The Knowledge Engineering Review, Vol. 16:1, 2001, 5–23
Printed in the United Kingdom

Constraint programming and maths programming

JEAN-FRANÇOIS PUGET[1] and IRVIN LUSTIG[2]

[1] *ILOG, 9 av. Verdun, 94250 Gentilly, France*
[2] *LOG, 1080 Linda Vista Ave., Mountain View, CA 94043, USA*

Abstract

Maths programming (MP) and constraint programming (CP) are two techniques that are able to solve difficult industrial optimisation problems. The purpose of this paper is to compare them from an algorithmic and a modelling point of view. Algorithmic principles of each approach are described and contrasted. Some ways of combining both techniques are also introduced.

1 Introduction

The simplex method for linear programming was invented by George Dantzig in 1947, and described in a paper entitled “Programming in a linear structure” (Dantzig, 1948). Fifty years later, linear programming is now a strategic technique used by thousands of businesses trying to optimise their global operations. In the mid-1980s, constraint programming was developed as a computer science technique, by combining developments in the artificial intelligence community with the development of new computer programming languages. Fifteen years later, constraint programming is now being seen as an important technique that complements traditional mathematical programming technologies as businesses continue to look for ways to optimise their business operations.

Developed independently as a technique within the computer science literature, constraint programming is now getting attention from the operations research community as a new and sometimes better way of solving certain kinds of optimisation problem. This paper presents a comparison between those two approaches, both from the algorithmic side and from the modelling side.

The two approaches are used for solving problems that occur in various industrial areas, such as production scheduling, transportation planning, production planning, timetabling, crew scheduling, car sequencing and so on. For any of these problems, the goal is to determine a set of decisions while satisfying various requirements (or constraints). For instance, decisions can be the starting time of production activities or the order in which cars should be produced. Often an objective function is given, e.g. the cost of production that must be minimised or the the total production that should be maximised.

These qualities define a class of problems that can be represented by a set of decision variables, the set of constraints that must be satisfied, and the appropriate objective function. The representation of a problem using decision variables and constraints is often called a model of the problem.

2 Maths programming

Maths programming started with the study of optimisation problems with linear constraints.

2.1 Linear constraints

A constraint is said to be *linear* if it is of the form

$$\sum_j c_j x_j \in [l, u] \tag{1}$$

where the c_j are numbers and the x_j variables.

For example, the following model contains 3 decision variables and 3 linear constraints.

$$\text{Maximise} \quad x + y + z \tag{2}$$
$$\text{subject to} \quad x + y \leq 1 \tag{3}$$
$$y + z \leq 1 \tag{4}$$
$$x + z \leq 1 \tag{5}$$
$$x, y, z \geq 0. \tag{6}$$

The optimal solution to the problem is $x = 0.5$, $y = 0.5$ and $z = 0.5$ for a value of the objective function equal to 1.5.

When the decision variables can take any real number and all the constraints are linear, we say that the problem is a *linear program* (LP). Some very efficient algorithms exist that are able to solve linear programs. The first one, called the simplex algorithm, was invented by Dantzig in 1947, and has been improved tremendously over the last 50 years. Although not a provably polynomial time algorithm, the simplex method is extremely efficient in practice. In the last 20 years, other types of algorithm have been proposed in order to solve LPs, including the elipsoid method (Khachiyan, 1979), Karmarkar's algorithm (Karmarkar, 1984) and the Newton barrier algorithm (Lustig *et al*., 1994). These algorithms all have a polynomial time running time complexity. The Newton barrier algorithm has been shown to be quite effective in practice. Note that for LP solvers, the measure for the size of the problem is not only the number of variables and constraints, but also the sparsity. In other words, the most relevant measure is the number of non-zero coefficients in the constraints of the problems, i.e. the number of occurences of variables in constraints. A dense problem, i.e. a problem where almost all variables appear in each constraint, is much harder to solve than a sparse one, i.e. one in which each constraint involves a very small number of variables. Both the simplex method and the barrier algorithm are applied in practice to even extremely large sparse problems, involving millions of variables and constraints, and these can be solved to optimality in a reasonable amount of time.

2.2 Mixed integer constraints

In many industrial optimisation problems the decisions that one can take are discrete in nature. For instance, one must produce an integer number of pieces in a factory, or one must use an integer number of workers. Those problems can often be expressed using variables that can only take integer values and linear constraints. In this case we say that we have a Mixed Integer Problem (MIP). MIPs are usually much harder to solve than LPs. Indeed, from a theoretical perspective, MIP is NP-complete, which means that no polynomial time algorithm is likely to exist for solving them.

For instance, the following problem is a MIP because the decision variables must take integer values:

$$\text{Maximise} \quad x + y + z \tag{7}$$
$$\text{subject to} \quad x + y \leq 1 \tag{8}$$
$$y + z \leq 1 \tag{9}$$
$$x + z \leq 1 \tag{10}$$
$$x, y, z \geq 0 \tag{11}$$
$$x, y, z \text{ integers}. \tag{12}$$

Current MIP solvers (systems that solve MIPs) use a two-tier architecture. The first tier is an LP solver, for instance a simplex algorithm. The LP solver is used with the *LP relaxation* of the problem at hand. This relaxation is obtained by transforming all integer decision variables into continuous variables, which are then represented on a computer as floating point, i.e. by removing integrality constraints. For instance, the LP relaxation of the above example is obtained by removing the last constraints.

The LP relaxation is then solved to optimality. Several outcomes can occur:

- The LP has no solution. Then the MIP has no solution.
- The LP is not bounded, i.e. there are solutions with arbitrary large values for the objective. An example of an unbounded LP is the maximisation of a variable without any constraints. In such a case, the MIP is also unbounded.
- The LP has a solution, and all the variables that should be integer in the MIP have integer solutions in the LP. Then the optimal solution to the MIP is found.
- The LP has a solution, and some variable that should take an integer value in the MIP has a non-integer value in the LP. In such a case the MIP has not been solved yet.

