Industrial Linear Optimization Problems Solved by Constraint Logic Programming

R. Bisdorff and S. Laurent

Centre de Recherche Public - Centre Universitaire, 162a av. de la Faïencerie, L-1511 Luxembourg

Abstract: In this article we try to illustrate that constraint logic programming (CLP) systems allow easy expression and solution of constrained decision problems. In order to do so, this paper proposes CLP solutions for two industrial linear optimization problems respectively using the Prolog III and the CHIP language. The first problem, a mixed linear multicriteria selection problem, illustrates the general linear rational solver. In order to fix some integer variables a branch and bound rounding heuristic is formulated. The second problem, a linear integer multicriteria location problem, is only concerned with integer finite domain variables and is particularly adapted to the CHIP system that provides a computation domain handling such variables.

Keywords: constraint logic programming, branch and bound techniques, finite domain computation, linear optimization, goal-programming, multicriteria selection, industrial disposing problem, industrial production scheduling

1. Introduction

This article presents the studies that were conducted at the Centre de Recherche Public – Centre Universitaire of Luxembourg in collaboration with the ARBED Luxembourg s.a. steel industry about the solving capacities of constraint logic programming (CLP) systems [9] in industrial linear optimization problems. Indeed, the declarative power of logic programming and the availability of integrated constraints system solvers make these CLP systems, like CLP($\mathcal{R}$) [9], Prolog III [4,5], or CHIP [6,7,8], to some extent potential alternatives for the traditional mathematical programming (MP) modelling environments and solvers, like MPSIII from Ketron and XPRESS, or OSL from IBM and the MP Fortran library proposed by NAG.

Originally, Prolog systems, based on the solving principle of logic unification of Horn clauses in a Herbrand universe, were designed to deal with natural language problems. Very soon however, when dealing with numerical problems, it became interesting to replace the simple Prolog unification with a general constraints solving mechanism, which led to the CLP systems like the Prolog III system [4]. The kernel of this Prolog system consists in a general constraints system solver, which first tests if a constraints system is solvable and then tries to simplify the system in order to make the solutions appear. The numerical sub-kernel, which is used for the resolution of rational linear equations and inequalities, includes a specially adapted simplex solver and a special dynamic simplification mechanism.

A first industrial problem, formulated as a mixed- linear goal-programming problem, concerns a multicriteria selection tool for the daily production in a coil rolling mill at the ARBED Dudelange plant [1,2]. This problem is solved with the help of the Prolog III system. Thus we can illustrate the generic linear-programming facilities of the CLP systems. To solve the complete mixed problem, we furthermore show the implementation of a simple branch-and-bound rounding heuristic.

The introduction of consistency-checking and constraint solving techniques also characterizes the CHIP system [7]. In addition to Prolog III, it offers these techniques as well in finite positive integer domains [8]. Some higher-order predicates for optimization by means of a kind of depth-first branch-and-bound technique thus allow combinatorial optimization and integer linear-programming problems to be solved.

To illustrate these features we present a second industrial problem, appearing as a complete integer problem, that concerns the optimal disposing of steel girders before expedition at the ARBED Differdange plant [3]. This problem is solved with the help of finite domain computations in the CHIP system. As a normal Prolog execution easily allows enumeration of a set of discrete solutions, this example will illustrate the ease of use of the CHIP integrated branch-and-bound solver.

2. Introducing the Industrial Problems

The problems considered appear at the ARBED Luxembourg s.a. steel industry. The first one, entitled "*coils selection problem*", is concerned with the production schedule of the rolling mill for coils at the ARBED Dudelange plant [1,2].

2.1. *The Coils Selection Problem*

In order to reach a particular thickness, coils are rolled at the ARBED Dudelange plant. Coils are defined by a set of characteristics such as weight, width, postrolling thickness, postrolling treatment and ultimate delay of production. Coils having all the same characteristics are gathered to form an order post.

The list of coils that will be produced during the next 8 hours has to be determined. These coils are selected among each post available in the stock and the following criteria restrict the possible choices.

The selected coils amount to a total weight that cannot exceed the production capacity of the rolling mill.

For each postrolling treatment, percentages are defined in order to restrict the number of selected coils that have to undergo this processing.

All the coils in stock belong to several width classes. In order to reduce the presetting checks occurring when a width class is changed, if a coil is selected in one of them, then at least a defined number of coils has to be chosen in this class.

Finally, the most urgent coils must be selected in order to respect the delivery delays.

The second industrial problem we consider, is related to the disposing of finished girders in the expedition parks at the ARBED Differdange plant [3]. We shall later on refer to this problem as the "*girders disposing problem*".

2.2. *The Girders Disposing Problem*

A lot of finished girders produced at the ARBED Differdange plant are not immediately shipped and have to be disposed until they leave. The plant uses expedition parks which are divided into zones of different lengths and capacities where these products can be stocked. Between two parks, wagons come to load the girders ready for expedition and rolling bridges help to carry the girders into a zone or onto a wagon.

A girder is characterized by length, weight, order and customer. Girders sharing all four characteristics are gathered to create an order post.

Posts having the same order or customer are usually loaded on a same wagon for a common destination. Therefore, they must be disposed in one zone or in several zones near each other in order to minimize the moves of the wagons and rolling bridges at the time of their expedition.The distance between two posts is expressed by the distance between the zones in which they are placed.

