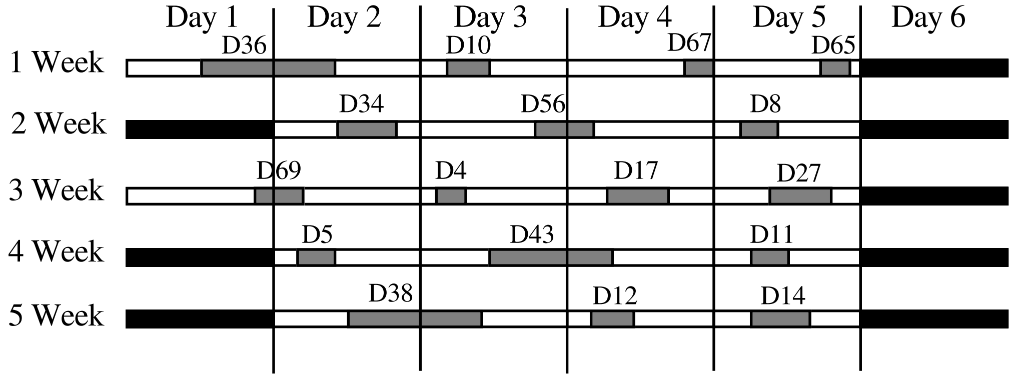

Figure 2: Solution of a CRP instance

A solution for this problem is depicted in Figure 2. We have 5 weeks (6 days each). Grey boxes represent duties. Each duty is numbered by its identifier. The black boxes represent complete days included in the weekly rests. There are two double weekly rests (between the first and the second week and between the fourth and the fifth week) out of five weekly rests. Note that the rest between the third and fourth weeks is not double since Duty 5 starts earlier than 6 a.m. The average weekly rest time is 59 hours and 53 minutes and the paid time is 120 hours and 32 minutes.

3 Operations Research Approaches

We now give a short description of some OR techniques which have been used for solving hard combinatorial optimization problems like Crew Rostering Problem. For a thorough introduction to such techniques see [23].

Combinatorial optimization is a branch of OR which studies problems of the form $min\{f(x) : x \in X\}$, where X is a finite *feasible solution set*, and $f(\cdot)$ is a real-valued *objective function* defined over X. Such problems cannot be solved trivially by enumeration in a reasonable computing time since X may have a huge cardinality.

In many cases, combinatorial optimization problems can be modeled as *integer linear programs*, i.e., as problems of the form:

$$\begin{aligned} & min\ c^T x \\ & Ax \geq b \\ & x \text{ integer} \end{aligned} \tag{1}$$

where the matrix A and the vectors b, c and x have appropriate dimensions. Notice that A and b define the feasible solution set, and the *cost vector* c defines the objective function. One of the most successful approaches for the solution of general integer linear programs is based on the use of *Lagrangian relaxation*, which is briefly outlined in the following. Let us suppose that constraint set $Ax \geq b$ can be conveniently subdivided into $A_1 x \geq b_1$ and $A_2 x \geq b_2$ in such a way that the removal of the first subset of constraints leads to an easily solvable problem. Given any non negative vector λ of multipliers for the constraints $A_1 x \geq b_1$, the associated Lagrangian relaxed problem is defined as:

$$\begin{aligned} & min\ c^T x - \lambda^T (A_1 x - b_1) \\ & A_2 x \geq b_2 \\ & x \text{ integer.} \end{aligned} \tag{2}$$

It is easy to see that the optimal solution value of (2) provides an optimistic estimate (*lower bound*) of the optimal solution value of (1). Indeed the relaxed problem has a wider feasible solution set and its objective function is not greater than the original one for all the feasible solutions to (1).

As an example, let us consider the following combinatorial optimization problem, known as the *Generalized Assignment Problem* (GAP). We are given n *items* and m *bins*. For each item-bin pair (i, j) the values p_{ij} and w_{ij} denote the *cost* incurred and the *weight* occupied if item i is assigned to bin j, respectively. Moreover, each bin j has a given weight capacity c_j. The problem is to assign each item to exactly one bin so as to minimize the total cost, without assigning to any bin a total weight greater than its capacity. A corresponding integer linear program is

$$min \sum_{i=1}^{n} \sum_{j=1}^{m} p_{ij} x_{ij} \tag{3}$$

$$\sum_{i=1}^{n} w_{ij} x_{ij} \leq c_j \quad j = 1, \ldots, m \tag{4}$$

$$\sum_{j=1}^{m} x_{ij} = 1 \quad i = 1, \ldots, n \tag{5}$$

$$x_{ij} \geq 0 \quad i = 1, \ldots, n; \ j = 1, \ldots, m \tag{6}$$

$$x_{ij} \quad \text{integer} \quad i = 1, \ldots, n; \ j = 1, \ldots, m \tag{7}$$

where each decision variable x_{ij} assumes value 1 if item i is assigned to bin j, and 0 otherwise. By relaxing in a Lagrangian fashion the capacity constraints (4), with Lagrangian multipliers $\lambda_1, \ldots, \lambda_m \geq 0$, we obtain

$$\begin{aligned} min \sum_{i=1}^{n} \sum_{j=1}^{m} p_{ij} x_{ij} + \sum_{j=1}^{m} \lambda_j \sum_{i=1}^{n} (w_{ij} x_{ij} - c_j) &= \\ min \sum_{i=1}^{n} \sum_{j=1}^{m} (p_{ij} + \lambda_j w_{ij}) x_{ij} - n \sum_{j=1}^{m} \lambda_j c_j & \end{aligned} \tag{8}$$

subject to (5), (6) and (7). The optimal solution of this relaxed problem is easily obtained by assigning each item i to the bin j such that the Lagrangian cost $p_{ij} + \lambda_j w_{ij}$ is a minimum.

The lower bounds obtained by Lagrangian relaxation can be used to drive heuristic and exact algorithms. Clearly one would like the lower bound to be as tight as possible. This is achieved by solving the so-called *Lagrangian dual problem* associated with (2), calling for the determination of the vector λ^* which maximizes the optimal solution value of (2). The Lagrangian dual problem is typically solved by iterative techniques such as subgradient optimization; however, in some particular cases (e.g., when the number of relaxed constraints is small) ad-hoc techniques are used.

Almost-optimal heuristic solutions to (1) can often be obtained by using the information given by the Lagrangian relaxed problem. For example, a very simple greedy heuristic algorithm can choose, at each iteration, for a variable x_j the value $\bar{x}_j$ which minimizes the lower bound increase obtained by imposing the further constraint $x_j = \bar{x}_j$. Also, heuristic algorithms can make an effective use of the so-called *Lagrangian cost vector*, $c - \lambda^T A_1$, which typically gives more reliable information than the original cost vector c. Examples of successful application of Lagrangian relaxation to combinatorial optimization problems can be found in [23].

4 Constraint Logic Programming: an Overview

CLP [15, 16] is a class of programming languages combining the advantages of Logic Programming (LP) ([18, 20]) and the efficiency of constraint solving. CLP "inherits" from LP the declarative semantics which make it possible to develop concise and easily modifiable programs in a relatively short time when compared to imperative languages. On the other hand, the inefficiency of LP in dealing with real-life applications makes LP unsuitable for practical use. Therefore, CLP has been proposed as an extension of LP in order to overcome its limitations. The CLP constraint solving over a computational domain replaces the weaker LP unification mechanism. This makes it possible to *a-priori* reduce the search space, thus increasing the computational efficiency. Thanks to the active use of constraints, the *a-priori* pruning allows us to delete portions of the search space that do not lead to a solution. The main idea behind this approach is to prevent failures instead of recovering from already happened failures, by means of expensive backtrackings.