In order to illustrate the latter case, let us look at our example again. Solving the LP relaxation results in a solution of cost 1.5, and following values for the decision variables $x = 0.5$, $y = 0.5$ and $z = 0.5$. The value of x is not integer. In such cases, the MIP solver will explore two possibilities: either $x \leq 0$ or $x \geq 1$. For each possibility, a copy of the problem is created, and one of the two constraints is added to it. The purpose of those additional constraints is to partition the search space into two regions, each one being easier to solve. Indeed, each of the two constraints on x does not rule out any solution to the problem, but they rule out the solution of the LP where $x = 0.5$. The MIP solver then works on each of the two subproblems recursively. This way of creating two subproblems is called *branching* on the value of variable x.

The same process is applied recursively on each subproblem, resulting in the traversal of a tree. The MIP solver then explores this tree looking for the best solution. Without loss of generality, assume the problem is a minimisation problem. Each time a solution to the MIP is found, its objective value is stored. This value is used as a bound for the rest of the search. Indeed, we are only interested in solutions that are greater than the one already found. Each time the LP relaxation of a node is solved, its objective value is compared to the best bound so far found. If it is lower, then the node cannot lead to a better solution, and it is discarded. This kind of tree search is called *branch and bound*, reflecting the use of both branching and best bounds.

The order in which nodes are explored in the tree is extremely important for the sake of efficiency. Modern MIP solvers allow various form of control over the ordering of nodes and the way a node is expanded into subnodes, e.g. MINTO (Nemhauser *et al.*, 1994). It is even possible to exploit the structure of the problem to be solved, for instance by providing special ways to branch on variables, like special ordered sets (Bockmayr & Kasper, 1998). Branching is not necessarily binary, i.e. more than two subproblems can be created when branching in a problem.

Besides advanced branching, modern MIP solvers use a wide variety of techniques to efficiently solve MIPs including, but not limited to, heuristics and cutting planes. It is beyond the scope of this paper to review all of them, but we can give some hints and examples as to why those techniques are useful.

A heuristic in a MIP setting is a procedure that tries to find an integral solution starting from a solution of the LP relaxation. Several techniques have been devised, but the general idea is often a variation of the rounding approach. Given a solution of the LP relaxation, the following process is applied: let S be the set of integer variables whose value is not integer. If S is empty, then we are done, an integral solution is found. If S is not empty, select a subset S' of S. For each variable of S', fix it to the nearest integer value, and solve the resulting LP. If the LP has a solution, the same process is applied recursively. If the LP has no solution, then the heuristic stops without a solution.

A cutting plane (a *cut*, in short) is a linear constraint that removes no integral solution. For instance, in our example, there are 4 feasible solutions to that problem:

$$x=0 \quad y=0 \quad z=0$$

$$x=1 \quad y=0 \quad z=0$$

$$x=0 \quad y=1 \quad z=0$$

$$x=0 \quad y=0 \quad z=1.$$

The linear relaxation has one optimal solution:

$$x=0.5,\ y=0.5,\ z=0.5.$$

That solution is not a feasible solution of the full problem. However, all the integer solutions satisfy the following constraint:

$$x+y+z\leq 1.$$

That constraint removes no feasible solution but it removes the optimal solution of the relaxation. In other words, this is a cut. When this cut is added to the problem, an optimal solution for the linear relaxation is

$$x=1,\ y=0,\ z=0.$$

This is also a solution of the original problem, therefore it is an optimal solution of the original problem.

There is a wide range of cuts, some fairly general, others specific to particular types of constraint. It is beyond the scope of this paper to review all of them and we refer the reader to (Nemhauser & Wolsey, 1988; Wolsey, 1998). The two techniques we have outlined, heuristics and cutting planes, can be applied at any node in the branch-and-bound tree. Usually they are applied at the root node only.

Maths programming encompasses further classes of problem beyond LP and MIP, including quadratic programming, nonlinear programming and semi-definite programming. We will not consider those more general classes, although it would probably be useful to study them from a CP perspective.

2.3 An example

Let us assume that a company plans to create a network of warehouses to supply its existing stores. Let us suppose in addition that the company already has a number of suitable sites for building warehouses and thus wants to know whether or not to create a warehouse on each such site. For each site chosen, the company wants to determine the optimal capacity for the warehouse. The company considers the average merchandise turnover as identical from one store to another. However, the distance among locations and the transportation infrastructure both lead to varying transportation costs $cost_{s,w}$ for each pair consisting of a store s and a warehouse w. The construction of a warehouse has a given cost *fixed*. All stores have to be supplied regularly. Each store will be supplied by only one warehouse. For each potential location w, only a finite capacity $capacity_w$ is possible. The objective is to minimise total cost by determining for each warehouse which stores should be supplied by it. The total is then the sum of supply costs plus the costs of building each warehouse.

The representation of a problem using decision variables and constraints is often called a *model* of the problem. A MIP model for this particular problem can be derived as follows. Let us denote by W the set of possible warehouse locations, and S the set of stores. We will use one decision variable $open_w$ per warehouse location w whose value will be 1 if a warehouse is built and 0 otherwise. We will also have one decision variable s, w for each pair consisting of a store s and a warehouse w, whose value is 1 if s is supplied by w, 0 otherwise.

The objective to be minimised is then:

$$\sum_{w} fixed \cdot open_{w} + \sum_{s,w} supplyCost_{s,w} \cdot supply_{s,w}. \tag{13}$$

A store is supplied by exactly one warehouse, i.e. only one variable $supply_{s,w}$ can have a value of 1 for a given store s:

$$\sum_{w} supply_{s,w} = 1, \forall s \in S. \tag{14}$$

If a store s is supplied by a warehouse w, then there must be a warehouse at location w. In other words, if $supply_{s,w}$ is equal to 1, then $open_{w}$ must be equal to 1:

$$supply_{s,w} \leq open_{w}, \forall w \in W, \forall s \in S. \tag{15}$$

The number of stores supplied by a warehouse w must be smaller than the capacity $capacity_{w}$ of the warehouse:

$$\sum_{s} supply_{s,w} \leq capacity_{w}, \forall w \in W. \tag{16}$$

The following program, expressed with the OPL language (Van Hentenryck, 1999)[1] represents this problem:

```
range Boolean 0..1;
var Boolean open[W];
var Boolean supply[S,W];

minimize
  sum(w in W) fixed * open[w] +
  sum(w in W, s in S) supplyCost[s,w] * supply[s,w]
subject to {
  forall(s in S)
    sum(w in W) supply[s,w] = 1;
  forall(w in W, s in S)
    supply[s,w] <= open[w];
  forall(w in W)
    sum(s in S) supply[s,w] <= capacity[w];
};
```

