SOLVING A CUTTING-STOCK PROBLEM WITH THE CONSTRAINT LOGIC PROGRAMMING LANGUAGE CHIP

M. DINCBAS AND H. SIMONIS
COSYTEC S.A.
4 rue Jean Rostand, 91893 Orsay, France
P. VANNENTARYCK
Brown University
Providence, Rhode Island, U.S.A.

Abstract—CHIP is a new constraint logic programming language combining the declarative aspect of logic programming with the efficiency of constraint manipulation techniques. In the present paper, we show an application of CHIP to a two-dimensional cutting stock problem. This problem is highly combinatorial and is generally solved by specific programs written in procedural languages. We present two approaches to solve this problem in CHIP and compare them with the standard ones. It turns out that, although CHIP greatly simplifies the problem statement, it is comparable in efficiency to specialized programs. Of particular interest is the ability to use symbolic constraints to prune the search space.

1. INTRODUCTION

Many real-life problems are constrained search problems. Most of them belong to the class of NP-complete problems [2]. Logic Programming provides a powerful language for a logical formulation of these problems. Its nondeterministic computation liberates the user from the tree-search programming. However, due to the inefficiency of their search procedure based on the generate and test paradigm, logic programming languages like Prolog [3] have been used until now to solve only toy problems.

Extending logic programming with constraint-handling techniques can improve its behaviour [4–11]. The general ideas behind these extensions are the use of some mathematical techniques to solve numerical constraints and the use of consistency techniques to solve symbolic constraints in order to prune the search space in an a priori way, instead of using them as tests, as in the case of the generate and test method.

In this paper, we show an application of the constraint logic programming language CHIP to a real-life cutting-stock problem occurring in a furniture factory. The problem is the cutting of two dimensional shelves of various sizes from standard wood boards minimizing the total waste. In the following, we first give a brief overview of CHIP, where we introduce informally numerical and symbolic constraint-handling on domain-variables and optimization techniques used to solve the problem. We then present the cutting stock problem which is solved in two steps. In the first step, all configurations satisfying technological constraints are computed with a nondeterministic logic program. In the second step, we search for an optimal selection of the configurations minimizing the total waste. For this second step, we show two different formulations of the problem in CHIP: an integer linear programming approach and a symbolic constraint programming approach. We finally compare these two approaches showing that the second approach, which exploits the unique character of constraint logic programming, is better than the first one. It is also more efficient than a specialized Fortran program developed especially for this problem [11].

This paper is a revised version of Ref. [1]. This work was performed while the authors were at the European Computer-Industry Research Centre (ECRC) in Munich, Germany.
We would like to thank H. Gallaire, A. Aggoun, F. Berthier, T. Graf and A. Herold for many fruitful discussions.
CHIP is a constraint logic programming language developed at the European Computer-Industry Research Centre (ECRC) in Munich, Germany [12]. It originates from the concept of active constraints [13], and extends usual logic programming languages by providing constraint-solving techniques on three computation domains:

- finite domain terms;
- Boolean terms;
- rational terms.

Finite domain terms are constructed from domain-variables [14], i.e., variables ranging over a finite set of constants. A particularly interesting class of domain-variables are those which range over natural numbers. CHIP has the ability to cope with arithmetic terms over such domain-variables. These terms are constructed from natural numbers, domain-variables over natural numbers, and the usual operators +, −, ⋅ and /.

CHIP provides a large variety of constraints on domain-variables, not only arithmetic, but also symbolic and user-defined constraints. This will be discussed below in more detail. Constraints on domain-variables are solved by consistency checking techniques [9,10], an important paradigm emerging from Artificial Intelligence research.

Boolean terms are built from truth values (0 and 1), from constants (atoms), from variables and from the Boolean operators and, or, xor, not, etc. Equality constraints over Boolean terms are solved through a unitary unification algorithm based on variable elimination [6,15].

Rational terms are linear terms over rational numbers and rational variables (i.e., variables which take their values in rational numbers). Equations, inequalities and disequations (i.e., ≠) over rational terms are solved through a symbolic simplex-like algorithm.

Besides the above extensions, CHIP also contains delay and demon mechanisms, which allow a data driven computation flow.

With these extensions, we have solved in CHIP several real-life problems from Operations Research [9,16] and Circuit Design [8] with an efficiency comparable to specific programs written in procedural languages. See [17] for a review of some applications of CHIP.