CLP has been defined by Jaffar and Lassez as a general scheme, [15]. This scheme can be specialized on a particular computational domain by defining the operations and constraints. Each specialization is a CLP language. Several languages have been proposed such as CLP on real numbers, rational numbers, integers, boolean and finite domains. In this paper, we are interested in CLP on finite domains, hereinafter referred to as CLP(FD).

In CLP(FD) languages, variables range on finite domains of integers. The variable's domains represent possible values that variables can assume during the computation. For example, $X :: [1..10]$ states that variable X can assume one of the integer values from 1 to 10, $Y :: [3, 5, 9]$ states that variable Y is either 3 or 5 or 9. Variables are linked by constraints that can be either mathematical constraints, such as $X > Y$, $X < Y$, $X = Y$, $X \leq Y$, $X \geq Y$, $X \neq Y$, or symbolic constraints. Symbolic constraints are more powerful constraints (which can also be defined by the user) with complex propagation mechanisms. A typical symbolic constraint is `atmost(Nb,[X`$_1$`, ...X`$_n$`],Val)`, available in most CLP(FD) solvers like CHIP [8] and ECL^{i}PS^e [9]. Declaratively, the constraint `atmost(Nb,[X`$_1$`, ...X`$_n$`],Val)` holds if and only if at most `Nb` elements `X`$_i$ are equal to `Val`. In the constraint, all the `X`$_i$ are domain variables. We will see, in Section 5.2, a possible use of these constraints in the crew rostering application.

In CLP(FD) variables' domains are restricted during the computation according to the constraints, thus achieving the *a-priori* pruning of the search space. The domain reduction is performed in CLP(FD) solvers thanks to the arc-consistency propagation algorithm [21]. A constraint is arc-consistent if and only if for each value of the first variable there exists a value of the second variable which is consistent with the constraint and vice versa. For example, given two variables $X :: [1..5]$ and $Y :: [1..5]$ linked by the constraint $X < Y$, the arc-consistency algorithm reduces the domains to $X :: [1..4]$ and $Y :: [2..5]$. In fact, there does not exist a value for Y compatible with the value $X = 5$, and, vice versa, there does not exist a value for X compatible with the value $Y = 1$. This algorithm is not complete in the sense that, at the end of the propagation, some domains can contain values that are incompatible with constraints. A typical example of the incomplete behavior is the following: we have three variables $X :: [1, 2]$, $Y :: [1, 2]$, $Z :: [1, 2]$ linked by the following constraints $X \neq Y$, $X \neq Z$ and $Y \neq Z$. In this case constraints are inconsistent but the arc-consistency algorithm does not detect the inconsistency. However, arc-consistency turns out to be a good trade off between computational complexity (a complete algorithm would have an exponential cost in the number of variables involved) and pruning of the search space.

The structure of a CLP(FD) program is the following:

```
solve(List_variables):-
      domain_definition(List_variables),
```

```
      constraint(List_variables),
      labeling(List_variables).
```

The solve parameter `List_variables` is a list of variables representing the problem solution. In the `domain_definition` step, variables are defined with the corresponding domains. In the `constraint` step, we state binary (n-ary) constraints. The underlying constraint solver propagates the constraints by using arc-consistency, thus reducing variable domain values. At the end of the propagation, we have either found a solution (each variable is instantiated to a feasible value) or we go on with a labeling step where a "guess" is performed by assigning a value to one of the free variables. This labeling process is iterated until all variables are successfully instantiated, i.e., a solution is found. During this step, constraints are always taken into account and propagated in order to prune, as much as possible, the search space. The labeling step, for real-life problems, concerns also heuristic rules which improve the search efficiency. These heuristics concern, for example, the variable to be instantiated next, the next value to be assigned and so on.

Let us see an example that shows the behaviour of a simple CLP(FD) program.

```
solve([X,Y,Z]):-
      X::1..10, Y::5..15, Z::4..9,
      X#>Y, Y##Z,
      labeling([X,Y,Z]).

labeling([FirstVar| Others]):-
     indomain(FirstVar),
     labeling(Others).
```

The syntax used is that of ECL^{i}PS^e [9]. The predicate symbol `::` associates a variable `Var` with a domain `Domain` in the following way: `Var :: Domain`. Binary constraints are always represented with an additional `#` symbol denoting the propagation. For example, the constraint `#>` is used to actively prune the variable domains linked by the constraint, while the symbol $>$ denotes only a test. The symbol `##` means $\neq$. In the first step, we define variables `X`, `Y` and `Z` and the corresponding (finite) domains. In the second step, we link the above-mentioned variables by means of constraints: `X` greater than `Y` and `Y` different from `Z`. The underlying constraint solver propagates the constraints by using an arc-consistency algorithm and reduces the domains accordingly. The resulting domains after the propagation are: `X::[6..10]`, `Y ::[5..9]` and `Z::[4..9]`. Since the binary constraints are not yet solved, they are delayed and will be awakened as soon as one domain changes. The labeling step starts with the reduced domains and performs a guess. Variable `X` is instantiated by means of the user-defined predicate `indomain/1` to one of the remaining values of the domain, i.e., `X=6`. Binary constraints are awakened since a domain is changed. The propagation of constraints (arc-consistency) causes the reduction of the domain of variable `Y` to the value 5 (the only one which is consistent with the value 6 of `X`). The domain of `Z` is reduced to [4,6,7,8,9]. The labelling step goes on with the instantiation of `Z` (`Y` is already instantiated), thus finding a solution. Other solutions can be achieved in backtracking (`indomain/1` has several choice points, one for each domain value).

In many cases, an optimization criterion must be satisfied, i.e., we have to find the best of the feasible solutions. Therefore, many CLP languages provide *branch and bound* procedures for minimization or maximization. Alternatively, a post-optimization procedure that locally improves the quality of the found solution can be implemented.

4.1 CLP Applications and Tools

In this section, we will give a sketch of real-life CLP applications and CLP tools. Recent literature (see [6, 7, 14]) has shown that the CLP(FD) approach is very useful in solving Constraint Satisfaction Problems (CSPs). A CSP can be defined in the following way: assume the existence of a set of variables $(X_1, X_2, \ldots, X_n)$ whose values range on finite domains $(D_1, D_2, \ldots, D_n)$, and a set of constraints. A constraint $c(X_{i1}, X_{i2}, \ldots, X_{ik})$ on k variables is a subset of the cartesian product $D_{i1} \times D_{i2} \times \ldots \times D_{ik}$ that represents which values of each variable are compatible. A solution of a CSP is an assignment of values to all variables, which satisfies all the constraints. The class of CSP involves different application areas such as planning, scheduling, resource allocation, assignment, placement and logistics [2, 6, 7, 19, 14]. They are characterized by common features. For instance, there is no efficient way of solving them as they often belong to the NP-complete class of problems. Therefore, different strategies and heuristic rules have been tested. Solutions to these problems based on imperative languages are usually very long to develop, expensive to maintain and difficult to modify. The requirements of this class of applications are subject to frequent modifications so the adopted programming language should support ease of code modification. Moreover, the changes should be incremental.

