# Constraint programming

Alexander Bockmayr John N. Hooker

May 2003

## Contents

# 1 Introduction

A discrete optimization problem can be given a declarative or procedural formulation, and both have their advantages. A declarative formulation simply states the constraints and objective function. It allows one to describe what sort of solution one seeks without the distraction of algorithmic details. A procedural formulation specifies how to search for a solution, and it therefore allows one to take advantage of insight into the problem in order to direct the search. The ideal, of course, would be to have the best of both worlds, and this is the goal of constraint programming.

The task seems impossible at first. A declarative formulation is static, and a procedural formulation dynamic, in ways that appear fundamentally at odds. For example, setting $x = 0$ at one point in a procedure and $x = 1$ at another point is natural and routine, but doing the same in a declarative model would simply result in an infeasible constraint set.

Despite the obstacles, the artificial intelligence community has developed ways to weave procedural and declarative elements together. The evolution of ideas passed through logic programming, constraint logic programming, concurrent constraint programming, constraint handling rules, and constraint programming (not necessarily in that order). One idea that has been distilled from this research program is to *view a constraint as invoking a procedure*. This is the basic idea of constraint programming.

## 1.1 Constraints as Procedures

A constraint programmer writes a constraint declaratively but views it as a procedure that operates on the solution space. Each constraint contributes a relaxation of itself to the *constraint store*, which limits the portion of the space that must be searched. The constraints in the constraint store should be easy in the sense that it is easy to generate feasible solutions for them. The overall solution strategy is to find a feasible solution of the original problem by enumerating solutions of the constraint store in a way to be described shortly.

In current practice the constraint store primarily contains very simple *in-domain* constraints, which restrict a variable to a domain of possible values. The domain of a variable is typically an interval of real numbers or a finite set. The latter can be a set of any sort of objects, not necessarily numbers, a fact which lends considerable modeling power to constraint programming.

The idea of treating a constraint as a procedure is a very natural one for a community trained in computer science, because statements in a computer program typically invoke procedures. This simple device yields a powerful tool for exploiting problem structure. In most practical applications, there are some subsets of constraints that have special structure, but the problem as a whole does not. Existing optimization methods can deal with this situation to some extent, for instance by using Benders decomposition to isolate a linear part, by presolving a network flow subproblem, and so forth. However, most methods that exploit special structure require that the entire problem exhibit the structure. Constraint programming avoids this difficulty by associating procedures with highly structured subsets of constraints. This allows procedures to be designed to exploit the properties of the constraints.

Strictly speaking, constraint programming associates procedures with individual constraints rather than subsets of constraints, but this is overcome with the concept of *global constraints*. A global constraint is a single constraint that represents a highly structured set of constraints. An example would be an `alldifferent` constraint that requires that a set of variables take distinct values. It represents a large set of pairwise disequations. A global constraint can be designed to invoke the best known technology for dealing with its particular structure. This contrasts with the traditional approach used in optimization, in which the solver receives the problem as a set of undifferentiated constraints. If the solver is to exploit any substructure in the problem, it must find it, as some commercial solvers find network substructure. Global constraints, by contrast, allow the user to alert the solver to portions of the problem that have special structure.

How one can solve a problem by applying special-purpose procedures to individual constraints? What links these procedures together? This is where the constraint store comes into play. Each procedure applies a *filtering* algorithm that eliminates some values from from the variable domains. In particular, it eliminates values that cannot be part of any feasible solution for that constraint. The restricted domains are in effect in-domain constraints that are implied by the constraint. They become part of the constraint store, which is passed on to the next constraint to be processed. In this way the constraint store "propagates" the results of one filtering procedure to the others.

Naturally the constraints must be processed in some order, and different systems do this in different ways. In programs written for the ILOG Solver, constraints are objects in a C++ program that determines how the constraints are processed. Programs written in OPL Studio have a more declarative look, and the system exerts more control over the processing.

A constraint program can therefore be viewed as a "program" in the sense of a computer program: the statements invoke procedures, and control is passed from one statement to another, although the user may not specify the details of how this is done. This contrasts with mathematical programs, which are not computer programs at all but are fully declarative statements of the problem. They are called programs because of George Dantzig's early application of linear programming to logistics "programming" (planning) in the military. Notwithstanding this difference, a constraint programming formulation tends to look more like a mathematical programming model than a computer program, since the user writes constraints declaratively rather than writing code to enforce the constraints.