The approach taken in CHIP is related to work in the Prolog-III (at the University of Marseille, France) and the CLP (at IBM Yorktown Heights, U.S.A.) projects which are also aiming at the introduction of constraint-solving techniques inside logic programming. Prolog-III [4] uses a simplex-like algorithm to solve linear equations and inequations on rational numbers. In addition, it provides a theorem proving method based on saturation for solving Boolean constraints. CLP [7] defines a formal framework providing a theoretical basis for building logic programming languages based on constraint-solving which generalizes the notion of unification. An instance of this scheme for handling linear equations and inequations on real numbers has been implemented in CLP[8] [18].

The application presented here only makes use of the finite domain extension and uses both numerical and symbolic constraints.

2.1. Constraint-Handling on Finite Domains

In the following we will briefly present (through examples) several constraint-handling techniques on finite domains which are used in this paper to solve the cutting stock problem: numerical constraint-handling, symbolic constraint-handling and search for an optimal solution.

2.1.1. Numerical Constraint-Handling

CHIP can deal with linear equations (noted as =), inequalities (noted as ≤, <, ≥, >) and disequations (noted as ≠), over natural numbers [6,9,12]. This kind of numerical constraints is handled in CHIP by reasoning about variation intervals. For instance, assume we have three domain-variables X, Y and Z with domain {0, ..., 9} and that the following inequalities hold:

\[ 2 \cdot X + 3 \cdot Y + 9 \cdot Z \leq 8, \]
\[ X + Y + Z \geq 3. \]

From the first inequality, CHIP deduces that \( X \leq 4, Y \leq 2, \) and \( Z = 0. \) These new constraints are propagated into the second inequality which gives \( X \geq 1. \) Thus the domains of the variables are
reduced, respectively, to \{1, \ldots, 4\}, \{0, 1, 2\} and \{0\}. Such constraint-handling reduces drastically the search space.

2.1.2. Symbolic Constraint-Handling

Part of the originality of CHIP is the handling of symbolic (e.g., non-numerical) constraints. In principle, any logic program can be considered as a constraint, and CHIP is able to handle them in an active way [9,10]. Symbolic constraints are solved with consistency checking techniques. As far as we know, CHIP is the only logic language that allows user-defined constraints and handles them actively.

Some frequently used constraints are provided as primitives in CHIP. One of the most useful symbolic constraint in CHIP is

\[
element(N, L, V),
\]

where \(N\) and \(V\) are domain variables and \(L\) is a list of constants. The meaning of this predicate is the \(N\)th element of the list \(L\) is \(V\). This predicate allows to express conditional and functional constraints. In fact it is not only seen as a function which gives the value \(V\) when \(N\) is known, but it is treated as an adirectional constraint which makes a correspondence between \(N\) and \(V\) through \(L\). When a new constraint is added onto \(N\) (resp., onto \(V\)), it is immediately propagated onto \(V\) (resp., onto \(N\)). For instance, assume that we have the constraint

\[
element(X, [5,3,9,1,4], C)
\]

and that the domains of \(X\) and \(C\) are, respectively, \{1, \ldots, 5\} and \{1,3,4,5,9\}. When, during the computation, we get a constraint \(C \geq 4\), it is immediately propagated by \(element\), which finds \(X \neq 2\) and \(X \neq 4\). The domains of \(X\) and \(C\) are now \{1,3,5\} and \{4,5,9\}. If, further in the computation, we get a new constraint \(X \leq 3\), then this constraint will be reconsidered, reducing the domain of \(X\) to \{1,3\} and that of \(C\) to \{5,9\}. This kind of demon-driven constraint propagation is the basis of the CHIP language.

2.1.3. Search for an Optimal Solution

Combinatorial problems often require to find a solution which optimizes (i.e., minimizes or maximizes) an objective function. CHIP provides some higher-order predicates for that purpose [9,12]. In the following, we will only consider the predicate

\[
\text{minimize}(P, \text{Function}),
\]

which is used in the cutting-stock problem. The first argument, \(P\), is a non-deterministic predicate. The second argument \(\text{Function}\) is a variable or a linear term over domain variables. This meta-predicate finds a solution of \(P\) such that \(\text{Function}\) is minimal. The search for an optimal solution uses a kind of depth-first branch and bound technique [10]. For instance, when a solution with a value \(F_0\) is found, the constraint \((\text{Function} < F_0)\) is dynamically added to the program. While searching a new solution, all branches of the search space where the value of \(\text{Function}\) is greater than \(F_0\) are pruned. The optimal solution is achieved when no new solution better than the most recent one can be found.