CLP(FD) has been proved to be an effective and flexible tool that meets these requirements by adopting a general step by step methodology to solve CSPs. In particular, we can identify the following steps:

- model the problem: find out constraints, heuristics, strategies and rules;
- constrain the problem;
- solve the problem using propagation of constraints and backtracking;
- evaluate the solution according to optimization criteria.

Quite recently an increasing number of CLP tools have been developed. The most important are CHIP [8], CLP(R) [17], Prolog III [5], BNR Prolog, ECL^{i}PS^e [9] and ILOG [25]. These tools contain constraint solvers in specified domains. The most common constraint solvers involve:

- rational and real numbers based mainly on simplex and gaussian elimination;
- boolean algebra based on variable elimination on a graph representation;
- finite domains based on local consistency techniques and bound propagation.

Our application has been developed by using a very flexible tool, ECL^{i}PS^e. ECL^{i}PS^e is a logic programming language that supplies a finite domain constraint solver. Mathematical constraints are embedded in the solver as well as some symbolic constraints. In addition, ECL^{i}PS^e makes it possible to implement user-defined symbolic constraints.

ECL^{i}PS^e also interfaces the C code. We used this capability in order to use the code of the OR lower bounding procedure explained in Section 6.1.

In the following, we present the application of CLP to the CRP. For the sake of readibility, we follow the structure of the program presented in Section 4. Our program is much more complex but the above-mentioned structure can be easily identified. Therefore, we present in Section 5 the first two steps, i.e., domain variable definition and constraint posting. In Section 6 we present the labeling step by focusing on the lower bound procedure and on heuristic rules used for reducing the complexity of the problem. In Section 7, we discuss the post-optimization procedure that allows us to find, among feasible solutions, the best one.

5 Modelling the problem

In this section, we show the modelling of the problem used for obtaining a heuristic solution to the Crew Rostering application. We have followed a general framework for modeling problems in CLP(FD), as explained in Section 4. First of all, we have defined variables and the corresponding domains. In the second step, the preprocessing phase takes into account some constraints which make it possible to prune variable domains before the search starts. Then, we state constraints on variables which can be either mathematical or symbolic. In this section we present the first two steps of the program presented in Section 4 with the addition of a preprocessing step to increase the efficiency of the whole code.

Modelling the problem is one of the most important steps in the system project since it strongly influences the efficiency of the whole architecture. For example, during this step, we have to evaluate whether to use redundant variables or constraints which, in several cases, improve system performance. Redundant variables can be useful in order to state constraints in a natural way. We will see an example in the following section. Since finite domain constraint solvers are incomplete, redundant constraints can reduce the search space, thus increasing the efficiency of the algorithm.

In order to improve algorithm performance, by further reducing the search space, we impose the condition that each roster (with the possible exception of the last one) is five weeks long.

5.1 Domain variables

The structure used to identify the temporal location of a duty is the following:

$$duty\_location(N, [Day, Week, Rost, Time], Prev\_duty, Next)$$

where N uniquely identifies the duty. Day, $Week$, $Rost$ are domain variables representing respectively the day, the week and the roster assigned to the duty, while $Time$ is a redundant domain variable representing, in a uniform way, the temporal position of the duty. These variables are linked by the following constraints[3]:

$$Time = 6Week + Day$$

$$Rost = (Week - 1)\ div\ 5 + 1$$

where variable Day ranges on an initial domain [1..5], i.e., the working days of the week. Variable $Week$ initially ranges on $1..MaxWeek$, where $MaxWeek$ is a reasonably large number, say the number of duties divided by two. $Rost$ initially ranges on $1..(MaxWeek - 1)\ div\ 5 + 1$ because, for efficiency, rosters are considered to be 5 weeks long. Finally, variable $Time$ ranges from 7 to $6\,MaxWeek{+}5$. This is an implementation choice. It is worth noting that this representation allows us to express all the constraints in the most natural way. However, it is a redundant representation since, for example, the variable $Time$ gives complete information about when the duty has to be executed. Therefore, given this variable, all the others (Day, $Week$ and $Rost$) are known by means of the above-mentioned relations. This redundancy introduces an overhead due to the propagation of the above constraints. However, on one hand it simplifies the problem representation and the constraint handling and, on the other hand, redundant constraints can be useful in incomplete constraint solvers such as the finite domain solver we used.

The domain variable $Next$ identifies the duty that follows[4]; during the `domain_definition` step, its domain is initialized by containing all possible duties plus values -2, -1 and 0 which represent, respectively, a double or single weekly rest or a technical interval.

[3] The operator div performs the integer division.

[4] Note that the duty that follows is not necessarily assigned to the next day.

5.2 Constraint definitions

Constraints involved in the problem have been defined in Section 2.1. In this section, we formalize these constraints and present some implementation details. Constraints can be divided into three groups according to the way they are handled during the computation:

- Preprocessing constraints;
- Local constraints;
- Research constraints.

5.2.1 Preprocessing Constraints

Some constraints (or relaxations of some constraints) can be effectively imposed by simply reducing the domain of variables during a preprocessing phase. These constraints are usually taken into account during the preprocessing phase. When this phase ends, some of them do not play any role during the search, while other constraints must be taken into account in order to generate feasible solutions. The constraints imposed during this step are the following:
a. The minimum rest between two duties is 18 hours, unless both duties are overnight. In this case the minimum rest is 22 hours. The formalization for these constraints is:

$$(f_i > 24h - 18h) \wedge (s_j < f_i + 18h - 24h) \Rightarrow Next_i \neq j$$

$$Overnight_i \wedge Overnight_j \wedge (f_i > 24h - 22h) \wedge (s_j < f_i + 22h - 24h) \Rightarrow Next_i \neq j$$

These constraints reduce variables $Next_i$ for each duty. In fact, consider the example in Section 2.3. At the beginning, for each duty the domain variable $Next_i$ contains all the duty identifiers. For instance, consider Duties 17 and 14. For these duties the first constraint can be instantiated as follows:

$$(f_{17} > 1440 - 1080 = 360) \wedge (s_{14} < f_{17} + 1080 - 1440 = 565) \Rightarrow Next_{17} \neq 14$$

The value 14 is therefore removed from the domain of $Next_{17}$. The second constraint concerns overnight duties. Consider Duties 4 and 5 of the example in Section 2.3. The instantiation of the second constraint is:

$$Overnight_4 \wedge Overnight_5 \wedge (f_4 > 120) \wedge (s_5 < 348) \Rightarrow Next_4 \neq 5$$

b. Each heavy overnight duty cannot be followed by another heavy overnight duty:

$$HeavyOver_i \wedge HeavyOver_j \Rightarrow Next_i \neq j$$

This constraint is similar to the one presented above. In the simple example of Section 2.3, however, we have just one heavy overnight duty, so the constraint is satisfied.
c. No more than one long duty per week is allowed

$$Long_i \wedge Long_j \Rightarrow Next_i \neq j$$

For example, duties 17 and 34 cannot be subsequent duties. Therefore, $Next_{17} \neq 34$ and $Next_{34} \neq 17$. This constraint will be subsumed by a tighter local constraint, see section 5.2.2. However, redundant constraints in incomplete constraint solvers can be useful in order to *a priori* prune the search space and increase the efficiency of the search process.

d. The following constraint is not explicitly defined by the problem but is implicit in the fact that a week is a sequence of 6 days, the sixth of which is always a rest.

$$(f_i < s_i) \Rightarrow Day_i \neq 5$$

This constraint holds for duties 36, 38, 43, 56, 67, 69. In fact, in the solution depicted in Figure 2, these duties are never scheduled on the fifth day of the week.