3 Constraint programming

A constraint programming (CP) language is a programming language with primitives for declaring decision variables, stating constraints and solving the resulting problems. The roots of CP can be traced back on the one hand to the constraint satisfaction problems (CSPs) in the 1970s, with the advent of arc-consistency techniques (Mackworth, 1977), and on the other hand to the language ALICE (Laurier, 1978) designed for stating and solving combinatorial problems. In the 1980s work in the logic programming community showed that the PROLOG language could be extended by replacing unification with more powerful constraint-solving algorithms. For instance, in 1980 PROLOG II used

[1] We will use OPL throughout the paper for stating examples as this language allows the representation of both mathematical programming and constraint programming. However, the concepts and examples discussed in the paper could have been stated using other languages.

a constraint solver to solve equations and disequations on terms. This idea was further generalised in the constraint logic programming (CLP) scheme (Jaffar & Lassey, 1987), and implemented in several PROLOG variants, SICStus (Carlsson, 1995), PROLOGIII (Colmerauer, 1990), Eclipse (Rodosek *et al*., 1997) and CHIP (Van Hentenryck, 1989). The 1990s have seen the development of programming languages not tied to PROLOG, but ones that offer extended functionalities compared to CLP. Some of them are libraries in mainstream languages, such as C++ [22]. Some others are special-purpose languages, such as Oz (Smolka, 1993), Claire (Caseau & Laburthe, 1995) and OPL (Van Hentenryck, 1999).

3.1 Constraint propagation

The core concept of constraint programming is to explicitly reason about the set of possible values of each variable. This set is often called the *domain* of the variable. Constraints are used for removing values from a variable's domain that do not satisfy the constraints.

To see what we mean, assume that x, y, and z are three variables with integer values in the closed interval [1,10], with the constraint $y < z$. The decision variable y can take a value only among those in the closed interval [1,10], so we can see that the value of y will be at least 1. Since the constraint states that z must be greater than y, $z = 1$ is no longer possible. For that reason, 1 is removed from the domain of z. Similarly, the domain of y becomes [1, 9]. The domain of x remains unchanged since no constraint involves x at this point.

Let us assume now that we add another constraint, say $x = y \times z$. Now the minimal possible value for y is 1, and the minimal possible value for z is 2, so x has to be at least 2. The domain of x is then reduced to [2,10]. Furthermore, as the maximal possible value for x is 10 and the minimal value of z is 2, y, which is equal to x/z, must be at most 5. This entire process, called *constraint propagation*, leads to the following domains for the variables: $x \in [2,10]$, $y \in [1,5]$, and $z \in [2,10]$.

3.2 Search

As our example shows, constraint propagation alone does not necessarily solve the problem completely, since several values are still possible for each decision variable. In such a case, a tree search is used, where each node represents a decision variable, and each branch represents a possible assignment for that variable. Returning to our example, let us suppose that the search begins with the variable x. From this node, several branches can be created, each one of them labelled by one of the values in the domain of x ([2,10]). Suppose we follow the first branch, assigning 2 to x. At this point, constraints are automatically used in order to further reduce the other variable domains, if needed. As the domain of x has changed, the constraint $x = y \times z$ will be activated. Propagation of this constraint reduces the domain of y to [1] and the domain of z to [2]. By now, all the decision variables have been assigned a value, and the problem is solved, so the search stops here.

It may happen that during the search, one node is inconsistent, which means a node is encountered where one or more decision variables have an empty domain. In such a case, no solution is possible and we must try some other choices. The standard way to do this is to undo the last choice we do, and remove all its consequences. In other words, we backtrack in the tree in order to follow another branch. This is done until a solution is found or the entire tree is exhausted.

Optimisation problems require a slight modification of this search. Each time a solution is found, the value of its objective is stored, and a new constraint is added to the problem, stating that the value of the objective must be strictly better than the one we just found. This ensures that we only look for solutions that improve upon the best already found.

Interestingly enough, the order in which the variables are considered is not fixed in advance: this order can be computed dynamically. For instance, a popular ordering is to choose as the next variable the one that has the least number of possible values in its domain. This ordering among the variables can be very important in improving performance.

3.3 Global constraints

One very important aspect of CP is the ability to express and reason about sophisticated constraints.

For instance, consider the following modification of our example:

$$y_1 \in [1, 2] \tag{17}$$

$$y_2 \in [1, 2] \tag{18}$$

$$y_3 \in [1, 2] \tag{19}$$

$$y_i \neq y_j, \qquad for\ i \neq j. \tag{20}$$

This problem has no solution, because we have 2 values for 3 variables. However, constraint propagation on each constraint does not reduce domains. Indeed, given a constraint $y_i \neq y_j$, for each value of y_i (say 1), there always exists a value in the domain of the other variable y_j (say 2) that satisfies the constraint.

The problem comes from the fact that each elementary constraint $y_i \neq y_j$ is treated separately. A global analysis of the problem shows that only two values are available for three variables. This kind of observation has led to the development of so called *global constraints*, where the domain reduction algorithms are applied to n-ary constraints. The set of pairwise inequality constraints above can be replaced by a global constraint of the form *alldiff* (y_1, y_2, y_3). Regin (1994) proposed a graph theoretic approach that computes the arc consistency domain reductions for the *alldiff* constraint in $O(n^{2.5})$. Puget (1998) has proposed a bound consistency algorithm that runs in $O(n * log(n))$.

Another very important type of global constraint is found in production scheduling problems, such as the job-shop or the flow shop problems. Several CP languages or systems incorporate such constraints, e.g. Aggoun & Beldiceanu (1993), Baptiste *et al.* (1995) and Beldiceanu & Contejean (1994).

3.4 An example

Let us revisit our warehouse location problem. It is possible to use a more direct representation of the constraints, involving fewer variables than in the MIP model. Indeed, we will use one decision variable $supplier_s$ whose value w is the warehouse that supplies s, instead of one decision variable per pair (s, w) in the MIP model. We will also use one decision variable $open_w$ per warehouse location w whose value will be 1 if a warehouse is built and 0 otherwise. The costs *supplyCost* and *fixed* are identical to the MIP model. We will introduce an extra variable $cost_s$ to ease the formulation of the model. That variable is equal to the supply cost incurred for store s:

$$cost_s = supplyCost_{s,\, supplier_s},\ \forall w \in W,\ \forall s\ in\ S. \tag{21}$$

Note that the fact that a decision variable can be used as an index in an array is one of the flexibilities offered by CP models.