3. THE CUTTING-STOCK PROBLEM

3.1. Problem Statement

The example presented here is taken from [11]. It is a two dimensional cutting stock problem occurring in furniture manufacturing. Two hundred large pieces of wood have to be cut into differently sized smaller shelves (see Figure 1).

The boards are cut in lots of 50 pieces each to simplify the set up the sawing machine. We therefore can divide all data by 50 and deal only with the 4 kind of boards required. For each type of shelf there is a certain demand, i.e., we need at least 4 pieces of shelf number 1, 16 of shelf 2, and so on. About 85% of the available area of the boards is required for these shelves.
The problem is further restricted by technological constraints imposed by the sawing machine. The machine has one set of saws working in the horizontal direction and two sets working in the vertical direction. Each board is first cut horizontally into different bands. Each band is then cut vertically in one of two possible ways. The result is a cutting plan for each board called a configuration. In order to reduce the number of possible configurations, there is an upper limit for the waste in each operation. The factory allows at most 10% waste for the horizontal cut and at most 4% for each vertical cut. As an additional constraint, the number of shelves with the same size per cutting is limited (otherwise we would construct many configurations containing many small pieces but no large ones).

For cutting stock problems in general, the number of configurations is rather large. Due to the strong technological constraints, the problem considered here allows only 72 possible configurations. Among these, we have to select 4 configurations which contain the different types of shelves in the required numbers. We can have other objectives as well. We can add, for example, the requirement to find a solution which reduces waste below a certain percentage or to find a solution which minimizes total waste. In the rest of this paper, we will search for an optimal solution minimizing the total waste.

![Figure 1. Board and shelves in the cutting-stock problem.](image)

3.2. Problem Description and Resolution in CHIP

The program to solve the problem consists essentially of two parts. In the first part, all admissible configurations are computed. This takes into account the constraints on horizontal and vertical cuts and rejects all cuttings which create too much waste or too many small pieces. In this example, there are 72 possible configurations with a relative waste between 0.89% and 17.90% (see Figure 2). The program to generate all the configurations is rather simple, since it uses the nondeterministic feature of logic programming languages. We only have to describe how to get admissible solutions. The search is automatically handled by backtracking, and we obtain all configurations using the higher-order \texttt{bagof} predicate.

The second part searches for an optimal choice of the configurations in order to minimize the total waste satisfying the customer demand. We now study this step in detail.

Remember that there are 6 different types of shelves: S1, S2, S3, S4, S5 and S6. In Figure 3 one of the 72 configurations (configuration number 22) is displayed.

Let $N_{i,j}$ denote the number of shelf $i$ in the configuration $j$. From the Figure 3 we can see that:

\[ N_{1,22} = 4, \ N_{2,22} = 4, \ N_{3,22} = 4, \ N_{4,22} = 2, \ N_{5,22} = 2, \ N_{6,22} = 0. \]

Let $L_i = [N_{i,1}, \ldots, N_{i,j}, \ldots, N_{i,72}]$ be the list of the number of shelf $i$ in all configurations. The configuration generation program gives us the lists $L_1, L_2, L_3, L_4, L_5$ and $L_6$. 

According to the customer demand, the required numbers for each shelf in the final solution is given in the following table.

<table>
<thead>
<tr>
<th>Shelf (S_i)</th>
<th>S_1</th>
<th>S_2</th>
<th>S_3</th>
<th>S_4</th>
<th>S_5</th>
<th>S_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number (D_i)</td>
<td>4</td>
<td>16</td>
<td>4</td>
<td>16</td>
<td>32</td>
<td>12</td>
</tr>
</tbody>
</table>

Note that these numbers are not imperative in the sense that we can exceed the quotas. The surplus can be used for further orders [11]. The problem to solve at this stage can now be defined as:

Select 4 (not necessarily different) configurations out of 72, satisfying the demand of the customer and minimizing the total waste.

In the following, we will show two different approaches to model and solve this problem in CHIP.