5.2.2 Local Constraints

Local constraints are stated and propagated within a single roster. Most of them can be easily formalized by using the symbolic constraint `atmost(Nb,[X`$_1$`, ...X`$_n$`],Val)`, available in most CLP(FD) solvers like CHIP [8] and ECL^{i}PS^e [9]. Declaratively, as explained in Section 4, the constraint

`atmost(Nb,[X`$_1$`, ...X`$_n$`],Val)`

holds if and only if at most `Nb` elements `X`$_i$ are equal to `Val`. For example, the following constraints are implemented as shown:

a. For each week identified by M, there can be no more than 2 duties with external rest, no more than 1 long duty and no more than 2 overnight duties:

$$atmost(2, WEstRest, M)$$

$$atmost(1, WLong, M)$$

$$atmost(2, WOver, M)$$

where $WEstRest$, $WLong$ and $WOver$ are lists containing the domain variables for $Week$ of duties with external rest, long duties and overnight duties, respectively. Variable M represents a number which identifies the week under consideration. For the example of Section 2.3, the lists $WestRest$, $WLong$ and $WOver$ contain respectively:

$$WestRest = [Week_{36}, Week_{38}, Week_{43}, Week_{56}]$$

$$WLong = [Week_{17}, Week_{34}]$$

$$WOver = [Week_4, Week_5, Week_8, Week_{36}, Week_{38}, Week_{43}, Week_{56}, Week_{67}, Week_{69}]$$

b. A similar constraint concerns the roster, and states that no more than 7 duties with external rest can be performed within a roster of 30 days. Therefore, if $RostEstRest$ is a list of domain variables of duties with external rest and R a number identifying the roster, the constraint can be formalized as:

$$atmost(7, RostEstRest, R)$$

where $RostEstRest$ in the example is:

$$RostEstRest = [Rost_{36}, Rost_{38}, Rost_{43}, Rost_{56}]$$

c. Within a roster of 30 days, the total paid time of all the included duties cannot exceed 170 hours

$$\sum_{i | Rost_i = N} p_i \leq 170h$$

The original constraints *b.* and *c.* are defined for each cyclic group of 30 days within a roster, imposing that the total number of duties with external rest is at most 7, and that the total paid

time of all the included duties cannot exceed 170 hours. However, if we consider rosters of 30 days (in our heuristic algorithm this happens in every roster but possibly the last one) the constraint can be reduced to the simpler form shown above. For the last roster these constraint are propagated in the complex form by evaluating each cyclic group of 30 days.
d. Every weekly rest must be at least 48 hours long.

$$\forall_{i|i=1..MaxWeek}\ r_i \geq 48h$$

where r_i is the time interval between the end of the last duty in a week and the start of the first duty in the next week. $MaxWeek$ is the overall number of weeks.
e. A pair of consecutive overnight duties cannot be followed by another duty starting earlier than 22 hours from the end of the second duty of the couple:

$$Overnight_i \wedge Overnight_{Next_i} \wedge f_{Next_i} > 24h - 22h \wedge s_j < f_{Next_i} + 22h - 24h \Rightarrow Next_{Next_i} \neq j$$

f. The number of double weekly rests must be at least equal to 40% of the total numebr of weekly rests. Let DWR be the number of double weekly rests and, SWR the number of single weekly rests, the constraint is expressed as follows:

$$DWR >= 0.4(SWR + DWR)$$

We have two extra constraints, the last two sequencing rules in section 2.1, which are stated in the system but are usually subsumed by other stronger constraints.

5.2.3 Research Constraints

Research constraints are usually stated dynamically during the labeling phase.
a. For each cyclic group of 7 consecutive days within a roster the total working time of the included duties cannot exceed 36 hours[5]

$$\sum_{i|Day_i=k..k+6} w_i \leq 36h$$

b. For each roster the average weekly rest time must be at least equal to 58 hours

$$\frac{\sum_{i|Rost_i=N} r_i}{NWeek} \geq 58h$$

Note that in each roster we have $NWeek = 5$ except for the last one where $NWeek \leq 5$.

6 Heuristics and Search

6.1 Lower bounding procedure

Simple lower bounds on the number of weeks in an optimal CRP solution can easily be obtained by considering each of the constraints imposing a limit either on the total number of duties with a given characteristic, or on the total working and paid time in a week and in a (cyclic) group of 30 consecutive days in a roster, respectively. In particular, a valid lower bound, lb_1, is defined as the maximum among the following values:

[5]This formalization is quite different from the one used in the implementation since Day is a domain variable ranging between 1 and 5. However, the constraint considers 7 consecutive days.

- number of long duties;
- $\lceil$(number of overnight duties)/2$\rceil$;
- $\left\lceil 5 \cdot \sum_{i=1}^{N} p_i/(170 \cdot 60) \right\rceil$;
- $\lceil 5 \cdot$ (number of duties with external rest)/7$\rceil$;
- $\left\lceil \sum_{i=1}^{N} w_i/(36 \cdot 60) \right\rceil$.

We now describe a more effective bounding procedure for CRP, proposed in [3], where all the rules for sequencing two consecutive duties within a roster are taken into account. The other constraints considered in the relaxation impose the condition that the total number of weekly rests is equal to the total number of weeks composing the rosters, and that the total number of double weekly rests is at least equal to 0.4 times the total number of weekly rests. For the lower bound, we do not require that the sixth day of each week is included in a weekly rest, neither that each roster is composed by an integer number of weeks. We next give a description of the resulting relaxed problem (called RP, for short, in the sequel) formulated as a graph theory problem.

We are given a complete directed multigraph $G = (V, A)$, where $V = \{1, \ldots, N\}$ is the set of vertices, each associated with a duty. The arcs represent the consecutive sequencing of duty pairs within a roster. As described in Section 2.1, two duties can be sequenced either directly, or with a single or double weekly rest between them. Therefore, the arc set A contains arcs of three different types and can be partitioned into three subsets, A_1, A_2 and A_3. For each pair of vertices $i, j \in V$ we have an arc $(i, j) \in A_1$, whose length c^1_{ij} is the minimum time (in minutes) between the start of duty i and the start of duty j when they are sequenced directly, i.e., in the same week. Similarly we have an arc $(i, j) \in A_2$ (resp., $(i, j) \in A_3$) whose length c^2_{ij} (resp., c^3_{ij}) is the minimum time (in minutes) between the start of duty i and the start of duty j when a single (resp., double) weekly rest is imposed between them. G has no loop, therefore we let $c^1_{ii} = c^2_{ii} = c^3_{ii} = +\infty$ for each $i \in V$. In the sequel arcs belonging to A_1 are also called *direct arcs*, while arcs belonging to A_2 (resp., A_3) are called *single rest arcs* (resp., *double rest arcs*). Matrices c^1, c^2 and c^3 can be easily computed from the input data, following the sequencing rules. Notice that, by definition of the arc lengths, for a given pair $i, j \in V$ the values $c^1_{ij}, c^2_{ij}, c^3_{ij}$ differ by integer multiples of 1440. Let $\alpha = 6 \cdot 1440$ be the number of minutes in a week.

In G, each cycle corresponds to a roster, the length of the cycle being the time required to perform the corresponding duties. Problem RP then calls for determination of a minimum-length set of disjoint cycles of G satisfying the following constraints:

- each vertex of G is covered by exactly one cycle;
- the total number of single or double rest arcs in the cycles has to be at least equal to the total length of the cycles, expressed in weeks;
- the total number of double rest arcs in the cycles has to be at least equal to 0.4 times the total number of single or double rest arcs.