If a store s is supplied by a warehouse w, then there must be a warehouse at location w. In other words, if $supply_s$ is equal to w, then $open_w$ must be equal to 1:

$$open_{supplier_s} = 1,\ \forall s\ in\ S. \tag{22}$$

The number of stores supplied by a warehouse w must be smaller than the capacity $capacity_w$ of the warehouse:

$$|\{s : supply_s = w\}| \leq capacity_w,\ \forall w \in W \tag{23}$$

where $|S|$ denotes the cardinality of set S.

This set of constraints can be stated as a single global constraint called atmost. There is no standard mathematical notation for that, so we will use the corresponding OPL statement:

```
atmost(capacity, all(w in W) w, supplier);
```

Finally, we want to minimise the total cost which is defined as

$$\sum_w fixed \cdot open_w + \sum_s cost_s \cdot supply_{s,w}. \tag{24}$$

It is possible to slightly improve the model. Indeed, it seems a good idea to try first to favour stores located near a warehouse since the cost of supplying them would be minimal. That can be achieved in searching first to fix the values of the $cost_s$ variables to their minimal values, then to deal with the other decision variables. This can be achieved in OPL by stating a `search` statement:

```
search {
  generate(cost);
};
```

The resulting model can be stated as follows in OPL.

```
int maxCost = max(s in S, w in W) supplyCost[s,w];
var int open[W] in 0..1;
var W supplier[S];
var int cost[S] in 0..maxCost;

minimize
  sum(s in S) cost[s] + sum(w in W) fixed * open[w]

subject to {
  forall(s in S )
    open[supplier[s]] = 1;
    atmost(capacity, all (w in W) w, supplier);
    forall(s in S)
    cost[s] = supplyCost[s,supplier[s]];
      };
search {
  generate(cost);
};
```

Note that this model is very different from the MIP model.

4 CP and MIP

After this tour of CP and the short introduction to MIP techniques, a first comparison can be made. We will compare the two approaches along three dimensions: the modelling ability, node processing and the search algorithm.

First of all, the two approaches have a lot in common as far as the fundamental modelling concepts are concerned. Indeed, they both include the following items: decision variables, constraints on variables, objective to be optimised and tree search.

We will also see later that cuts have a counterpart in CP, usually called *redundant constraints*. A redundant constraint is a constraint that does not remove solutions, but does improve constraint propagation.

On the contrary, CP is clearly more powerful as a representation language, since it allows the representation of constraints in a more natural and compact way. For instance, the MIP model for warehouse location uses only 0 to 1 variables and has $(|W|+1)\times|S|$ of these variables. The CP model uses $|W|+2\times|S|$ variables only. Similarly, the global constraint *atmost* is equivalent to a set of constraints in the MIP model. The extended CP expressiveness may be useful for the debugging and tuning of a model.

Node processing techniques are quite different. In MIP, an LP solver is used for solving the continuous relaxation of the problem, giving either an infeasibility, or a lower bound, on the objective

function. In CP, constraints are propagated, giving either an infeasibility (if a domain becomes empty), or a lower bound, on the cost variable.

The search part is in fact the part with the fewest differences. It must now be clear that CP uses a search technique very close to the branch-and-bound algorithm used in MIP. Indeed, a very similar tree is searched: each node represents a decision variable; each branch represents a value assignment for the variable. Modern MIP solvers allow flexible search strategies, e.g. Beale & Tomlin (1970) and Nemhauser *et al*. (1994).

The relative efficiency of the two approaches cannot be determined a priori, and will depend on the problem, as we will see later.

Instead of contrasting the two approaches, one can wonder if some mixed approach is possible. The answer is yes. For instance, Beringer and De Backer (1995) proposed a mixed search that can be used where constraint propagation is used on all constraints and an LP solver is used with all the linear constraints of the model (i.e. its LP relaxation). Constraint propagation can deduce new bounds on the domains of the variables. Such bounds are then added to the LP relaxation. Conversely, the LP solver can detect that the LP relaxation is inconsistent. It can also compute a lower bound (assuming minimisation) on the objective function. In this way, we have a branch-and-bound search that combines CP and MIP techniques.

This area of research has recently been very active. Several authors have tried to compare the two approaches using different angles, e.g. Bockmayr & Kasper (1998) and Ottoson *et al*. (1999). Some have extended the cooperation scheme outlined above to non-linear constraints, such as piecewise linear functions in Refalo (1999). Some others (e.g. Focacci *et al*. (1999b)) have used global constraints to generate cuts. Others have embedded linear solvers or flow algorithms in global constraints, (e.g. Focacci *et al*. (1999a) and Regin (1994)). This area is quite promising and will undoubtedly lead to dramatic improvements over current methods.

A deeper exposure of all the potential combinations of the two techniques is beyond the scope of this paper. However, we will present two additional examples that will help us to understand how the differences between the two techniques have an impact on the way real problems are modelled and solved.

5 A resource-allocation problem

Let us consider the following resource allocation and scheduling problem.[2]

5.1 A CP model

Five products are given and five machines are available (typical industrial examples involve hundreds of products). The problem is to decide how many units of each product `i` is produced on each machine `k`. This corresponds to decision variables `x[i,k]`. The objective is to maximise the total production:

```
range Five 1..5;
maximize sum (i in Five, k in Five) x[i,k]
```

The time needed to produce a unit of product `i` does not depend on the machine and is noted `d[i]`. The production of product `i` on a machine `k` is done with a single non-interruptible operation O_{ik}, whose starting time is denoted by decision variables `s[i,k]`. The end time of activity O_{ik} is `d[i]*x[i,k]+s[i,k]`. This activity has to be finished before a given limit called the *horizon* of the machine, denoted `h[k]`. When no units of product `i` are produced (i.e. `x[i,k]=0`), then `s[i,k]` is set to 0.

[2] This problem is taken from the noswot model in MIPLIB.