3.2.1. Integer Linear Programming Approach

In this approach, we introduce a variable $x_j$ for each configuration $j$, which denotes the number of occurrences of this configuration in a solution. Since we should have 4 configurations in a solution, $x_j$ is between 0 and 4. Furthermore, we observe that we cannot have 4 times the same configuration in a solution. Therefore, the domain of the variables $x_j$ is $\{0, \ldots, 3\}$. The problem is then to find an assignment of values from the domain $\{0, \ldots, 3\}$ to $x_1, \ldots, x_{72}$, satisfying the
constraints and minimizing the cost. The constraint, which says that the number of configurations in a solution is 4, is expressed by the following equality

$$\sum_{j=1}^{72} X_j = 4.$$ 

Let $N_{i,j}$ denote the number of shelf $i$ in the configuration $j$. The term

$$\sum_{j=1}^{72} N_{i,j} \times X_j$$

represents the number of occurrences of shelf $i$ in the solution. This number must be greater than or equal to the demand of the customer ($D_i$). This constraint is expressed by the inequality

$$\sum_{j=1}^{72} N_{i,j} \times X_j \geq D_i, \quad (1 \leq i \leq 6).$$

The above 7 numerical constraints express all the constraints of the problem. What remains to state is the objective function. Let $C_j$ denotes the cost (i.e., the waste) of the configuration $j$. This is computed in the first part of the program. Thus, the term

$$\sum_{j=1}^{72} C_j \times X_j$$

represents the total cost of the solution. The objective of the problem can then be expressed as

$$\text{minimize} \sum_{j=1}^{72} C_j \times X_j.$$ 

In the following, we give the sketch of the corresponding program in CHIP (the second part concerning the optimal selection of the configurations). Note that in the actual program these constraints are generated automatically from the first part.

```
progl(Xs, Cost) :-
cuttingl(Xs, Cost),
minimize(labeling(Xs), Cost).

cuttingl([X1, ... , X72], Cost) :-
[X1, ... , X72] :: 0..3,
X1 + ... + X72 = 4,
N1,1 \times X1 + ... + N1,72 \times X72 \geq 4,
N2,1 \times X1 + ... + N2,72 \times X72 \geq 16,
N3,1 \times X1 + ... + N3,72 \times X72 \geq 4,
N4,1 \times X1 + ... + N4,72 \times X72 \geq 16,
N5,1 \times X1 + ... + N5,72 \times X72 \geq 32,
N6,1 \times X1 + ... + N6,72 \times X72 \geq 12,
Cost = C_1 \times X1 + ... + C_{72} \times X_{72}.
labeling([]).
labeling([X|Y]) :-
indomain(X),
labeling(Y).
```
In the above program, the predicate cutting1 sets up the constraints of the problem, and the labeling procedure assigns values to variables $X_j$, whose domain is $\{0, \ldots, 3\}$. $Wi,j$ and $Cj$ are parameters computed in the first part of the program. The higher-order predicate minimize is used to find a solution of the labeling predicate in order to minimize Cost. This is a typical integer linear programming problem [20], which can be handled as it is in CHIP.

5.2.2. Symbolic Constraint Programming Approach

In the above description of the problem, we have tried to find the values of the variables $X_j$, i.e., the number of occurrences of each configuration in the solution. Since this leads to a nice mathematical model in terms of an integer linear programming problem, this is the usual way of modeling this problem [11]. We now turn to another way of stating the problem which exploits the unique character of constraint logic programming in general and of CHIP in particular.

Basic Approach

Instead of finding the number of occurrences of each configuration in the solution (which must contain 4 configurations), we will try to find directly which configurations are selected in the solution. Let us call $SC_1, SC_2, SC_3$ and $SC_4$ the variables denoting the 4 selected configurations in the solution. The domain of these variables is obviously $\{1, \ldots, 72\}$. In this case, the problem becomes:

Find an assignment of values from the domain $\{1, \ldots, 72\}$ to $SC_1, SC_2, SC_3$ and $SC_4$ satisfying the constraints and minimizing the cost.

The expression of the constraints and the cost in this model is a bit more complicated, since there is no direct mathematical formulation of them. Let us see how can we formulate the cost function.

$$Cost = Cost_1 + Cost_2 + Cost_3 + Cost_4.$$  

$Cost_k$ denotes the cost of the selected configuration $SC_k$. Therefore, we need a constraint to make the correspondence between these two variables. This can be achieved in CHIP by the abovementioned element constraint. The constraint

$$element(SC_k, [C_1, \ldots, C_{72}], Cost_k)$$

expresses the relationship between $SC_k$ and $Cost_k$ through the list of costs of the configurations $C_j$. Remember that $C_j$ is known for all configurations $j$. If $SC_k$ is given the value $j$, then $Cost_k$ is equal to $C_j$. This constraint is not only used to give a value to $Cost_k$ when the value of $SC_k$ is known but also, and especially, to propagate constraints from $SC_k$ to $Cost_k$, and vice-versa. Since there are 4 selected configurations in a solution, we will have 4 constraints relating these configurations to their costs.