Problem RP can also be formulated as the following integer linear program. For each arc $(i, j) \in A_l$, $i, j = 1, \ldots, N$, $l = 1, 2, 3$, we introduce a binary variable x^l_{ij}, equal to 1 if arc $(i, j) \in A_l$ is in the optimal solution, and 0 otherwise. Moreover, we introduce an integer variable y, representing the minimum number of single or double rest arcs in the solution, and an integer variable z, representing the minimum number of double rest arcs in the solution. The model then reads

$$v(\mathrm{RP}) = min \sum_{i=1}^{N} \sum_{j=1}^{N} (c_{ij}^1 x_{ij}^1 + c_{ij}^2 x_{ij}^2 + c_{ij}^3 x_{ij}^3) \tag{9}$$

subject to

$$\sum_{i=1}^{N} (x_{ij}^1 + x_{ij}^2 + x_{ij}^3) = 1, \qquad j = 1, \ldots, N \tag{10}$$

$$\sum_{j=1}^{N} (x_{ij}^1 + x_{ij}^2 + x_{ij}^3) = 1, \qquad i = 1, \ldots, N \tag{11}$$

$$y \geq \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} (c_{ij}^1 x_{ij}^1 + c_{ij}^2 x_{ij}^2 + c_{ij}^3 x_{ij}^3)}{\alpha} \tag{12}$$

$$\sum_{i=1}^{N} \sum_{j=1}^{N} (x_{ij}^2 + x_{ij}^3) \geq y \tag{13}$$

$$z \geq 0.4 \ y \tag{14}$$

$$\sum_{i=1}^{N} \sum_{j=1}^{N} x_{ij}^3 \geq z \tag{15}$$

$$x_{ij}^1, x_{ij}^2, x_{ij}^3 \in \{0, 1\}, \qquad i, j = 1, \ldots, N \tag{16}$$

$$y, z \geq 0 \text{ integer.} \tag{17}$$

Constraints (10) and (11) require that each vertex has exactly one in-going arc and one out-going arc, respectively. Constraints (12) and (13) ensure that the total number of single or double rest arcs is at least equal to the total length of the solution, expressed in weeks. Similarly, constraints (14) and (15) ensure that the total number of double rest arcs is at least equal to 0.4 times the total number of single and double rest arcs.

Note that $v(RP)$ is expressed in minutes, but due to the structure of the arc lengths, it always corresponds to an integer number of days, which represents a lower bound on the total number of days required to cover all the duties. Hence y is a lower bound on the number of weeks in an optimal CRP solution.

A Lagrangian lower bound on $v(\mathrm{RP})$ can be obtained as follows. We first relax in a Lagrangian way constraints (13) and (15), with nonnegative Lagrangian multipliers λ_1 and λ_2, respectively, obtaining the objective function

$$min\ \lambda_1 y + \lambda_2 z + \sum_{i=1}^{N}\sum_{j=1}^{N}(\bar{c}^1_{ij}x^1_{ij} + \bar{c}^2_{ij}x^2_{ij} + \bar{c}^3_{ij}x^3_{ij}), \tag{18}$$

where $\bar{c}^1_{ij} := c^1_{ij}$, $\bar{c}^2_{ij} := c^2_{ij} - \lambda_1$, $\bar{c}^3_{ij} := c^3_{ij} - \lambda_1 - \lambda_2$, for all $i, j = 1, \ldots, N$ are the Lagrangian costs for the x-variables. Furthermore, we replace constraint (12) with

$$y \geq \frac{\lambda_1 y + \lambda_2 z + \sum_{i=1}^{N}\sum_{j=1}^{N}(\bar{c}^1_{ij}x^1_{ij} + \bar{c}^2_{ij}x^2_{ij} + \bar{c}^3_{ij}x^3_{ij})}{\alpha}. \tag{19}$$

Let $\mathrm{LRP}(\lambda_1, \lambda_2)$ denote problem (18), (10), (11), (19), (14), (16) and (17) and let $v(\mathrm{LRP}(\lambda_1, \lambda_2))$ be its optimal solution value. It can be shown that this value is a valid lower bound on $v(\mathrm{RP})$.

Problem $\mathrm{LRP}(\lambda_1, \lambda_2)$ can be solved by using an associated *Assignment Problem* (AP)[6]. Given multipliers $\lambda_1, \lambda_2 \geq 0$, an optimal solution to the problem can indeed be computed by (see [3])

i) solving the AP on the cost matrix defined by $d_{ij} := min\{\bar{c}^1_{ij}, \bar{c}^2_{ij}, \bar{c}^3_{ij}\}$ for $i, j = 1, \ldots, n$, thus obtaining the solution value $v(\mathrm{AP})$;

ii) determining the minimum y such that

$$\alpha y \geq \lambda_1 y + \lambda_2 \lceil 0.4y \rceil + v(\mathrm{AP}),\ \ y \geq 0 \text{ integer}; \tag{20}$$

iii) defining $z := \lceil 0.4y \rceil$.

Since the above-defined AP can be solved in $O(n^3)$ time, the overall time complexity for solving $\mathrm{LRP}(\lambda_1, \lambda_2)$ is $O(n^3)$.

One is interested in finding Lagrangian multipliers λ_1^*, λ_2^* maximizing $v(\mathrm{LRP}(\lambda_1^*, \lambda_2^*))$. In principle this could be done by using standard iterative techniques, such as subgradient optimization, solving an AP at each iteration. Even if AP's can be solved quickly in practice, such a procedure could be very time consuming for large-size CRP instances. Furthermore, computational experience has shown that for the real-world CRP instances we solved, the optimal values for λ_1, λ_2 are usually $\lambda_1^* = \lambda_2^* = 1440$. For small-size CRP instances, in a few cases, the optimal values for the Lagrangian multipliers are $\lambda_1 = 1440$, $\lambda_2 = 0$. We compute a lower bound value for CRP as $v^* := max\{v(\mathrm{LRP}(1440, 1440)), v(\mathrm{LRP}(1440, 0))\}$, by solving two AP's. This value is expressed in minutes, but due to the structure of the arc lengths, and to the values we assign to multipliers λ_1, λ_2, it always corresponds to an integer number of days, namely $d^* := v^*/1440$. The corresponding lower bound on the number of weeks in a solution to CRP is clearly given by $lb_2 := \lceil d^*/6 \rceil$. Although this lower bound is computed by neglecting many constraints imposed on CRP, the corresponding value is typically very tight (see Section 8).

Clearly the maximum of lb_1 and lb_2 gives a valid overall lower bound lb on the number of weeks.

[6] Given an $N \times N$ matrix D, the AP associated with D calls for a permutation π of $\{1, \ldots, N\}$ which minimizes $\sum_{i=1}^{N} d_{i\pi(i)}$.

6.2 Labeling Step

The construction of a feasible solution is a rather trivial problem: in fact, we can construct all the rosters sequentially and independently by choosing, one after the other, the chain of duties that will compose each roster. Once we have stated all the constraints, very little backtracking is needed since the solver automatically removes almost all inconsistent values. This simple strategy has a drawback: the feasible solution obtained can easily be far from the optimal one.

In order to find a good solution we exploit the information given by the lower bounding procedure, which, applied to a set of duties D, gives us the minimum number of weeks W needed to find a solution to problem D. This means that each feasible solution to the problem will need at least W weeks; moreover, if we find a feasible solution W weeks long, then we are sure that this solution is an optimal one.