Because of the specific characteristics of girders and zones, only a limited number of posts of a certain length can be disposed in a same zone.

In the following section we will illustrate the linear optimization capacities of CLP systems by solving the coils selection problem.

3. Solving General Linear Problems with CLP

In order to illustrate the use of a simplex solver in a CLP system, we shall first formulate the coils selection problem as a mixed goal-program. As the Prolog III system at the time of our experimentation did not provide any primitive predicate for immediate solving of a mixed MP problem, first, a relaxed rational solution in Prolog III is proposed. To fix the integrity constraints we then show the implementation of a branch-and-bound rounding heuristic. Incidentally this also allows us to show the Prolog programming style.

3.1. *Mathematical Formulation of the Coils Selection Problem*

The multicriteria coils selection problem can be viewed as a mathematical-programming problem where the last criterion - specifying that the most urgent coils have to be selected first - figures as an objective function:

$$\min_{x \geq 0} \; F = c \cdot x$$

x is a vector of n unknowns, i.e. the number of selected coils in each post available in the stock and c is a vector of estimated delays for the end of the milling process related to each post.

The limitation of the stock and the other criteria, defined above, give rise to the following constraints:

a) availability of coils in the stock:

$$x \leq u$$

where u is a constant vector representing the number of available coils in each post in stock,

b) the selected coils x amount to a total weight confined to a given interval $[b_m, b_M]$:

$$b_m - x_b^e \leq a \cdot x \leq b_M + x_b^e$$

where a is a constant vector which associates a unit weight with each coil of the corresponding post. The unknowns x_b^e are deviational variables which have to be minimized according to the relative preference priority given to this selection criterion.

c) the percentage of selected coils x which have to undergo the postrolling treatment t is confined to a given interval $[p_m^t, p_M^t]$:

$$\frac{p_m^t}{100} - x_t^e \leq \frac{a \cdot (x \bullet v^t)}{a \cdot x} \leq \frac{p_M^t}{100} + x_t^e$$

where v_t is a characteristic vector filtering the posts related to treatment t and «•» represents the term by term product operator. This constraint may be activated for each possible postrolling treatment t. As in the preceding constraint, x_t^e represents a vector of deviational variables.

d) if a coil is selected in a given width class w, then at least k coils have to be selected in this class:

$$x_w^b \; b_w \leq v^w \cdot x \leq x_w^b \; (v^w \cdot u)$$

where v^w is a characteristic vector filtering the posts of width class w and x_w^b is a binary variable specifying when coils have been selected or not in the width class w. We define b_w as follows:

$$b_w = \min (k, v^w \cdot u)$$

i.e. either k if there are at least k coils having the width w in stock or $v^w \cdot u$, the rest of coils having the width w in stock.

According to constraints *(b)* and *(c)*, the objective function becomes:

$$\min_{x \geq 0,\, x^e,\, x^b} F = p^e \cdot (F, x^e, x^b)$$

where p^e represents the relative priorities of the selection criteria.

This problem appears as a mixed-linear goal-programming problem, where the x are integer variables, the x^e are rational variables and the x^b are binary variables. It can be formulated as a Prolog III program on the basis of the following general solution.

3.2. *Solving Rational Linear Programming Problems*

CLP systems like Prolog III or CHIP provide facilities to state and solve linear programming problems. As a concrete example, we consider the following general case:

$$\min_{x \geq 0} F = c \cdot x$$

under the constraints

$$Ax = b.$$

Prolog III [4,5], like CHIP [6,7], can easily express this constraints system with coefficients and variables represented by rational numbers of possibly infinite precision. Moreover, these languages allow this problem to be solved by an adapted simplex method. The corresponding Prolog III program could be the following (The syntax of Prolog III follows the "Marseille" style, that is < ... > indicates a list, and constraints are enclosed in accolades):

```
min(c, b, x, F)->
      constraint(A, x, b)
      scal_prod(c, x, F)
      minimize(F),
      { x = <x1, x2, ..., x3>,
        c = <c1, c2, ..., cn>,
        A = < <a11, a12, ..., a1n>,
              <a21, a22, ..., a2n>,
              ...,
              <am1, am2, ..., amn>>,
        b = <b1, b2, ..., bm>};
```

Where the vectors x and c are represented by lists of n terms, the vector b is expressed as a list of m values, and the matrix A as a list of m vectors represented again by lists of n coefficients.

The predicate `constraint` dynamically constructs the linear constraints system:

```
constraint(<>, _, <>)->;
constraint(<a>.A, x, <b>.b')->
      scal_prod(a, x, r),
      constraint(A, x, b'),
      { r <= b};
```

On the other hand, the `scal_prod` predicate calculates the scalar product of two vectors in a recursive call:

```
scal_prod(<>, <>, 0)->;
scal_prod(<x>.X, <y>.Y, x * y + s)->
      scal_prod(X, Y, s);
```

Finally, the primitive predicate `minimize` computes, via an adapted simplex solver, the optimum solution value m of F considering the current system of constraints, adds the constraint `{F = m}` to this constraints system and simplifies it.

The simple and straightforward specification of a general linear MP problem demonstrates the declarative power of CLP systems. The Prolog text is quite close to the mathematical formulation and to some extent, the CLP system acts like a modelling environment. This interesting characteristic can be well illustrated with the help of the coils selection problem.