## 1.2 Parallels with Branch and Cut

The issue remains as to how to enumerate solutions of the constraint store in order to find one that is feasible in the original problem. The process is analogous to branch-and-cut algorithms for integer programming, as Table 1 illustrates. Suppose that the problem contains variables $x = [x_1, \ldots, x_n]$ with domains $D_1, \ldots, D_n$. If the domains $D_j$ can all be reduced to singletons $\{v_j\}$, and if $x = v = [v_1, \ldots, v_n]$ is feasible, then $x = v$ solves the problem. Setting $x = v$ in effect solves the constraint store, and the solution of the constraint store happens to be feasible in the original problem. This is analogous to solving the continuous relaxation of an integer programming problem (which is the "constraint store" for such a problem) and obtaining an integer solution.

If the domains are not all singletons, then there are two possibilities. One is that there is an empty domain, in which case the problem is infeasible. This is analogous to an infeasible continuous relaxation in branch and cut. A second possibility is that some domain $D_j$ contains more than a single value, whereupon it is necessary to enumerate solutions of the constraint store by branching. One can branch on $x_j$ by partitioning $D_j$ into smaller domains, each corresponding to a branch. One could in theory continue to branch until all solutions are enumerated, but as in branch and cut, a new relaxation (in this case, a new set of domains) is generated at each node of the branching tree.

Relaxations become tighter as one descends into the tree, since the domains start out smaller and are further reduced through constraint propagation. The search continues until the domains are singletons, or at least one is empty, at every leaf node of the search tree.

The main parallel between this process and branch-and-cut methods is that both involve *branch and infer*, to use the term of Bockmayr & Kasper (1998). Constraint programming infers in-domain constraints at each node of the branching tree in order to create a constraint store (relaxation). Branch and cut infers linear inequalities at each node in order to generate a continuous relaxation. In the latter case, some of the inequalities in the relaxation appear as inequality constraints of the original integer programming problem and so are trivial to infer, and others are cutting planes that strengthen the relaxation.

Another form of inference that occurs in both constraint programming and integer programming is *constraint learning*, also known as the *nogood generation*. Nogoods are typically formulated when a trial solution (or partial solution) is found to be infeasible or suboptimal. They are constraints designed to exclude the trial solution as search continues, and perhaps other solutions that are unsatisfactory for similar reasons. Nogoods are closely parallel to the integer programming concept of Benders cuts, which are likewise generated when solution of the master program yields a suboptimal or infeasible solution. They are less clearly analogous to cutting planes, except perhaps separating cuts, which are generated to “cut off” a nonintegral solution.

Constraint programming and integer programming exploit problem structure primarily in the inference stage. Constraint programmers, for example, invest considerable effort into the design of filters that exploit the structure of global constraints, just as integer programmers study the polyhedral structure of certain problem classes to generate strong cutting planes.

There are three main differences between the two approaches.

- Branch and cut generally seeks an optimal rather than a feasible solution. This is a minor difference, because it is easy to incorporate optimization into a constraint programming solver. Simply impose a bound on the value of the objective function and tighten the bound whenever a feasible solution is found.

- Branch and cut solves a relaxation at every node with little or no constraint propagation, whereas constraint programming relies more on propagation but does not solve a relaxation. (One might say that it “solves” the constraint store in the special case in which the domains are singletons.) In branch and cut, solution of the relaxation provides a bound on the optimal value that often allows pruning of the search tree. It can also guide branching, as for instance when one branches on a variable with a nonintegral value.

- The constraint store is much richer in the case of branch-and-cut methods, because it contains linear inequalities rather than simply in-domain constraints. Fortunately, the two types of constraint store can be used simultaneously in the hybrid methods discussed below.

## 1.3 Constraint Satisfaction

Issues that arise in domain reduction and branching search are addressed in the *constraint satisfaction* literature, which is complementary to the optimization literature in interesting ways.