Since, for real-life instances, the value returned by the lower bounding procedure is typically very close to the number of weeks needed for the optimal solution, we can see the lower bounding procedure as an "oracle" capable of telling us how many weeks we should use for our solution.

Therefore, by trying to satisfy the additional constraint on the number of weeks to be used, we actually transform an optimization problem, in which it is easy to find a feasible solution, into a more difficult constraints satisfaction problem whose solution, if any exists, is usually a very good one. While generating the solution, we update the evaluation of the number of weeks needed (by recalculating the lower bound) every time we complete the construction of a roster.

The problem of finding a quasi-optimal feasible solution is typically very hard, mainly because of the huge number of constraints which need to be satisfied and the very large size of the search space. Different strategies have been used in order to reduce the search space and to increase the overall efficiency. Two of them will be explained in detail for their importance and generality: *bottleneck analysis* and *jumping selection strategy*. Also some aspects of the symmetries of the problem will be considered.

6.2.1 Bottleneck Analysis

Bottleneck analysis is a heuristic strategy which follows the *First Fail Principle*.

Consider a flow of liquid through a pipe. If the diameter of the pipe is not constant, the rate at which the liquid can flow through the pipe is equal to the rate at which the liquid can flow through the narrowest portion of the pipe.

In dealing with scheduling problems a very useful strategy is to identify bottlenecks in the flow of material and try to schedule those bottlenecks first, in order to obtain the highest throughput, see [2].

In a CRP, performing a bottleneck analysis means detecting the most difficult constraint that has to be satisfied, and trying to handle first those duties which participate most in that constraint.

A simple example is the following: suppose we have a problem P defined by the set of duties D. The lower bound value of the problem is equal to 10 weeks; suppose we have 20 overnight and 3 long duties. Since a constraint exists imposing no more than 2 duties per week, in order to find a solution 10 weeks long, we need to fill up completely, in each week, all the capacity of overnight duties; therefore, in each week we need at least and at most 2 overnight duties. Following the *First Fail Principle*, it is thus convenient to handle the overnight duties first. On the other hand, a constraint exists imposing no more than 1 long duty per week. In order to find a solution 10 weeks long, we need to have an average of 0.3 long duties per week, which is far from the maximum of long duties; then, the constraint on long duties is not very tight and we do not worry too much about it.

In the following, we will describe the exact strategy adopted in the algorithm. The key idea is to assign a score to each duty (depending on its contribution to each tight constraint), and to impose it on each week. We compute an overall score greater than a certain threshold.

The constraints used for this analysis are the following:

1. for each week there can be no more than 2 overnight duties
2. for each week there can be no more than 1 long duty
3. for each roster (5 weeks) there can be no more than 7 duties with external rest.

Constraints concerning the working and the paid time are not taken into account in this analysis since they are implicitly considered in the lower bound procedure. In this way, we always obtain rosters with good values of working and paid time.

We first consider dynamic coefficients, α_O, α_L, and α_E which are computed as follows. Given the number of overnight duties, long duties and duties with external rest not yet assigned, N_O, N_L and N_E, we first discard the zero-valued coefficients and then compute their minimum, MIN. Then we have

$$\alpha_O = \frac{N_O}{MIN},\ \alpha_L = \frac{N_L}{MIN},\ \alpha_E = \frac{N_E}{MIN}$$

Given the values of α_O, α_L, and α_E, the score of each duty is given according to the following criteria:

- if duty i is overnight $\rightarrow$ score $\delta S_i = \frac{1}{2} \cdot \alpha_O$
- if duty i is long $\rightarrow$ score $\delta S_i = 1 \cdot \alpha_L$
- if duty i has external rest $\rightarrow$ score $\delta S_i = \frac{5}{7} \cdot \alpha_E$
- if duty i has a combination of two such characteristics, its score is equal to the highest score given by each single characteristic.

Knowing the number of overnight, long, and external rest duties (N_O, N_L and N_E respectively), and knowing an evaluation of the number of weeks available W (given by the lower bounding procedure), before starting the construction of each week, we compute the values:

$$T_O = \frac{N_O}{2 \cdot W},\quad T_L = \frac{N_L}{W},\quad T_E = \frac{7 \cdot N_E}{5 \cdot W}$$

these values give us information on the tightness of the three constraints.

The overall score we want to obtain for the current week is:

$$S = \alpha_O \cdot T_O + \alpha_L \cdot T_L + \alpha_E \cdot T_E$$

Therefore, the following *condition on the acceptance of a week* has to hold:

$$\sum_{i \in week} \delta S_i \geq S$$

It is worth noting that finding a set of duties such that this inequality is satisfied is not guaranteed; in fact, the number of weeks W used to calculate T_k is an optimistic evaluation of the number of weeks of the optimal solution. In case we cannot find any set of duties that satisfy the inequality, we consider the set of duties with the minimum value of δS_i.

6.3 Jumping Selection Strategy

Rosters are constructed sequentially. Also, for each roster, weeks are constructed sequentially. The first duty to be considered is the first available where $s_i > 6:30\ a.m.$. In this way, we solve the following constraint by subsumption: if an overnight duty precedes a single weekly rest, the first duty of the week cannot start earlier than 6:30 a.m. As a matter of fact, given a certain duty, we first instantiate the variable $Time$ which gives us complete information on variables Day and $Week$. Second, we choose the next duty to be performed by instantiating the variable $Next$.

At the end of each week we commit the choices made so that no backtracking is possible from one week to the previous one. Nevertheless, the very large domain of variables $Next$ leads to a *thrashing effect* [12] which essentially concerns failures in different parts of the search space due to inconsistent instantiation in the highest level of the tree.

In order to minimize the thrashing behavior, we adopt a strategy called *jumping selection*: instead of considering all possible values of the current variable $Next$ as valid options, we reduce *a priori* its domain.

We divide the set of possible next duties (represented by values in the domain of current $Next$ variable) into homogeneous groups. One group can contain duties with a common characteristic such as long duties, another group contains overnight duties and so on. For each group we consider only the duties which are the most likely to succeed, then we skip the remaining duties in the group and change the kind of duty.

This improvement, on one hand, makes it possible to avoid the backtracking thrashing behavior, but, on the other hand, leads to an incomplete search algorithm. However, this incompleteness does not prevent us from finding good solutions.

In each homogeneous group, the first duty we try to instantiate as variable $Next$ is that duty which produces the minimum time lag. For example, suppose we need to select a value to assign to $Next_i$, suppose also that f_i is equal to $2:10$, and that between i and $Next_i = j$ a time interval of 18 hrs needs to exist, then the first choice on duty j will be that duty which has the least start time greater than or equal to 20:10 (2:10 + 18).

If our current choice fails, the next choice will fall into that duty which produces the minimum time lag, and ends earlier than the last failed duty.

This strategy tries to minimize the time lag between duties and therefore also technical intervals. In addition, it tries (if required by failures) to leave the most amount of time at the end of the week for further duties.

The hypothesis underlying this strategy is the following: if a duty k, ending at time T, generates a failure, then every duty with the same characteristics as those of duty k, ending later than T would generate a failure as well. Even if such hypothesis is not always true, it sounds reasonable, and gives very good results.

6.4 Symmetries

Following the definition given in [24], a symmetrical problem is a problem where some permutation of the variables maps a solution onto another solution.