The production of each product is limited by the following constraints

```
forall(i in Five, k in 1..4) x[i,k] <= c[i] ;
forall(i in Five) x[i,5] <= c1[i];
forall(i in Five) sum(k in Five) x[i,k] <= p[i];
sum(k in Five) x[2,k] >= 5;
```

where the capacities `c`, `c1` and `p` are data of the problem. Total production is limited by

```
sum (i in Five, k in Five) x[i,k] <= 43;
```

The above describes a resource allocation problem. The scheduling part of the problem is as follows. The machines are unary resources, which means that two operations O_{ik} and O_{jk} using the same machine cannot overlap in time. This means that either O_{ik} finishes before O_{jk} starts, or O_{jk} finishes before O_{ik} starts. This can be stated as follows:

```
forall(i in Five, j in Five: i < j) {
  forall(k in Five) {
    d[j] * x[j,k] + s[j,k] <= s[i,k]
    \/
    d[i] * x[i,k] + s[i,k] <= s[j,k]
  };
};
```

In addition, we have that

- the activities are optional (i.e. O_{ij} is non-existent when `x[i,k]` is null);
- there is a transition (setup) time `tr[i,j]` between all existing activities. This means that if activity O_{ik} is followed by activity O_{jk} on machine `k`, then there is a delay `tr[i,j]` between the end of O_{ik} and the start of O_{jk}.

In order to model optional activities, a set of 0–1 variables is introduced: `w[i,k]` is 1 if and only if `x[i,k]` is non-null. This is encoded with the following constraints.

```
forall(i in Five, k in Five) {
  x[i,k] >= w[i,k];
  d[i] * x[i,k] + s[i,k] <= h[k] * w[i,k] ;
}
```

Introducing the transition times, the disjunctions on the resources are written as follows:

```
forall(i in Five, j in Five: i < j) {
  forall(k in Five) {
    tr[j,i] * w[j,k] + d[j] * x[j,k] + s[j,k] <= s[i,k]
    \/
    tr[i,j] * w[i,k] + d[i] * x[i,k] + s[i,k] <= s[j,k]
  };
};
```

This ends our model. One can see that CP is flexible and powerful enough to let us start a rather complex, although small, problem in a rather concise way.[3]

[3] Global constraints tailored to scheduling problems could be used, e.g. Aggoun & Beldiceanu (1993) and Baptiste *et al*. (1995). However we preferred to use a naive CP model for the sake of clarity.

Solving the above model to optimality is in fact extremely difficult. Without further help, the CP system cannot solve it in a reasonable time.

5.2 Search

The first improvement is to describe a search strategy. A simple one is to first decide the amount of product to be produced, i.e. find values for the `x[i,k]` variables, then decide the starting time of the activities, i.e. the `s[i,k]` variables.

This search algorithm can be expressed in OPL as follows:

```
search{
  generate(x);
  generate(s);
}
```

This branching strategy can be improved. In fact, as we are interested in maximising production, it makes sense to try to produce the products that cost the least to be produced, i.e. products `i` such that `d[i]` is minimal. Alternatively, we can try to produce as little as possible of the most expensive products, i.e. those products `i` such that `d[i]` is maximal.

```
search {
  forall(i in Five ordered by decreasing d[i])
    generate(w[i]);
  generate(x);
  generate(s);
};
```

5.3 Redundant constraints

When a model is difficult to solve, it is often a good idea to try to express the same constraints in a different way. For instance, it usually helps to explicitly state an implicit constraint. Note that this idea is close to the use of cuts in MIP.

Let us consider one machine and all the activities using that machine. All the activities must be finished before the horizon of the machine. However, the activities cannot overlap in time. Thus the time needed to execute all the activities is at least equal to the sum of the durations of the activities, and it must be smaller than the horizon of the machine. Therefore, we can add the following constraint:

```
forall (k in Five) {
  sum(i in Five) d[i] * x[i,k] <= h[k] ;
};
```

Using this constraint and the search strategy expressed above, the problem can be solved to optimality in roughly one hour.

This can be improved. Indeed, if there are `m` activities on a machine, the number of transitions is `m–1`. The total time of those is at least `(m–1) * Tmin`, where `Tmin` = *min*`(tr[i,j])`. Given that `m` is equal to `sum(i in Five)w[i,k]`, the constraint can be tightened to

```
forall (k in Five) {
  sum(i in Five)
    (d[i] * x[i,k] + Tmin * w[i,k]) <= h[k] + Tmin;
};
```

5.4 Removing symmetries

The above model is also symmetrical: the first 4 machines are identical. Therefore one can add symmetry-breaking constraints:

```
x[2,1] >= x[2,2];
x[2,2] >= x[2,3];
x[2,3] >= x[2,4];
```

Indeed, for any solution to the problem, it is possible to permute the first 4 machines such that the amount of product produced on machine 1 is greater than on machine 2, itself being greater than on machine 3, and so on.

5.5 Cooperative solvers

As most constraints are linear constraints, it makes sense to use a cooperative approach, using both constraint propagation and an LP solver, as explained in Section 4. This is done by adding the qualifier `with linear relaxation` to the model.

The resulting model is shown in Appendix A.

5.6 A MIP formulation

The same problem can be solved as well using a MIP solver. In order to do so, we must express the problem using only linear constraints. The main difficulty in this respect is the presence of disjunctions of linear constraints.

One way to express the disjunction is to introduce a 0–1 variable for each disjunction, and to relate that variable to the constraint using a "big M" approach. Mathematically, a disjunction

$$(lhs_1 \leq rhs_1) \vee (lhs_2 \leq rhs_2) \tag{25}$$

can be replaced by two linear constraints using an extra 0–1 variable t, and a large number M:

$$lhs_1 \leq rhs_1 + M \times t \tag{26}$$

$$lhs_2 \leq rhs_2 + M \times (1 - t). \tag{27}$$

The number M must be greater than or equal to the largest possible value of

$$max(rhs_1 - lhs_1,\ rhs_2 - lhs_2). \tag{28}$$

The disjunction in our example

```
tr[j,i] * w[j,k] + d[j] * x[j,k] + s[j,k] <= s[i,k]
\/
tr[i,j] * w[i,k] + d[i] * x[i,k] + s[i,k] <= s[j,k]
```

can be replaced by the two linear constraints

```
tr[i, j] * w[i,k] + d[i]*x[i,k]
  + s[i,k] <= s[j,k] + h[k]* (1 - t[i,j,k]);
tr[j, i] * w[j,k] + d[j]*x[j,k]
  + s[j,k] <= s[i,k] + h[k] * t[i,j,k] ;
```

where `t[i,j,k]` is a new 0–1 variable, and "big M" is `h[k]`.