3.3. *Solving the Relaxed Rational Coils Selection Problem*

In order to formulate the constraints *(a)*, *(b)* and *(c)*, we construct the following `inrange` predicate which restricts term by term the elements of a list to particular boundary values `L_m` and `L_M`:

```
inrange(<>, <>, <>)->;
inrange(<x>.X, <m>.L_m, <M>.L_M)->
      inrange(X, L_m, L_M),
      {x >= m, x <= M};
```

The availability constraint *(a)*, combined with the fact that `x >= 0`, can be expressed with the help of the `inrange` predicate called with appropriate arguments:

```
inrange(x, v_0, u);
```

with `v_0` being the null vector of same dimension as `x`.

By combining this `inrange` predicate with the previously defined `scal_prod` predicate, the production capacity constraint *(b)* is again easily formulated:

```
scal_prod(a, x, r),
inrange(<r>, <b_m - x_eb>, <b_M + x_eb>);
```

where `x_eb` is the deviational variable attached to this constraint.

For each postrolling treatment t, the constraint *(c)* is expressed in the same manner:

```
vect_prod(x, v_t, x_t),
scal_prod(a, x_t, r_t),
inrange(<r_t>, <p_mt/100*r - x_et>,
        <p_Mt/100*r + x_et>);
```

with the following `vect_prod` predicate realizing the term by term product of two vectors:

```
vect_prod(<>, <>, <>)->;
vect_prod(<x>.X, <y>.Y, <x*y>.Z)->
      vect_prod(X, Y, Z);
```

Finally, the objective function is stated as follows:

```
scal_prod(c, x, F),
scal_prod(p_e, <F>.x_e, F'),
minimize(F');
```

assuming that `x_e` and `p_e` take the following form:

```
x_e = <x_e1, x_e2, ..., x_et, ...>,
p_e = <p_F, p_e1, p_e2, ..., p_et, ...>;
```

The CLP formulation thus appears as a formal model rather close to the mathematical formulation. In this sense the CLP systems appear as modelling environments for linear MP problems.

As mentioned above, the Prolog III system actually does not provide any primitive predicate for immediate solving of an integer or a mixed linear MP problem. Instead, we used a simple rounding procedure in order to transform the optimal values of x found with the `minimize` predicate into integers. This approach, quite acceptable in a goal-programming context, has proved fairly adequate in practice. At this point, it is important to note the impossibility to implement a traditional rounding procedure based on the dual solution of the problem as the simplex tableau and the rational solution base are not accessible through the `minimize` predicate.

Experimentation also showed that the rational values of the x_w^b vector are not uniquely determined. To get a single rational solution, we first solved the relaxed problem which does not consider the "width class" constraint *(d)*. In a second step, we implemented a progressive branch-and-bound like rounding procedure in order to satisfy the omitted constraints. This also allows us to show the Prolog programming style.

3.4. *Satisfying the «Width Class» Constraint*

The optimal values of the x vector which satisfy the relaxed rational problem being found, we use a progressive branch and bound technique to find a solution satisfying the constraint *(d)*.

First, all the selected coils which do not respect the constraint *(d)* are eliminated from the solution and the corresponding width classes are kept in a list W. The x_w^b variables associated with these classes are non-integers. We get a feasible solution `z` and its objective function's value G. The process is decomposed into three steps.

The bounding step consists in searching, for all unsatisfied width classes W, for the corresponding list of possible improvements L due to the fact that coils are selected in these classes as explained before:

```
bound(_, _, <>, <>)->;
bound(z, G, <w>.W, <l>.L)->
      compute(z, w, G0),
      maximum(G - G0, 0, l),
      bound(z, G, K, L);
```

The `compute` predicate is a predefined external user-rule written in C, which calculates the value of the objective function if we select the coils of class `w` (The possibility of defining external user-rules in C makes the Prolog III system very attractive for developers). The `maximum` predicate chooses the best possible improvement related to this class. Again, we miss any information about the simplex tableau and therefore it is not possible to use dual variables to select the best improvements. Instead we used a simple heuristic selecting rule like «the most urgent coils».

At the separating step, we consider the maximum improvement `p1` found in the list `L` related to a width class `w`. We compute the solutions `z0`, `z1` and their objective functions `G0` and `G1` corresponding to the two possible values of the x_w^b variable:

```
separate(z, G, L, W, z0, z1, G - p0,G - p1,
         W')->
      max_dif(L, W, w, p1),
      remove(W, w, W'),
      new_solution(z, w, z1),
      {z0 = z, p0 = 0};
```

Of course, the solution associated with the choice $x_w^b = 0$, for which no coils of the class `w` are selected, is the actual solution `z` and the objective function's value is `G`.

The `max_dif` predicate determines the class `w` which has the maximum improvement in the list `L`. The `remove` predicate takes the class `w` out of the set `W` and the `new_solution` predicate is again a C user-rule which computes the solution vector `z1` associated with the choice of the class `w`.