In the CRP, we can identify two different kinds of symmetries:

- symmetries on the group of rosters, due to the fact that a solution is defined by a set of unsorted rosters: solution ($Roster_A$, $Roster_B$, $Roster_C$) is perfectly equivalent to solution ($Roster_C$, $Roster_B$, $Roster_A$)

- symmetries on each single roster, due to the fact that a roster can be defined by a cyclic group of weeks: roster $Roster_A$ composed of ($Week_1$, $Week_2$, $Week_3$, $Week_4$, $Week_5$) is perfectly equivalent to roster $Roster'_A$ composed of ($Week_4$, $Week_5$, $Week_1$, $Week_2$, $Week_3$).

Symmetries, in a CSP, are always sources of trouble and should be avoided by imposing additional constraints. In fact, symmetries greatly increase the search space and may transform an easy problem into an unsolvable one [24].

As regards this particular problem, for example, it would be very easy to eliminate the first kind of symmetries by imposing on each roster an overall working time smaller than the previous one.

However, since at the end of each week we commit all the choices made, backtracking is never deep enough to find two equivalent rosters or two equivalent solutions. Therefore, we do not need to make any effort to eliminate the symmetries. However, the second kind of symmetry allows us to make some committed choices which do not compromise the completeness of the algorithm, while highly increasing its efficiency. We establish, for example, that the first duty of the first week of each roster must start after 6:30 a.m. This choice allows us to satisfy some constraints between the last and the first week of each roster, the satisfaction of which would be otherwise computationally very expensive. Moreover, it does not compromise the completeness of the algorithm since it is very unlikely to have a roster in which each first duty of each week starts earlier than 6:30 a.m.

7 Post-Optimization

In this section, we present a post-optimization procedure which allows us to improve the first solution found. Given a feasible solution, the aim is to make some local changes in that solution in order to find a better one.

Since each roster has a duration of 5 weeks, except for the last roster (the only one that can be less than 5 weeks long), from now on we will refer to the last roster as R^*, while we will refer to any of the others as roster R_i in general. The post-optimization procedure is based on the following idea: we look for local improvements by iterating the following steps. We evaluate the quality of each roster but the last one by using the lower bounding procedure, which gives us an optimistic evaluation of the number of weeks needed. Then, we select the worst roster R_1 and the one R_2 which, together with R_1, has the highest probability of improving the solution.

Then, we reconsider and solve a CRP for rosters R_1, R_2 and the last one. Therefore, we consider the rostering problem defined by the set of duties

$$\{duty\, k | k \in R_1 \cup R_2 \cup R^*\}$$

If the solution found improves the overall solution, we substitute the rosters considered (R_1, R_2, R^*) with the new ones. Otherwise, we change the pair of rosters considered and restart the post-optimization procedure.

In this way, we can improve the solution mainly because the instance considered is smaller than the original one, and we can use more expensive and accurate heuristics. We also change some parameters of the algorithm in order to force the new solution (R'_1, R'_2, R'^*) to be different from the previous one (R_1, R_2, R^*).

Two points need to be clarified:

- how to choose the two rosters R_1 and R_2
- how to decide whether substituting R_1, R_2, R^* with R'_1, R'_2, R'^* improves the overall solution or not.

We use a simple criterion to choose R_1 and R_2: R_1 will be the worst roster in the current solution that has not been selected yet. R_2 will be the roster, not yet chosen as R_1, which, together with R_1 and R^*, has the highest probability of improving the solution.

It is clear that we need a function $val(R)$ that evaluates the quality of a roster R. Let S be the set of all duties; let D_R be the set of all duties in roster R ($D_R = \{duty\, k | k \in R\}$); let $lb(D)$ be the lower bound value calculated on the set of duties D, as described in Section 6.1. We can thus define $val(R)$ such as:

$$val(R_i) > val(R_j) \Leftarrow lb(S \setminus D_{R_i}) < lb(S \setminus D_{R_j})$$

$$val(R_i) > val(R_j) \Leftarrow lb(S \setminus D_{R_i}) = lb(S \setminus D_{R_j}) \,\&\, lb(D_{R_i}) > lb(D_{R_j})$$

whose meaning is the following: roster R_i is better than roster R_j if the expected number of days needed to solve the rest of the problem $(S \setminus D_{R_i})$ is less then the expected number of days needed to solve $(S \setminus D_{R_j})$. If the expected number of days needed to solve $(S \setminus D_{R_i})$ is equal to the expected number of days needed to solve $(S \setminus D_{R_j})$ then, since R_i and R_j are both 5 weeks long, we say that R_i is better than R_j if the expected number of days needed to solve D_{R_i} is greater than the expected number of days needed to solve D_{R_j}.

Again we can use the lower bound to choose R_2: we define

$$val_{R_1}(R) = lb(D_{R_1} \cup D_{R^*} \cup D_R)$$

Since R^* uses N weeks ($N \leq 5$), then the partial solution R_1, R_2, R^* uses exactly $10 + N$ weeks for all R in the solution. Therefore, we have the highest probability to improve the solution by considering (as roster R_2) the roster R with the minimum value $val_{R_1}(R)$ which represents the minimum number of days needed to solve the problem $P(D_{R_1} \cup D_{R^*} \cup D_R)$.

Once we have obtained the new partial solution R'_1, R'_2, R'^*, we decide whether we have to substitute it to R_1, R_2, R^* or not according to the following criterion: replace R_1, R_2, R^* with R'_1, R'_2, R'^* if

$$lb(R'^*) < lb(R^*) \textit{ or}$$

$$lb(R'^*) = lb(R^*) \textit{ and } max(val(R'_1), val(R'_2)) > max(val(R_1), val(R_2)) \textit{ or}$$

$$lb(R'^*) = lb(R^*) \textit{ and } max(val(R'_1), val(R'_2)) = max(val(R_1), val(R_2)) \textit{ and}$$

$$min(val(R'_1), val(R'_2)) > min(val(R_1), val(R_2))$$

The first condition ($lb(R'^*) < lb(R^*)$) is easy to understand: since the number of weeks used for problem $S \setminus D_{R^*}$ and for problem $S \setminus D_{R'^*}$ is the same, the lower the value $lb(R^*)$, the higher the probability of reducing the number of weeks used in R^*, i.e., used in the overall solution.

Given that $lb(R'^*)$ is equal to $lb(R^*)$, an intuitive way of considering improvements in the solution (as explained in the definition of function $val(R)$) is the following: replace R_1, R_2, R^* with R'_1, R'_2, R'^* if

$$lb(S \setminus (D_{R'_1} \cup D_{R'_2})) < lb(S \setminus (D_{R_1} \cup D_{R_2}))$$

However, this strategy would need the (usually expensive) calculation of those two lower bound values, as well as the calculation of $lb(S \setminus D_{R'_1})$ and $lb(S \setminus D_{R'_2})$ (if R'_1 and R'_2 become part of the solution) since it is needed to evaluate $val(R'_1)$ and $val(R'_2)$.

Moreover, real cases have shown that the optimal solution is reached when almost each roster has the same value $val(R)$, equal to the highest possible. It is thus convenient to encourage any improvement in that direction (as we can do by using the criterion shown above).

Even if the results of the procedure are encouraging, it is sometimes quite computationally expensive. The improvement of this procedure will be a subject of future work.

8 Results

We have implemented the system on a Pentium 100 MHz with 16 Mb RAM and the Linux operating system. We have used the Constraint Logic Programming language ECL^{i}PS^e and its finite domain library, in particular. The lower bounding procedure is written in C and linked to the CLP program.