Table 1: Comparison of constraint programming search with branch and cut.

| | *Constraint Programming* | *Branch and Cut* |
|---|---|---|
| *Constraint store (relaxation)* | Set of in-domain constraints | Continuous relaxation (linear inequalities) |
| *Branching* | Branch by splitting a non-singleton domain, or by branching on a constraint | Branch on a variable with a noninteger value in the solution of the relaxation |
| *Inference* | Reduce variable domains (i.e., add in-domain constraints to constraint store); add nogoods | Add cutting planes to relaxation (which also contains inequalities from the original IP); add Benders or separating cuts* |
| *Bounding* | None | Solve continuous relaxation to get bound |
| *Feasible solution is obtained at a node...* | When domains are singletons and constraints are satisfied | When solution of relaxation is integral |
| *Node is infeasible...* | When at least one domain is empty | When continuous relaxation is infeasible |
| *Search backtracks...* | When node is infeasible | When node is infeasible, relaxation has integral solution, or tree can be pruned due to bounding |

*Commercial solvers also typically apply preprocessing at the root note, which can be viewed as a rudimentary form of inference or constraint propagation.

Perhaps the fundamental idea of constraint satisfaction is that of a *consistent* constraint set, which is roughly parallel to that of a convex hull description in integer programming. In this context, "consistent" does not mean feasible or satisfiable. It means that the constraints provide a description of the feasible set that is so explicit that a feasible solution can be found without backtracking.

If an integer/linear programming constraint set is a convex hull description, it in some sense provides an explicit description of the feasible set. Every facet of the convex hull of the feasible set is explicitly indicated. One can solve the problem easily by solving its continuous relaxation. There is no need to use a backtracking search such as branch and bound or branch and cut.

In similar fashion, a consistent constraint set allows one to solve the problem easily with a simple greedy algorithm. For each variable, assign to it the first value in its domain that, in conjunction with the assignments already made, violates no constraint. (A constraint cannot be violated until all of its variables have been assigned.) In general one will reach a point where no value in the domain will work, and it is necessary to backtrack and try other values for previous assignments. However, if the constraint set is consistent, the greedy algorithm always works. The constraint set contains explicit constraints that rule out any partial assignment that cannot be completed to obtain a feasible solution.

Thus consistency, like integrality, allows one to solve the problem without backtracking. The idea of consistency does not seem to have developed in the optimization literature, although cutting planes and preprocessing techniques serve in part to make a constraint set more nearly consistent. Since perfect consistency is as hard to achieve as integrality, weaker forms of consistency have been defined, including $k$-consistency, arc consistency, generalized arc consistency, and bounds consistency. These are discussed in Section 2 below.

The concept of consistency is closely related to domain reduction. A constraint set is *generalized arc consistent* if for every $v \in D_j$, the variable $x_j$ takes the value $v$ in some feasible solution. Thus if a filtering algorithm for some constraint reduces domains as much as possible, it achieves generalized arc consistency with respect to that constraint. A filtering algorithm that operates on domains in the form of numeric intervals achieves *bounds consistency* if it narrows the intervals as much as possible.

The constraint satisfaction literature also deals with search strategies, variable and value selection in a branching search, and efficient methods for constraint propagation. These are discussed in Section 3.

## 1.4 Hybrid Methods

Constraint programming and optimization have complementary strengths that can be profitably combined.

- Problems often have some constraints that propagate well, and others that relax well. A hybrid method can deal with both kinds of constraints.
- Constraint programming's idea of global constraints can exploit substructure in the problem, while optimization methods for highly structured problem classes can be useful for solving relaxations.
- Constraint satisfaction can contribute filtering algorithms for global constraints, while optimization can contribute relaxations for them.

Due to the advantages of hybridization, constraint programming is likely to become established in the operations research community as part of a hybrid method, rather than as a technique to be used in isolation.

The most obvious sort of hybrid method takes advantage of the parallel between constraint solvers and branch-and-cut methods. At each node of the search tree, constraint propagation creates a constraint store of in-domain constraints, and polyhedral relaxation creates a constraint store of inequalities. The two constraint stores can enrich each other, since reduced domains impose bounds on variables, and bounds on variables can reduce domains. The inequality relaxation is solved to obtain a bound on the optimal value, which prunes the search tree as in branch-and-cut methods. This method might be called a *branch, infer and relax* (BIR) method.

One major advantage of a BIR method is that one gets the benefits of polyhedral relaxation without having to express the problem in inequality form. The inequality relaxations are generated within the solver by relaxation procedures that are associated with global constraints, a process that is invisible to the user. A second advantage is that solvers can easily exploit the best known relaxation technology. If a global constraint represents a set of traveling salesman constraints, for example, it can generate a linear relaxation containing the best known cutting planes for the problem. Today, much cutting plane technology goes unused because there is no systematic way to apply it in general-purpose solvers.