The branching step chooses either the solution $x_w^b = 0$ or the solution $x_w^b = 1$ according to the improvement associated with these respective possibilities.

```
branch(z0, z1, G0, G1, W, z', G')->
      branch_bound(z0, G0, W, z', G'),
      {G0 <= G1};
branch(z0, z1, G0, G1, W, z', G')->
      branch_bound(z1, G1, W, z', G'),
      {G0 > G1};
```

Finally, the main predicate `branch_bound` gathers the three cases developed above:

```
branch_bound(z, G, <>, z, G)->;
branch_bound(z, G, W, z, G)->
   bound(z, G, W, L)
   max_list(L, p),
   {p = 0};
branch_bound(z_init, G_init, W, z_fin, G_fin)-
             >
   bound(z_init, G_init, W, L)
   max_dif(L, W, _, p)
   separate(z_init, G_init, L, W, z0, z1, G0,
            G1, W')
   branch(z0, z1, G0, G1, W', z_fin, G_fin),
   {p > 0};
```

If the list `W` of width classes is empty, the problem is solved. If the list of possible improvements only contains zero values then the actual solution `z` is the final one. Otherwise, the bounding, separating and branching steps are executed.

3.5. *Experimental Results and Discussion*

This solution was tested on a real selection problem. 100 coils had to be selected among 429 coils divided into

167 order posts. The total weight of the chosen coils had to be confined between 2000 and 2200 tons. The percentage of selected coils for each of the three postrolling treatments had to be respectively between 18 and 32%, 28 and 32%, 48 and 52%. Finally, at least four coils, if possible, had to be selected in each width class.

On a Sun Sparc Station IPC the generation of the constraints system required 8 seconds while the simplex resolution for the relaxed rational problem takes 65 seconds. This model can easily be translated into a CHIP program (this translation requires only syntactic modifications due to the different underlying Prolog dialects) which would demand, for the same data, 9 seconds for constraints generation and 19 seconds for the optimization part.

The coils selection solution illustrates not only the declarative power of CLP systems but also their solving capacities. The execution times are quite satisfying and may be compared to traditional MP solvers. To some extent, they appear as modelling environments for linear MP problems as they provide not only an integrated simplex solver but also an integrated dynamic management of the constraints system.

But at this point the first drawback of the actual CLP systems for solving linear MP problems appears. The CLP simplex solver is embedded as a black box and it returns only the primal solution without any information about the context of the solution as described for instance in the simplex tableau (A reasonable suggestion for the CLP software industry would be to try to include optionally an access to the simplex tableau for experienced MP users).

But if rational variables could be avoided, these techniques should not be necessary for solving an integer linear programming problem. In fact, the coils selection problem could be reformulated as a complete integer problem. However, the CHIP system gives an answer to these kinds of problems because it provides the possibility of working with positive finite integer domain variables. The following solution of the girders disposing problem is a representative example using this powerful tool.

4. Solving Integer Linear Programming Problems in CHIP

Indeed, CHIP's positive finite integer domains facilitate the modelling and solving of discrete combinatorial problems [7,8].

4.1. Finite Domain Computations in CHIP

CHIP uses finite domain variables whose values range over a finite set of constants or natural numbers. Moreover, constraints between these variables can easily be established: arithmetic constraints, symbolic constraints (which state a logical or functional dependence between domain variables) and even user-defined constraints. CHIP also treats the constraints in an active way by solving them immediately or by reducing the search space as much as possible. The following example, defining a relation between two variables, illustrates these characteristics (The syntax of CHIP follows the "Edinburgh" style, that is, lists are delimited with square brackets. The %-phrases comment on the actual domains of the variables):

```
relation(N,V):-
    N :: 1 .. 5,      % N = {1, 2, 3, 4, 5}
    Image = [5, 3, 9, 1, 4],
    element(N,Image,V),% V = {5, 3, 9, 1, 4}
    V #>= 4,          % N = {1, 3, 5}, V = {5, 9, 4}
    N #<= 3,          % N = {1, 3}, v = {5, 9}
    indomain(N).      % {(N=1, V=5), (N=3, V=9)}
```

First, a domain variable `N` is defined. Its values range over the finite set of consecutive integers from `1` to `5`. The `element` predicate defines a functional relation between `N` and `V`. In fact, `N` specifies an index to the list `[5,3,9,1,4]` for the variable `V` (if `N=1` then `V=5`...). Therefore, the domain of `V`, that is the image of the `relation(N,V)`, corresponds to this finite set of constants. The next constraint requires that `V` must be superior or equal to `4`. This implies that the values `1` and `3` are removed from the domain of `V` and, according to the `element` constraint, `N` is now restricted to the integers set `[1,3,5]`. Thus co-restrictions on the `relation(N,V)` may be activated. The last constraint acts like a restriction on the domain of the relation. It excludes the integer `5` from `N`'s domain and consequently, `V` can only take the values `5` and `9`.

Thus, by simply testing the constraints, the CHIP system deduces the restricted domains of the concerned variables. Then a predefined `indomain` predicate generates the different solutions of this example. It instantiates `N` to the smallest value in its domain. On backtracking it assigns the next higher value in the domain to the variable and it tries all possible values from the smallest to the biggest. To each generated value of `N` corresponds a particular value of `V` with respect to the `element` constraint. Therefore, we obtain two solutions: `N = 1, V = 5` and `N = 3, V = 9`.

This labeling process can be exploited in order to find values of domain variables which optimize some objective function by using a kind of depth-first branch-and-bound technique as we did for the coils selection problem [8].