The CLP paradigm is very suitable for this kind of application since the problem domain can be straightforwardly mapped in a CLP language. Finite domain constraint solvers like ECL^{i}PS^e provide a very powerful set of symbolic constraints (as, for example the constraint *atmost/3* used in Section 5.2), and the possibility of writing new user-defined constraints together with their propagation mechanisms. These capabilities are very useful in dealing with real-life problems because they make it possible to express complex relations between variables and the way of handling them.

In the following table, we present some results in terms of computational time and the quality of the solution found. We report the lower bound of the problem (*lb*) explained in Section 6.1, the results of the OR approach presented in [3] and our approach in terms of number of weeks and CPU time. We present the results for real-life problems (for increasing number of duties, N) taken from the Italian Railway Company.

N	*lb*	OR		Combined CLP/OR	
Duties	Weeks	Weeks	Time (sec)	Weeks	Time (sec)
21	7	7	7	7	2
33	11	11	15	11	6
69	19	19	585	19	246
134	39	39	328	39	447
164	48	48	95	48	107
360	108	111	307	111	466
386	118	118	398	118	1045
525	164	164	1066	164	1883

The quality of the solution found is the same in the OR and in the combined CLP/OR approaches. In seven out of eight cases the solution value is equal to the lower bound value (thus being optimal). As concerns the performance of the two approaches, they are very similar. Thus, the results are very encouraging since the CLP/OR prototype has been developed in a very short time and the code is concise, readable, extensible and easily modifiable, while the implementation of the OR approach required a considerably bigger effort, even if the corresponding computing times are slightly smaller.

In the CLP/OR approach, a significant fraction of the CPU time is used by the post-optimization procedure. In fact, if we accept a slightly worse solution, the computational time consistently decreases. In the next table, we show computational results that show how much the post-optimization procedure influences the performances and the quality of the solution. In the table, the first column corresponds to the size of the problem, the second and the third are the number of weeks of the first solution and the CPU time for reaching it, respectively. The following columns represent increasingly better solutions (until the best one found) and the corresponding CPU times.

N	I sol.		II Sol		III sol		IV sol	
Duties	Weeks	Time (sec)	Weeks	Time (sec)	Weeks	Time (sec)	Weeks	Time (sec)
21	7	2						
33	11	6						
69	20	77	19	246				
134	40	183	39	447				
164	49	80	48	107				
360	113	381	112	446	111	466		
386	119	525	118	1045				
525	167	1080	166	1208	165	1620	164	1883

In the problem with 386 duties, if we accept a solution of 119 weeks (one more than the optimal solution), the computational time is divided by two. This also happens in the instance of 134 duties. In this case the computational time for finding a solution with 40 weeks (again, one more than the optimal solution) is almost one third of the whole time. Some instances, as the one with 525 weeks, require three post-optimization steps because the quality of the first solution is worse than the optimal one (3 more weeks). The post-optimization procedure makes it possible to increase the quality of the solution, but it takes about half the CPU time. Therefore, a future work will be devoted to improving this procedure.

9 Conclusions

In this paper, we have presented a constraint-based approach to a crew rostering application for the Italian Railways. Previous approaches to this problem were mainly based on OR techniques and on a mathematical formalization of the problem.

To our knowledge, there is no other CLP approach to the CRP. Our approach is mainly based on the CLP paradigm enhanced with a lower bounding procedure taken from the OR field [3]. As a future work, we will investigate a pure CLP approach by using sophisticated CLP tools like CHIP [8] and its global constraints.

The use of the lower bounding procedure taken from the OR field allows us to obtain excellent performance even on large sized and hard problems. In addition, we think that by improving some heuristics and the post-optimization procedure, we can increase the performance, since we noticed that a significant fraction of the CPU time is taken by the post-optimization procedure.

Moreover, by using the CLP approach, we can benefit from the declarative way of programming which leads to short, readable, extensible and easily modifiable programs. In addition, the conciseness of programs and the extremely rapid prototyping and testing time make it possible to rapidly develop alternative versions and compare their efficiency and usefulness.

References

[1] L. Bianco, M. Bielli, A. Mingozzi, S. Ricciarelli, and M. Spadoni. A heuristic procedure for the crew rostering problem. *European Journal of Operational Research*, 58:272–283, 1992.

[2] S. Breitinger and H.C.R. Lock. *Using Constraint Logic Programming for Industrial Scheduling Problem*, chapter 9. Elsevier Science B.V, 1995.

[3] A. Caprara, M. Fischetti, P. Toth, and D. Vigo. Modeling and solving the crew rostering problems. Technical report, DEIS Università di Bologna OR-95-6, 1995.

[4] P. Carraresi and G. Gallo. A multilevel bottleneck assignment problem for the bus drivers' rostering problem. *European Journal of Operational Research*, 16:163–173, 1984.

[5] A. Colmerauer. An introduction to Prolog III. *ACM Communication*, 33(7):70–90, 1990.

[6] M. Dincbas, P. Van Hentenryck, and H. Simonis. Solving the car sequencing problems in Constraint Logic Programming. In *ECAI88, European Conference on Artificial Intelligence*, 1988.

[7] M. Dincbas, P. Van Hentenryck, and H. Simonis. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8 (1-2):75–93, 1990.

[8] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer System (FGCS88)*, pages 693–702, 1988.

[9] ECRC. *ECL^iPS^e User Manual Release 3.3*, 1992.

[10] M. Gamache and F. Soumis. A method of optimally solving the rostering problem. Technical report, Les Cahiers du GERAD G93-40, 1993.

[11] M. Gamache, F. Soumis, G. Marquis, and J. Desrosiers. A Column Generation Approach for Large Scale Aircrew Rostering Problem. Technical report, Les Cahiers du GERAD G94-20, 1994.

[12] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie Mellon University, 1979.

[13] N. Guerinik and M. Van Caneghem. Solving Crew Scheduling Problem by Constraint Programming. In U.Montanari and F.Rossi, editors, *Proceedings of the 1st Conference of Principles and Practice of Constraint Programming*, pages 481–498, 1995.

[14] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[15] J. Jaffar and J.L. Lassez. Constraint Logic Programming. In *Proceedings of the Conference on Principles of Programming Languages*, 1987.

[16] J. Jaffar and M.J. Maher. Constraint Logic Programming: a Survey. *Journal of Logic Programming Special Issue on 10 years of Logic Programming*, 19-20:503–582, 1994.

[17] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The CLP(R) language and system. *ACM Transaction on Programming Language and Systems*, 14(3):339–395, 1992.

[18] R. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[19] E. Lamma, P. Mello, and M. Milano. A Distributed Constraint-Based Scheduler for Railway Traffic. *Artificial Intelligence in Engineeering*, 1996. To appear.

[20] J.W. Lloyd. *Foundation of Logic Programming.* Springer-Verlag, 1987. Second Extended Edition.

[21] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[22] E. Contejean N. Beldiceanu. Introducing Global Constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.

[23] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley and Sons - New York, 1988.

[24] J.F. Puget. On the Satisfiability of Symmetrical Constrained Satisfaction Problems. Technical report, ILOG Headquarters, 1993.

[25] J.F. Puget. A C++ implementation of CLP. Technical report, ILOG Headquarters, 1994. N.94–01.

[26] D.M. Ryan. The Solution of Massive Generalized Set Partitioning Problem in Air-crew Rostering. *Journal of the Operational Research Society*, 43:459–467, 1992.