The resulting model is very close to the CP one, and is reproduced in Appendix B. Note that one could furthermore avoid the creation of the variables `t[i,j,k]` when `i` is greater than `j`.

5.7 Results

In order to assess the various approaches, we have solved the models on the same laptop PC using ILOG OPL Studio 3.1.

The optimal solution has an objective value of 41. All the variables of the problem are equal to 0 except those below.

```
x[1,1] = 9
x[1,2] = 9
x[1,3] = 9
x[1,4] = 9
x[2,5] = 5
w[l,l] = 1
w[1,2] = 1
w[1,3] = 1
w[1,4] = 1
w[2,5] = 1
```

This means that there is exactly one activity per product, and one per machine. In this case, the scheduling part of the problem is trivial.

Timing results and number of nodes in the search tree are shown in Table 1. The row meanings are as follows. CP + coop corresponds to the model in Appendix A, CP to the same model without the linear relaxation, MIP to the model in Appendix B. A "w/o sym" indicates that the model is modified by removing the symmetry-breaking constraints, "w/o tightening" indicates that the constraint described in Section 5.3 has been removed.

These results show that the various approaches have quite similar running times. This is not sufficient to draw any conclusions on the relative efficiency of each approach. We can note, however, that similar techniques are useful for both approaches: removing symmetries and adding cuts (redundant constraints) greatly improve the running times.

The above results also show that the cooperation is more powerful than CP alone. This may be explained by the structure of this example. Indeed, all constraints are linear except the disjunctions that correspond to the scheduling part of the problem. However, in the optimal solution, there is exactly one activity per product, and one per machine. In this case, the scheduling part of the problem is trivial, i.e. the scheduling constraints are easily satisfied. Therefore the resource allocation part of the problem (the linear constraints) dominates.

In fact very few general statements can be made on the relative efficiency of both approaches. Based on our experience in the last ten years, one can only say that if the MIP model is quite the same size as the CP model, and if the LP relaxation is tight, then the MIP solver is likely to be as efficient, if not better, than the CP model. If this is not the case, i.e. if the CP model is much more compact, then CP is likely to be better, as we will see.

Table 1 Result on a resource allocation problem

	time	nodes
CP + coop	22	8954
CP	126	292208
CP + coop w/o sym	85	36182
CP + coop w/o tightening		
MIP	24	5171
MIP w/o sym	98	22372
MIP w/o tightening	84	20645

6 A graph-colouring example

The previous example could be solved satisfactorily using MIP. We will now look at another example, where using linear constraints is not as powerful as CP. The problem is a special kind of graph colouring.[4] We are given `numnodes` nodes, and `numedges` edges. We must label each node with a number between `0` and `numnodes`, and each edge with a number between `0` and `numedges`. Any two nodes must have different labels. Any two edges must have different labels. Furthermore, given an edge `e` between nodes `e.i` and `e.j`, the label of `e` must be equal to the absolute difference of the labels of nodes `e.i` and `e.j`. If a graph can be coloured in this way it is said to be *graceful*.

6.1 A CP model

A CP model is quite straightforward. An edge will be represented by an object:

```
range nodes l..numnodes;
range labels 0..numedges;
struct Edge {
  nodes i;
  nodes j;
};
```

We associate one decision variable per node and one per edge for the labels.

```
var labels nl[nodes];
var labels el[edges];
```

The constraints are then straightforward. Note that we use the global constraint `alldiff` to state a set of difference constraints in a single statement.

```
solve {
  alldifferent(nl);
  alldifferent(el);
  forall(e in edges) {
    el[e] = abs(nl[e.i] - nl[e.j]);
  }
};
```

The search is also quite simple. We first look for the labels of the nodes, then the labels of the edges.

```
search {
  generate(nl);
  generate(el);
};
```

6.2 Improving the CP model

The previous model can be significantly improved. First of all, consider an edge `e`. Its label is equal to `abs(nl[e.i] - nl[e.j])`, where `nl[e.i]` and `nl[e.j]` are the labels of the extremities of `e`. However, those two numbers must be different according to the problem definition. Therefore, the label of `e` must be at least `1`. This is a redundant constraint, or a cut. Adding this redundant constraint will significantly help the `alldiff` constraint propagation.

[4] This example is very different from the classical graph-colouring examples where two nodes linked by an edge must have a different color. Here, the problem is much more complex and small instances were still open until CP was applied to the problem.

Table 2 Results on a graph-colouring problem

		model 1	model 1	model 2	model 2
numnodes	numvars	time	bt	time	bt
8	24	0.4	691	0.02	88
12	38	83	187663	2.8	5715
16	52	> 100000		6.6	11421
20	66			4274	5301372

Second, it is possible to use a more powerful propagation algorithm (Puget, 1998) for the `alldiff` constraint. This is done by adding the `onRange` qualifier in OPL. Third, once nodes are labelled, labels for edges are known. We can therefore remove the search for their solution. The modified model is the following:

```
struct Edge {
  nodes i;
  nodes j;
};
range labels 0..numedges;
var labels nl[nodes];
var labels el[edges];
solve {
  alldifferent(nl) onRange;
  alldifferent(el) onRange;
  forall(e in edges) {
    el[e] = abs(nl[e.i] - nl[e.j]);
    el[e] > 0;
  };
};
search {
  generate(nl);
};
```

In order to evaluate the two models, we took a set of graphs such that it was not yet known whether a graceful colouring existed. The results are shown in Table 2. Time is given in seconds, and the column "bt" indicates the number of backtracking operations in the search.

The modified model is significantly more efficient, and can easily solve open problems in graph theory. This also shows that even with a powerful problem-solving tool, the way the problem is stated can have a dramatic impact on performance. In that case, the modified model is much more efficient because the `alldiff` constraints are used more effectively.