The constraints expressing the demand of the customer will be formulated in the following. Let $M_i,k$ denote the number of occurrences of shelf $i$ in the selected configuration $k$. The term

$$\sum_{k=1}^{4} M_i,k$$

gives the total number of occurrences of shelf $i$ in the solution. The constraint

$$\sum_{k=1}^{4} M_i,k \geq D_i$$

expresses the user requirement concerning shelf $i$. Since we have 6 shelves, we will have 6 constraints of this type. Now how to compute the values of $M_i,k$? For this purpose, we will use again the element predicate. The constraint

$$element(SC_k, [M_{i,1}, \ldots, M_{i,j}, \ldots, M_{i,72}], M_{i,k})$$
expresses the correspondence between $SC_k$ and $N_{i,k}$ through the list of the number $N_{i,j}$ of shelf $i$ in the configuration $j$. Remember that the parameters $N_{i,j}$ have been computed in the first part of the program. Since there are 6 shelves and 4 selected configurations in a solution, we will have 24 constraints of this type. Thus, the above presented 34 constraints (28 symbolic and 6 numerical) express all the constraints of the problem. They can be used to solve the problem. Although the above approach can solve the problem, it is still possible to improve it in several ways which are now studied.

**Finding a Good Solution**

A first improvement consists in finding a good solution, which helps pruning the search space early. For this purpose, a good strategy is to select the configuration with small costs at the beginning of the labeling procedure. This can be achieved very easily by a renaming of the configurations, i.e., reorder the costs in ascending order and give the number 1 to the configuration with the smallest cost, and so on, until the number 72 to the configuration with the greatest cost. Thus, the elements of the list $[C_1, \ldots, C_2, \ldots, C_{72}]$ are now in ascending order. Note that this heuristics is only used to speed up the search but does not endanger the finding of the optimal solution.

**Exploiting Symmetries**

The second improvement concerns the symmetrical aspect of the problem formulation in this approach. Indeed, the 4 selected configurations, $SC_1$, $SC_2$, $SC_3$, and $SC_4$, play a perfect symmetrical role in a solution $S$. Suppose we have a solution $S = [1, 2, 46, 51]$. Any permutation of this solution (for instance, $S = [46, 1, 51, 2]$) is also a solution. In order to avoid to consider these symmetrical solutions, we can impose an order on the selected configurations. This can be expressed by the constraints

$$SC_1 \leq SC_2,$$
$$SC_2 \leq SC_3,$$
$$SC_3 \leq SC_4.$$

Note that there is no such symmetry to exploit in the first approach.

**Using Dichotomic Search**

The last possible improvement we present is to change the labeling procedure. The idea is to split the possible values for the variables, instead of giving them values. This makes possible to reject a set of inconsistent values at once. The basic step of this labeling is an iteration on all variables. For each of them, we split its domain into two parts. The following clauses are a simple description of this central step.

```prolog
splitting([]).

splitting([X|Y]) :-
    split(X),
    splitting(Y).

split(X) :-
    midvalue(X, Mid),
    chose(X, Mid).

choose(X, Mid) :-
    X =< Mid.
choose(X, Mid) :-
    X > Mid.
```
This step is iterated until all variables get instantiated. Splitting the domain of a variable is a nondeterministic step. Basically, we chose the middle value in the domain and specify that the variable is either smaller or equal to (in the first case), or greater than (in the second case), this value.

**Sketch of the Final Program**

We give, in the following, the sketch of the corresponding program in CHIP. Again, in the actual program, these constraints are generated automatically from the configurations.

```
prog2(Xs, Cost) :-
cutting2(Xs, Cost),
    minimize(labeling(Xs), Cost).
```

```
cutting2([SC1, ..., SC4], Cost) :-
    [SC1, ..., SC4] :: 1..72,
    SC1 ≤ SC2,
    SC2 ≤ SC3,
    SC3 ≤ SC4,
    element(SC1, [C1, ..., Cj, ..., C72], Cost1),
    ... element(SC4, [C1, ..., Cj, ..., C72], Cost4),
    element(SC1, [N1,1, ..., N1,j, ..., N1,72], M1,1),
    ... element(SC4, [N1,1, ..., N1,j, ..., N1,72], M1,4),
    M1,1 + M1,2 + M1,3 + M1,4 ≥ 4,
    ... M6,1 + M6,2 + M6,3 + M6,4 ≥ 12,
    Cost = Cost1 + Cost2 + Cost3 + Cost4.
```