A generic example of such an optimization procedure is given below:

```
optimize(Varlist, Objective) :-
    define_domains(Varlist),
    generate_contraints(Varlist),
    objective_function(Varlist, Objective),
    min_max(labeling(VarList), [Objective]).
```

First a list of finite domain variables, `Varlist`, is defined. Then a second predicate generates constraints on these domains. A third call will construct an objective function in order to evaluate the possible instantiations of `Varlist` by means of the `Objective` value. And finally, a higher-order predicate `min_max` will search among the possible instantiations of `Varlist`, generated by the following `labeling` predicate, for the solution `Varlist` which is optimal for the `Objective`.

```
labeling([]).
labeling([V|Vs]):-
    indomain(V),
    labeling(Vs).
```

In cases where such an optimal solution is difficult to obtain, variants of these predicates allow the adaptation of the searching strategy by using ε-optimality or time-out to shorten the search time.

The solution of the girders disposing problem, which we present in the following section, takes advantage of these finite domain facilities.

First the mathematical formulation of the problem is described. Then the essential points of the corresponding CHIP program are presented and finally, the results are discussed.

4.2. *Mathematical Formulation of the Girders Disposing Problem*

Assume that $Z = \{z_1, z_2, ...z_m\}$ is the set of m zones composing the expedition parks and that $P = \{p_1, p_2, ...p_n\}$ is a set of n posts, where the first p posts are already in stock and the following n-p posts still have to be stocked. The final disposing of all the posts can be formulated as a correspondence $\lambda = (P, Z, L)$, where L represents a relation between the set P of posts and the set Z of zones.

To simplify the presentation we suppose that each post may only be disposed entirely onto one of the existing zones and each post is actually disposed somewhere. Thus the correspondence λ describes a functional relation between P and Z and the inverse relation L^{-1} defines a partition P/L^{-1} on P that describes the final gathering of posts in the zones.

The disposing problem can now be formulated as the search for a particular relation L such that the resulting partition P/L^{-1} satisfies our disposing criteria, for instance it should gather those posts that have the same order or customer.

In a lot of real cases, the dimension of the set of all possible relations between posts becomes important. For instance, if 150 new posts have to be disposed among parks composed of 40 zones where 500 other posts are already in stock, there are in theory about 40^{150} possibilities. But the disposing capacity of different zones and other access restrictions, like length considerations, limit the zones to be taken into consideration for each post. Indeed, the girders that form a particular post p_i have all the same length lp_{i_i} and a certain total weight w_i ($i = 1, ... N$). On the other hand, each zone z_j ($j = 1, ... m$) is characterized by its length lz_j and its capacity c_j. Therefore, only a limited number of posts of a certain length can be stocked in the same zone. This leads to the following capacity and length constraints.

Let $P_{/L_j^{-1}}$ denote the set of posts stocked in the zone z_j.

$\forall z_j \in Z$:

Length constraint: $\forall p_i \in P_{/L_j^{-1}} \quad lz_j \geq lp_i$

Capacity constraint: $\sum_{p_i \in P_{/L_j^{-1}}} w_i \geq c_j$

We shall note Λ the set of all possible relations L between posts and zones under the above constraints.

Now, our first disposing criterion concerns the gathering of the posts of the same customer. In order to evaluate the possible partitions P/L^{-1} according to this criterion, we need to introduce an evaluation of the geographical spreading of the posts over the zones. To do so, we define a distance between the location of the posts. If z_j and $z_{j'}$ represent the zones where the posts p_i and $p_{i'}$ are disposed, i.e. we have the correspondences p_iLz_j and $p_{i'}Lz_{j'}$, their distance $d_{ii'}$ is evaluated in the following way:

$d_{ii'} = d_1$ if $z_j = z_{j'}$,
$= d_2$ if z_j and $z_{j'}$ are contiguous,
$= d_3$ if z_j and $z_{j'}$ are separated by r zones ($r>1$),
$= d_4$ if z_j and $z_{j'}$ are in different parks linked by a rolling bridge or a common access road
$= d_5$ if z_j and $z_{j'}$ are in different parks not linked.

with $d_1<d_2<d_3<d_4<d_5$, in order to match our underlying disposing preferences.

Let G'_j and G_j be groups of posts related to a particular customer c_j, where the first set of posts is already in stock and the second has to be disposed. In order to evaluate the resulting spreading of these posts, we construct the corresponding sets of distances:

$Dc_j = \{ d_{ii'} / p_i, p_{i'} \in G_j \wedge i < i' \}$ and

$Dc'_j = \{ d_{ii'} / p_i \in G_j \wedge p_{i'} \in G'_j \}$.

where $D_{ii'}$ represents the distance between the zones z_j and $z_{j'}$ where the posts p_i and $p_{i'}$ are stocked such that we have the relations p_iLz_j and $p_{i'}Lz_{j'}$. The evaluation of the disposing of the posts for customer j is summarized by the following formula:

$$d_c^j = \sum_{(Dc_j \cup Dc'_j)} d_{ii'}$$

And our first criteria may be formulated by the following objective function:

$$\min_{L \in \Lambda} F_c = \sum_j d_c^j$$

A second disposing criterion concerns the gathering of the posts belonging to the same order o_k of a particular customer c_j.

We can reuse the same spreading evaluation as before. Let O'_{jk} and O_{jk} be sets of posts of the same order o_k of customer c_j, that are in stock, respectively have to be disposed. We construct the corresponding set of distances between these posts.