6.3 A MIP model

A MIP model is much more difficult to obtain. We have two difficulties. First of all, the `alldiff` constraints are not linear constraints. The usual way to linearise this is to introduce a set of 0–1 variables where the CP model uses an integer decision variable.[5]

For each variable `el[i]`, a set of 0–1 variables `elb[i,l]` is introduced. The variable `elb[i,l]` is one if and only if the variable `el[i]` is equal to `1`. Therefore for a given `i` only one of the variables `elb[i,l]` can be equal to `1`. This is expressed by the constraints

[5] Note that a similar difference was observed in the warehouse location problem.

```
forall(i in edges) {
  sum(1 in labels) elb[i,l] = 1;
  sum(1 in labels) l*elb[i,l] = el[i];
};
```

Then the `alldiff(el)` constraint can be stated saying that at most one of the two variables `elb[i,l]` and `elb[j,l]` can be non-null:

```
forall(i in edges, j in edges : i <> j)
  forall(l in labels)
    elb[i,l] + elb[j,l] <= 1;
```

Similar statements are used for linearising the `alldiff(nl)` constraint.

The second difficulty is to linearise the absolute value constraints. We will use the "big M" approach. Indeed,

```
el[e] = abs(nl[e.i] – nl[e.j]);
```

is equivalent to

```
el[e] <= nl[e.i] – nl[e.j] & el[e] >= nl[e.i] – nl[e.j]
\/
el[e] <= nl[e.j] – nl[e.i] & el[e] >= nl[e.j] – nl[e.i];
```

Using a big M approach, we can translate this into

```
el[e] <= nl[e.i] – nl[e.j] + 2*numedges*t[e];
el[e] >= nl[e.i] – nl[e.j] – 2*numedges*t[e];
el[e] <= nl[e.j] – nl[e.i] + 2*numedges*(1-t[e]);
el[e] >= nl[e.j] – nl[e.i] – 2*numedges*(1-t[e]);
```

Last, the constraint

```
el[e] > 0;
```

can be translated into

```
elb[e, 0] = 0;
```

The resulting model is a MIP. This model can be used to solve some problems with a MIP solver, but is not powerful enough to solve the graphs solved by the CP approach. This is quite understandable; when the CP approach uses `numnodes + numedges` variables, the MIP statement uses `(numnodes + numedges)*numedges` variables, which are much larger.

7 Conclusions

Two powerful techniques for solving optimisation problems have been presented, namely constraint programming (CP) and maths programming (MP). It appears that CP techniques have a lot in common with mixed integer programming (MIP), a particular kind of MP technique: they share the same basic entities for modelling, namely decision variables, constraints and objective function. Both techniques also rely on a tree search that can be controlled in various ways. The difference mainly comes from the modelling ability, where CP can express a wider set of constraints, and processing done at each node. CP uses propagation algorithms embedded in global constraints in order to reduce the set of possible values for each variable, whereas MIP solvers rely on solving the linear relaxation of the problem. Despite those differences, similar ideas can be used to help solvers: cuts in MIP and redundant constraints in CP.

We have presented MIP and CP on three examples, showing various modelling techniques. These examples show that it is extremely difficult to predict the relative efficiency of each technique from a practical point of view.

We believe that each technique has its own strength, and that various combinations will prove useful in the future. This is indeed a very active research area, and we have briefly presented some recent

related work. The growing interest in the convergence of CP and MIP is also witnessed by the development of modelling systems that encompass both, e.g. OPL.

References

Aggoun, A and Beldiceanu, N, 1993, "Extending CHIP to solve complex scheduling problems" *Mathematical and Computer Modeling* **17**(7) 57–73.

Baptiste, P, Le Pape, C and Nuijten, P, 1995, "Incorporating efficient operations research algorithms in constraint-based scheduling" *Proceedings of 1st International Joint Workshop on Artificial Intelligence and Operations Research*, Timberline, Oregon.

Beale, ELM and Tomlin, JA, 1970, "Special facilities in general mathematical programming system for non-convex problems using ordered sets of variables" *Proceedings of the Fifth International Conference on Operations Research* 447–454.

Beldiceanu, N and Contejean, E, 1994, "Introducing global constraints in CHIP" *Mathematical and Computer Modeling* **20**(12) 97–123.

Beringer, H and De Backer, B, 1995, "Combinatorial problem solving in constraint logic programrming with cooperating solvers" in C Beirle and L Plumer (eds) *Logic Programming: Formal Methods and Practical Applications, Studies in Computer Science and Artificial Intelligence*, Elsevier.

Bockmayr, A and Kasper, T, 1998, "Branch and infer: a unifying framework for integer and finite domain constraint programming" *INFORMS J. Computing* **10**(3) 287–300.

Carlsson, M, 1995, *SICStus Prolog User's Manual. SICS Research Report* Swedish Institute of Computer Science.

Caseau, Y and Laburthe, F, 1995, "The Claire documentation" *LIENS Report 96–15* Ecole Normale Superieure, Paris.

Colmerauer, A, 1990, "Une introduction à PROLOG III" *Actes AVIGNON 90* 27–70.

Dantzig, GB, 1948, "Programming in a linear structure" *Comptroller*, USAF, February.

Focacci, F, Lodi, A and Milano, M, 1999a, "Cost-based domain filtering" in J Jaffar (ed.) *Principle and Practice of Constraint Programming* Lecture Notes in Computer Science, Springer.

Focacci, F, Lodi, A and Milano, M, 1999b, "Solving TSP with time windows with constraints" *Sixteenth International Conference on Logic Programming*.

Jaffar, J and Lassez, J-L, 1987, *Constraint Logic Programming. Proceedings of POPL 1987*.

Karmarkar, N, 1984, "A new polynomial-time algorithm for linear programming" *Combinatorica* **4** 373–395.

Khachiyan, LG, 1979, "A polynomial algorithm in linear programming" *Soviet Mathematics Doklady* **20** 191–194.

Lauriere, J-L, 1978, "A language and a program for stating and solving combinatorial problems *Artificial Intelligence* **10**(1).

Lustig, IJ, Marsten, RE and Shanno, DF, 1994, "Interior-point methods for linear programming: Computational state of the art" *ORSA Journal on Computing* **6** 1–14.

Mackworth, AK, 1977, "Consistency in networks of relations" *Artificial Intelligence* **8** 99–118.

Nembauser, GL and Wolsey LA, 1988, *Integer and Combinatorial Optimization* Wiley, Chichester.

Nemhauser, GL, Savelsberg, MWP and Sigismondi, GC, 1994, "MINTO, a Mixed INTeger Optimizer" *Operations Research Letters* **15** 47–58.