3.2.3. Comparison of the Two Approaches

Let us now compare the two approaches to solve this problem in CHIP. The first approach uses the (pure) Integer Linear Programming feature of CHIP. This is the usual way to solve this problem in Operations Research. In fact, since the domain of the variables is limited to \( \{0, \ldots, 3\} \), the problem is usually transformed into a problem of 0-1 variables, by putting

\[
X_j = 2 \cdot Y_j + Z_j,
\]

(where the domain of \( Y_j \) and \( Z_j \) is \( \{0, \ldots, 1\} \)) [11]. Therefore, 0-1 Integer Programming techniques can be used. Note also that linear programming methods like simplex cannot be used for this problem because of the integrality constraints on the variables and in fact they yield a very poor approximation in this example due to the small size of the domains. The theoretical complexity of this first approach is \( 4^{72} \) (72 variables with domain \( \{0, \ldots, 3\} \)) that is about \( 10^{43} \). This program finds the first solution with a total cost of 3325 (which corresponds to 8.31% of waste) after 33
The optimal solution with a cost 1658 (i.e., 4.14% of waste) is found after 322 seconds. The proof of the optimality requires an additional 13 seconds. The total execution time is then about 6 minutes. Using dichotomic search for this formulation does not lead to any improvement.

The second approach is a more natural way of formulating the problem, as it takes into account the particularities of the problems. It amounts to find which configurations are selected in a solution rather than to find the number of occurrences of each configuration in a solution. For that purpose, the second approach exploits the unique feature of CHIP to handle symbolic constraints. While numerical constraint-handling in CHIP brings a plus to logic programming, symbolic constraint-handling brings a plus in Operations Research. The theoretical complexity of this approach is $72^4$ (4 variables with domain $\{1, \ldots, 72\}$), that is, about $10^7$. Without using the dichotomic search, CHIP finds a first solution with a waste of 0.47% in 3 seconds. The optimal solution is reached after 63 seconds, and the proof of optimality requires 39 seconds. Using dichotomic search, CHIP finds the first solution (which is also the optimal one) in 1.5 seconds and the proof of optimality requires another 1.5 seconds.

Table 1 summarizes these results. A1 represents the first approach, A21 represents the second approach without dichotomic search, and A22 represents the second approach with the dichotomic search. F, O, P and T, respectively, represent the times to find the first solution, to find the optimal solution, to prove optimality and the total time. Finally, note that, the time given in the original paper of Costa [11] to find a solution with a waste of 6.44% (i.e., not the optimal one) is 20 seconds on an IBM 370/168 (a machine faster than a VAX/785) with a specific Fortran program developed to solve this problem.

### Table 1. Computation results for the cutting-stock problem.

<table>
<thead>
<tr>
<th>Approaches</th>
<th>F</th>
<th>O</th>
<th>P</th>
<th>T = O + P</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>33 sec.</td>
<td>322 sec.</td>
<td>13 sec.</td>
<td>335 sec.</td>
</tr>
<tr>
<td>A21</td>
<td>3 sec.</td>
<td>66 sec.</td>
<td>39 sec.</td>
<td>105 sec.</td>
</tr>
<tr>
<td>A22</td>
<td>1.5 sec.</td>
<td>1.5 sec.</td>
<td>1.5 sec.</td>
<td>3 sec.</td>
</tr>
</tbody>
</table>

4. CONCLUSION

In this paper, we have presented an application of CHIP to a real-life, cutting stock problem in a furniture factory. We have shown the description and the resolution of the problem using the constraint logic programming features of CHIP. The first part of the program, which computes all possible configurations, benefits from the declarative statement and nondeterministic computation features of logic programming. For the second part of the program, which computes an optimal selection of the configurations, two different formulations have been presented: an integer linear programming approach and a symbolic constraint programming approach. The first approach uses the numerical constraint handling mechanism of CHIP. The second approach, which exploits the unique feature of CHIP to handle symbolic constraints, has been shown not only more natural than the first one but also more efficient.

CHIP shows that logic programming languages can be extended by constraint-solving techniques and sophisticated search procedures, providing a powerful problem-solving tool for a large range of application domains. Programs are concise, easy to write and to modify, thus providing a flexible tool as an alternative to specific programs.

### References