Another promising approach to hybrid methods uses generalized Benders decomposition. One partitions the variables $[x, y]$ and searches over values of $x$. The problem of finding an optimal value for $x$ is the *master problem*. For each value $v$ enumerated, an optimal value of $y$ is computed on the assumption that $x = v$; this is the *subproblem*. In classical Benders decomposition, the subproblem is a linear or nonlinear programming problem, and its dual solution yields a Benders cut that is added to the master problem. The Benders cut requires all future values of $x$ enumerated to be better than $v$. One keeps adding Benders cuts and re-solving until no more Benders cuts can be generated.

This process can be generalized in a way that unites optimization and constraint programming. The subproblem is set up as a constraint programming problem. Its "dual" can be defined as an *inference dual*, which generalizes the classical dual and can be solved in the course of solving the primal with constraint programming methods. The dual solution yields a generalized Benders cut that is added to the master problem. The master problem is formulated and solved as a traditional optimization problem, such as a mixed integer programming problem. In this way the decomposition scheme combines optimization and constraint programming methods.

BIR and generalized Benders decomposition can be viewed as special cases of a general algorithm that enumerates a series of problem restrictions and solves a relaxation for each. In BIR, the leaf nodes of the search tree correspond to restrictions, and their continuous relaxations are solved. In Benders, the subproblems are problem restrictions, and the master problems are relaxations. This provides a basis for a general scheme for integrating optimization, constraint programming and local search methods (Hooker 2003).

## 1.5 Performance Issues

A problem-solving technology should be evaluated with respect to modeling power and development time as well as solution speed.

Constraint programming provides a flexible modeling framework that tends to result in succinct models that are easier to debug than mathematical programming models. In addition, it quasi-procedural approach allows the user to provide the solver information on how best to attack the problem. For example, users can choose global constraints that indicate substructure in the model, and they can define the search strategy conveniently within the model specification.

Constraint programming has other advantages as well. Rather than choose between two alternative formulations, the modeler can simply use both and significantly speed the solution by doing so. The modeler can add side constraints to a structured model without slowing the solution, as often happens in mathematical programming. Side constraints actually tend to accelerate the solution by improving propagation.

On the other hand, the modeler must be familiar with a sizeable lexicon of global constraints in order to write a succinct model, while integer programming models use only a few primitive terms. A good deal of experimentation may be necessary to find the right model and search strategy for efficient solution, and the process is more an art than a science.

The computational performance of constraint programming relative to integer programming is difficult to summarize. Constraint programming may be faster when the constraints contain only two or three variables, since such constraints propagate more effectively. When constraints contain many variables, the continuous relaxations of integer programming may become indispensible.

Broadly speaking, constraint programming may be more effective for scheduling problems, particularly resource-constrained scheduling problems, or other combinatorial problems for which the integer programming model tends to be large or have a weak continuous relaxation. This is particularly true if the goal is to find a feasible solution or to optimize a min/max objective, such as makespan.

Integer programming may excel on structured problems that define a well-studied polyhedron, such as the traveling salesman problem. Constraint programming may become competitive when such problems are complicated with side constraints, such as time windows in the case of the traveling salesman problem, or when they are part of a larger model.

It is often said that constraint programming is more effective for "highly-constrained" problems, presumably because constraint propagation is better. Yet this can be misleading, since one can make a problem highly constrained by placing a tight bound on a cost function with many variables. Such a maneuver is likely to make the problem intractable for constraint programming.

The recent trend of combining constraint programming and integer programming makes such comparisons less relevant, since the emphasis shifts to how the strengths of the two methods can complement each other. The computational advantage of integration can be substantial. For example, a hybrid method recently solved product configuration problems 300 to 600 times faster than either mixed integer programming (CPLEX) or constraint programming (ILOG Solver) (Ottosson & Thorsteinsson 2000). The problems required selecting each component in some product, such as a computer, from a set of component types; thus one might select a power supply to be any of several wattages. The number of components ranged from 16 to 20 and the number of component types from 20 to 30.