$Do_{jk} = \{ d_{ii'} / p_i, p_{i'} \in O_{jk} \wedge i < i' \}$ and

$Do'_{jk} = \{ d_{ii'} / p_i \in O_{jk} \wedge p_{i'} \in O'_{jk} \}$.

The evaluation of the disposing of posts of all the orders o_k for customer c_j, is summarized by the following formula:

$$d_o^j = \sum_k \left(\sum_{(Do_{jk} \cup Do'_{jk})} d_{ii'} \right)$$

And the second criterion gives rise to the following objective function minimizing the above spreading evaluation for all customers:

$$\min_{L \in \Lambda} F_o = \sum_j d_o^j$$

The overall optimal disposing of the posts considering both criteria may be achieved by a goal programming strategy which minimizes conjointly the sum of the two above objective functions.

As posts of the same order in the parks are more often loaded together for expedition than those related to the same customer, we balance the overall objective function by two priority coefficients r_o and r_c with $r_o >> r_c$ in order to insist on the corresponding minimization. Finally we obtain the following goal-program:

$$\min_{L \in \Lambda} F = r_c \cdot F_c + r_o \cdot F_o$$

Thus our simplified girders disposing problem appears as a fairly simple multicriteria linear location problem, where the decision variables have finite domains.

This will allow us to formulate and solve this problem with the help of the finite computations resources of the CHIP language. As already noticed above, our solution will follow closely the above mathematical formulation.

4.3. Solving the Girders Disposing Problem in CHIP

We must first define the decision variables and their domain.

To identify the posts, we use an index variable `I` with domain `1..N`. This index variable `I` allows us, by means of the `element` predicate (described in section 4.1.), to associate a post with its characteristics like order-number, customer-number, length of its girders and total weight of the post.

In order to identify the disposing zones we use a finite domain variable `J` with domain `1..M` that gives us, in a similar way, an index to the zone's park-numbers, lengths and disposing capacities.

Now, as the correspondence λ between the set of posts and the set of zones is supposed to describe a functional relation, we may express this relation by a set of `N` location variables, which act as our decision variables. For each post p_i, a location variable `Li` represents the number of the zone where it is or will be disposed. Therefore, the domain of these decision variables corresponds to the possible zone's numbers, that is the consecutive integers from `1` to `M`. Let's call `L` the list of our `N` location variables.

```
L = [L1, L2, ..., LN],
L :: 1..M.
```

In order to ensure that the weights, disposed by a correspondence λ onto the zones, do not exceed the disposing capacities of the zones, we define for each zone a charging variable indicating the actual weight disposed onto that particular zone. These charging variables take their values between 0 and the disposing capacity of the corresponding zone.

```
DisposingCap = [Ca1, Ca2, ...., CaM].
Charges = [Ch1, Ch2, ..., ChM],
Ch1::0:Ca1, Ch2::0:Ca2, ..., ChM::0:CaM,
```

In this domain declaration, the ':' symbol specifies domains in which only the minimum and maximum values are interesting. Here the minimum value is used to represent the actual charging of the zone that we initialize to 0. On the other hand, the '..' symbol, used in the location variables declaration, indicates that domain values can be removed inside the indicated bounds.

In order to initialize the actual stock situation, we assume that a list `Disposed` gives the posts `Pi` that are already disposed onto the zones `Zi`. Thus, we have to instantiate the corresponding location variables and to modify the charging capacities of the zones concerned. In order to access the location and the charging variables by their index, we gather these variables in a `LocTerm`, respectively in a `CapTerm`:

```
Disposed = [[P1, Z1], [P2,Z2], ..., [PQ, ZQ]],
Locterm = relation_var(L1, L2, ..., LN),
CapTerm = charging_var(Ch1, Ch2, ..., ChM),
PostWeights = [Wp1, Wp2, ..., WpN],
in_stock(Disposed, LocTerm, CapTerm,
        PostWeights).
```

where the `in_stock` predicate is defined in a recursive way on the list of disposed posts in the following way:

```
in_stock([], _, _, _).
in_stock([[I, J]|Ds], LocTerm, CapTerm,
            PostWeights) :-
      arg(I, LocTerm, LI)
      LI #= J,
      arg(J, CapTerm, ChJ),
      element(I, PostWeights, WI),
      ChJ #>= WI
      in_stock(Ds, LocTerm,CapTerm,
            PostWeights).
```

First, the `arg` predicate associates `I` with the `I`th variable in the `LocTerm`. The domain of this variable `LI` is then constrained to the number of zone `J`. The same `arg` predicate is used to extract the `J`th charging variable out of the `CapTerm` and the corresponding lower bound is restricted to the weight of the `I`th post. This weight is extracted from the `PostWeights` list with the help of the `element` predicate.

This concludes the definition of domains of the decision variables and their data context.

We must now state the access restrictions imposed on the relation `L` by the length constraint.

```
length_constraint(ToBeDisposed, LocTerm,
                  GirderLengths, ZoneLengths).
```

`ToBeDisposed` represents the list of posts that have to be disposed in the parks. This constraint uses the following predicates in order to reduce the domain of each location variable `LI` to the number of zones that are long enough.

```
length_constraint([], _, _).
```

```
length_contraint([I | T], LocTerm,
                 ZoneLengths):-
      element(I, GirderLengths, LpI),
      arg(I, LocTerm, LI),
      element(LI, ZoneLengths, LzLI),
      LzLI #>= LpI,
      length_constraint(T, LocTerm,ZoneLengths).
```

Again the `element` and the `arg` predicates allow us, first to access the length of the girders of post `I`, that is `LpI`, and then to access the corresponding location variable `LI`. This disposing zone designated by `LI` is associated by the `element` predicate with a certain zone length `LzLI`. But this zone length `LzLI` must always be superior or equal to the disposed girders length `LpI`. The symbolic link, instantiated by the `element` predicate, automatically puts such a co-restriction on `LI`.