Ottoson, G, Thorsteinsson, ES and Hooker, JN, 1999, "Mixed global constraints and inference in hybrid CLP-IP solvers" *CP99 Post-Conference Workshop on Large-Scale Combinatorial Optimization and Constraints*.

Puget, J-F, 1998, "A fast algorithm for the bound consistency of alldiff constraints" *Proceedings of 15th National Conference on Artificial Intelligence (AAAI-98)* 359–366.

Puget, J-F and Leconte, M, 1995, "Beyond the glass box: constraints as objects" *Logic Programming: Proceedings of the 1995 International Symposium* 513–527.

Refalo, P, 1999, "Tight cooperation and its application in piecewise linear optimization" in J. Jaffar (ed.) *Principle and Practice of Constraint Programming* Lecture Notes in Computer Science, Springer.

Regin, JC, 1994, "A filtering algorithm for constraints of difference in CSPs" *Proceedings of the 12th National Conference on Artificial Intelligence* 362–367.

Regin, JC, 1999, "Arc consistency for global cardinality constraints with cost" in J. Jaffar (ed.) *Principle and Practice of Constraint Programming* Lecture Notes in Computer Science, Springer.

Rodosek, R, Wallace, M and Hajian, M, 1997, *A New Approach to Integrating Mixed Integer Programming and Constraint Logic Programming* Baltzer Journals.

Smolka, G, 1993, "Survey of Oz: a higher-order concurrent constraint language" *Proceedings of ICLP'93 Post-Conference Workshop on Concurrent Constraint Programming*.

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

Van Hentenryck, P, 1999, *The OPL Optimization Programming Language* MIT Press.

Wolsey, LA, 1998, *Integer Programming* John Wiley & Sons.

8 Appendix A: a CP model

```
range Five 1..5;
int d[Five] = [20833 , 29762 , 34722 , 22401, 20833];
int c[Five] = [ 96000, 67200 , 57600, 89280, 96000];
int cl[Five] = [ 76800, 53760 , 46080, 71424 , 76800];
int h[Five] = [200000, 200000, 200000, 200000, 160000];
int tr[Five, Five] = [[0 ,    7500 , 6667 , 2500 , 2500],
                      [6667 , 0 ,    6667 , 6667 , 6667],
                      [6667 , 7500 , 0 ,    6667 , 6667],
                      [2500 , 7500 , 6667 , 0 ,    2500],
                      [6667 , 7500 , 6667 , 6667 , 0]
] ;
int p[Five] = [38, 9, 2, 1, 2];
var int+ x[Five, Five] in 0..100;
var int+ w[Five, Five] in 0..1;
var int+ s[Five, Five] in 0..1000000000 ;
maximize with linear relaxation
  sum (i in Five, k in Five) x[i,k]
subject to {
  sum (i in Five, k in Five) x[i,k] <= 43;
   forall (i in Five, k in Five) {
    x[i,k] >= w[i,k];
    d[i] * x[i,k] + s[i,k] <= h[k] * w[i,k] ;
   };
   forall (i in Five, k in 1..4) x[i,k] <= c[i] ;
   forall (i in Five) x[i,5] <= c1[i];
   forall (i in Five) sum(k in Five) x[i,k] <= p[i];
   sum(k in Five) x[2,k] >= 5;
   forall (i in Five, j in Five: i < j){
    forall (k in Five) {
      tr[j,i] * w[j,k] + d[j] * x[j,k] + s[j,k] <= s[i,k]
      \/
      tr[i,j] * w[i,k] + d[i] * x[i,k] + s[i,k] <= s[j,k]
    };
   };
   forall (k in Five) {
    sum(i in Five) (d[i] * x[i,k] + 2500 * w[i,k]) <= h[k] + 2500;
   };
   x[2,1] >= x[2,2];
   x[2,2] >= x[2,3];
   x[2,3] >= x[2,4];
 };
search {
 forall(i in Five ordered by decreasing d[i])
  generate(w[i]);
 generate(x);
 generate(s);
};
```

9 Appendix B: a MIP model

```
range Five 1..5;
float d[Five] = [20833 , 29762 , 34722 , 22401, 20833];
float c[Five] = [ 96000, 67200 , 57600, 89280, 96000];
float c1[Five] = [ 76800, 53760 , 46080, 71424 , 76800];
int h[Five] = [200000, 200000, 200000, 200000, 160000];
float tr[Five, Five] = [[0 ,    7500 , 6667 , 2500 , 2500],
                        [6667 , 0 ,    6667 , 6667 , 6667],
                        [6667 , 7500 , 0 ,    6667 , 6667],
                        [2500 , 7500 , 6667 , 0 ,    2500],
                        [6667 , 7500 , 6667 , 6667 , 0]
] ;
int p[Five] = [38, 9, 2, 1, 2];
var int+ x[Five, Five] in 0..100000;
var int+ w[Five, Five] in 0..1;
var int+ t[Five, Five, Five] in 0..1;
var float+ s[Five, Five];
maximize sum (i in Five, j in Five) x[i,j]
subject to {
 sum (i in Five, j in Five) x[i,j] <= 43;
 forall (i in Five, j in Five) {
  x[i,j] >= w[i,j];
  d[i] * x[i,j] + s[i,j] <= h[j] * w[i,j] ;
 };
 forall (i in Five, j in 1..4) x[i,j] <= c[i] * w[i,j];
 forall (i in Five) x[i,5] <= c1[i] * w[i,5];
 forall (i in Five) sum(j in Five) x[i,j] <= p[i];
 sum(j in Five) x[2,j] >= 5;
 forall (i in Five, j in Five: i < j){
  forall (k in Five) {
   tr[i, j] * w[i,k] + d[i]*x[i,k] + s[i,k]
    <= s[j,k] + h[k]* (1 - t[i,j,k]);
   tr[j, i] * w[j,k] + d[j]*x[j,k] + s[j,k]
    <= s[i,k] + h[k] * t[i,j,k] ;
  };
 };
 forall (i in Five, j in Five: i >= j)
  forall (k in Five) {
   t[i,j,k] = 0; // could be avoided
 };
 forall (k in Five) {
    sum(i in Five) (d[i] * x[i,k] + 2500 * w[i,k]) <= h[k] + 2500;
   };
 x[2,1] >= x[2,2];
 x[2,2] >= x[2,3];
 x[2,3] >= x[2,4];
};
```