We are now able eventually to enumerate all feasible correspondences of the set Λ by using the above mentioned `labeling` predicate. But to be feasible, we must respect the capacity constraint. Indeed, if we want to dispose post `I` onto zone `J`, that is instantiate `LI` to the value `J`, we need to know the currently remaining charging capacity of zone `J` in order to see if it still has enough capacity for the post `I`. This is done by the predefined predicated `domain_info` where `Charge` and `CMax` are the boundary values of the charging variable `ChJ`.

```
arg(I, LocTerm, LI),
arg(LI, CapTerm, ChLI),
domain_info(ChLI, Charge, CMax, _, _, _),
```

Now, we can verify the capacity constraint, i.e.

```
element(I, PostWeights, WI),
RestCapacity is CMax - Charge,
WI =< RestCapacity,
```

If `WI` doesn't exceed the `RestCapacity`, then the disposing of the `I`th post onto zone `LI` raises the lower bound of the corresponding charging variable:

```
Charge is Charge + WI,
ChLI #>= Charge.
```

Notice that this part of the program cannot be executed unless the post and the corresponding zone are known, that is the location variable `LI` is actually instantiated to a ground value. We overcome this difficulty by using a delay declaration.

We gather the preceding lines in one predicate call:

```
capacity_constraint(ToBeDisposed, LocTerm,
                    CapTerm, Post_weigths),
```

and we delay its execution until its second argument is instantiated:

```
?- delay capacity_constraint(ground, ground,
                    any, any, any).
```

This concludes the generating of constraints on the set of possible values for the decision variables `L`.

Now we need to formulate our objective function.

For the purpose of the exposition, we concentrate on the construction of the first objective function $F_c = \sum_j d_c^j$.

In order to compute each d_c^j related to one customer, we need to define and sum up the corresponding distances. For each group G_j and G'_j of posts belonging to the same customer, we search for the corresponding groups of location variables G_{Lj} and G'_{Lj}.

Then the recursive `dist_construct` and the `construct` predicates elaborate the sum of the associated sets of distances Dc_j and Dc'_j, each distance between two posts being represented by a domain variable whose boundary values are `D1` and `D5` according to the distance function. The park numbers associated with each zone are supposed to be grouped in the list `ParkNumbers`.

```
dist_construct([], G'Lj, ParkNumbers, 0),
dist_construct([L|GLj], G'Lj, ParkNumbers,
                    D + D' + Ds):-
      [D, D'] :: D1..D5,
      construct(L, GLj, ParkNumbers, D),
      construct(L, G'Lj, ParkNumbers, D'),
      dist_construct(GLj, G'Lj, ParkNumbers,
                    Ds).

construct(L1, [], ParkNumbers, 0).
construct(L1, [L2|Ls], ParkNumbers, D + Ds):-
      element(L1, ParkNumbers, Park1),
      element(L2, ParkNumbers, Park2),
      distance(L1, L2, Park1, Park2, D).
      construct(L1, Ls, ParkNumbers, Ds).
```

Then the `distance` predicate, described below, applies the distance evaluation function. We assume that `D1`, `D2`, `D3`, `D4`, `D5` represent the different distance evaluations allowed between 2 posts:

```
distance(Li, Lj, Parki, Parkj, Dij):-
 Li = Lj,
 Dij is D1, !.   % Same zones
distance(Li, Lj, Parki, Parkj, Dij):-
 Parki = Parkj,
 Abs is abs(Li - Lj),
 Abs = 1,
 Dij = D2, !.    % Contiguous zones
distance(Li, Lj, Parki, Parkj, Dij):-
 Parki = Parkj,
 R is abs(Li - Lj),
 R > 1,
 Dij is D3,!.    % Zones Separated
                        by R>1 zones
distance(Li, Lj, Parki, Parkj, Dij):-
 linked_parks(Parki, Parkj),
 Dij is D4, !.   % Parks linked by a rolling
              bridge or a common access road
distance(Li, Lj, Parki, Parkj, Dij):-
 Dij is D5.        % Parks not linked
```

Again, this predicate must be delayed until its first arguments are instantiated:

```
?- delay distance(ground, ground, any, any,
                    any).
```

By adding the different d_c^j constructed with the preceding predicates we evaluate the objective function F_c for all the customers.

In a similar way, we construct the objective function `Fo` related to all the orders. And the complete solution will be the sum `Rc*Fc + Ro*Fo`, where each term is multiplied by its respective priority coefficient `Rc` and `Ro`. We need a variable `F :: 0:1000000` with a big domain to represent this important sum of distances.

As mentioned above, CHIP provides a higher-order optimization predicate `min_max` that will minimize this objective function on the domain of location variables `L`:

```
min_max(labeling(L), [Fc]).
```

This call will search for the values of the decision variables `L`, which minimize the distances between posts having the same customer and orders, while respecting the constraints and relations introduced before. It uses the above mentioned `labeling` predicate that allows the generation of values from the domain of the location variables.

4.4. *Experimental Results*

The efficiency of our CHIP solution depends on the state of the expedition parks and on the number of posts having the same characteristics. If a lot of zones are unoccupied then an optimal solution will be found quickly. But if the parks are full then it will be slower. This difference in execution time is accentuated whenever the number of posts belonging to the same order or customer grows. In this latter case the sum of all the distances between them becomes important and difficult to minimize.

Our solution was tested on a real disposing problem where 148 posts divided into 38 customers had to be stocked onto 40 unoccupied zones. An optimal solution for this important number of posts would have taken too much time if we had not used a variant of the `indomain` predicate which allows the generation to be started with a specific value. Thus, for each group of posts belonging to the same customer, we search for an appropriate number of zone having enough capacity and appropriate length to receive all these posts (or a big part of them). By starting the labeling process with these numbers, an optimal solution is found in 34.06 seconds.

However, if the availability of the zones is restricted, i.e. there are a lot of posts already disposed onto the zones, execution time takes hours. To solve the problem nevertheless we simplified the problem by optimizing the disposing for each customer and his orders separately. This relaxed solution `Fr` gives very good results in acceptable time (11 seconds for empty parks and 2 hours 8 minutes for full parks).

As explained before, different options of the `min_max` predicate allow us to adapt the searching strategy. Thus, we retried to reach a solution to the complete problem which would be better than the relaxed solution in a maximum of 30 minutes with $\varepsilon = 0,1$.

```
min_max(labeling(L), 0, Fr, 10, 1800).
```

The complete optimal solution, better than the relaxed solution `Fr`, must take value between 0 and this value `Fr`. To dismiss insignificant ameliorations, we limit the search to solutions at least 10 percent better than `Fr` and we limit execution time imperatively to 1800 seconds. In our concrete tests, no such ε-optimal solution could be found.

Therefore, once the global method doesn't succeed rapidly, we can treat each customer separately and find an approximated solution. In order to test if this result is far from the complete optimum, we may use the above variant of the `min_max` predicate to try to ε-ameliorate this approximation in a limited time. Finally, if the problem becomes too difficult, we still have the possibility to divide the posts into smaller production periods.

4.5. *Discussion*

The CHIP solution of the girders disposing problem illustrates the finite domain computation facilities proposed. The program text follows rather closely the mathematical formulation as already observed about the Prolog III solution above. But when programming in CHIP the generation of the constraints on the decision domains and the objective function needs basic Prolog programming knowledge. Because of this problem, new versions of CHIP will provide facilities to formulate cumulative constraints like our disposing capacity constraint in a simpler way.

Nevertheless, the finite domain component of the CHIP system appears as a kind of modelling system for linear integer programming problems. Our CHIP solution is small in size and may easily be modified to adapt the solution to changing specifications. In the same way, it is easy to construct generic abstract solutions for interesting general problems.

The integrated branch-and-bound solver gives satisfactory results if the combinatorial size of the problem can be kept within reasonable limits.

5. Conclusion

We have developed solutions to linear industrial problems by using constraint logic programming languages. The Prolog III and CHIP systems, according to their declarative aspects, allow the modelling of problems in short programs easy to understand and to modify. Moreover, their constraint solving capacities permit finding the optimal solution to constrained linear decision problems.

The general solving capacities of linear rational problems, illustrated in a Prolog III solution to the coils selection, showed performances comparable to those of traditional MP solvers. But a lack of information return from the integrated simplex solver appeared. It is not possible to access dual solutions for rounding purposes, for instance.

Solving linear integer problems, like for instance our girders disposing solution, showed the interesting facilities of finite domain computations in CLP systems like the CHIP environment, but good Prolog programming is welcome.

To conclude, we would say that CLP systems should not be dismissed as potential modelling and solving tools for linear MP problems.

6. References

[1] Bisdorff, R., and Gabriel, A., "Industrial linear optimization problems solved by constraint logic programming", *First International Conference on the Practical Application of Prolog*, London, 1992.

[2] Bisdorff, R., Gabriel, A., and Laurent, S., "Optimisation linéaire en Prolog III", *Twelfth International Conference Avignon92*, Avignon, 1992.

[3] Bisdorff, R., and Laurent, S., "Industrial disposing problem solved in CHIP", *Paper submitted for communication at the Tenth International Conference on Logic Programming*, Budapest, 1993.

[4] Colmerauer, A., "Opening the Prolog III Universe", *Byte Magazine*, Aug. 1987.

[5] Colmerauer, A., "An introduction to Prolog III", *Communication of ACM*, vol. 33, July 1990.

[6] Dincbas, M., Van Henteryck, P., Simonis, H., Aggoun, A., and Graf, T., "Applications of CHIP to industrial and engineering problems", *Proceedings of the First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 1988.

[7] Dincbas, M., Van Henteryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F., "The constraint logic programming language CHIP", *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, 1988, 693-702.

[8] Dincbas, M., Van Henteryck, P., Simonis, H, "Solving large combinatorial problems in logic programming", *J. Logic Programming*, New York, vol. 8, N° 1 &2, January/March, 1990.

[9] Jaffar, L., and Lassez J.-L., "Constraint Logic Programming", *Conference on Principles of Programming Languages*, Munich, 1987.