In another study, a hybrid method based on Benders decomposition resulted in even greater speedups for machine scheduling (Jain & Grossmann 2001, Hooker 2000, Thorsteinsson 2001). Each job was scheduled on one of several machines, subject to time windows, where the machines run at different speeds and process each job at a different cost. The speedups increase with problem size and reach five to six orders of magnitude, relative to CPLEX and the ILOG Scheduler, for 20 jobs and 5 machines. Section 4.3 discusses this problem in detail, and Section 4.5 surveys other applications of hybrid methods.

## 2 Constraints

In this section, we give a more detailed treatment of the declarative and procedural aspects of constraint reasoning.

### 2.1 What is a Constraint?

A constraint $c(x_1, \ldots, x_n)$ typically involves a finite number of decision variables $x_1, \ldots, x_n$. Each variable $x_j$ can take a value $v_j$ from a finite set $D_j$, which is called the *domain* of $x_j$. The constraint $c$ defines a relation $R_c \subset D_1 \times \cdots \times D_n$. It is satisfied if $(v_1, \ldots, v_n) \in R_c$.

A *constraint satisfaction problem* is a finite set $C = \{c_1, \ldots, c_m\}$ of constraints on a common set of variables $\{x_1, \ldots, x_n\}$. It is *satisfiable* or *feasible* if there exists a tuple $(v_1, \ldots, v_n)$ that satisfies simultaneously all the constraints in $C$. A *constraint optimization problem* involves in addition an objective function $f(x_1, \ldots, x_n)$ that has to be maximized or minimized over the set of all feasible solutions.

Many constraint satisfaction problems are NP-complete.

## 2.2 Arithmetic versus Symbolic Constraints

The concept of "constraint" in constraint programming is very general. It includes classical mathematical constraints like linear or nonlinear equations and inequalities, which are often called *arithmetic constraints.* A crucial feature of constraint programming, however, is that it offers in addition a large variety of other constraints, which may be called *symbolic constraints.* In principle, a symbolic constraint could be defined by any relation $R \subset D_1 \times \cdots \times D_n$. However, in order to be useful for constraint programming, it should have a natural declarative reading, and efficient filtering algorithms (see Sect. 2.6). Symbolic constraints that arise by grouping together a number of simple constraints, each on a small number of variables, into a new constraint involving all these variables together, are called *global* constraints. Global constraints are a key concept of constraint programming. On the declarative level, they increase the expressive power. On the operational side, they improve efficiency.

## 2.3 Global Constraints

We next give an overview of some popular global constraints.

**Alldifferent** The constraint $\texttt{alldifferent}([x_1, \ldots, x_n])$ states that the variables $x_1, \ldots, x_n$ should take pairwise different values (Régin 1994, Puget 1998, Mehlhorn & Thiel 2000). From a declarative point of view, this is equivalent to a system of disequations $x_i \neq x_j$, for all $1 \leq i < j \leq n$. Grouping together these constraints into one global constraint allows one to make more powerful inferences. For example, consider the system $x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3$, with 0-1 variables $x_1, x_2, x_3$. Each of these constraints can be satisfied individually, they are *locally consistent* in the terminology of Sect. 2.4. However, given a global view of all constraints together, one may deduce that the problem is infeasible. A variant of this constraint is the symmetric `alldifferent` constraint (Régin 1999b).

**Element** The `element` constraint $\texttt{element}(i, l, v)$ expresses that the $i$-th variable in a list of variables $l = [x_1, \ldots, x_n]$ takes the value $v$, i.e., $x_i = v$. Consider an assignment problem where $m$ tasks have to be assigned to $n$ machines. In integer programming, we would use $mn$ binary variables $x_{ij}$ indicating whether or not task $i$ is assigned to machine $j$. If $c_{ij}$ is the corresponding cost, the objective function is $\sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$. In constraint programming, one typically uses $m$ domain variables $x_i$ with domain $D_i = \{1, \ldots, n\}$. Note that $x_i = j$ if and only if $x_{ij} = 1$. Using constraints $\texttt{element}(x_i, [c_{i1}, \ldots, c_{in}], c_i)$, with domain variables $c_i$, the objective function can be stated as $\sum_{i=1}^{m} c_i$.

**Cumulative** The `cumulative` constraint has been introduced to model scheduling problems (Aggoun & Beldiceanu 1993, Caseau & Laburthe 1997, Baptiste, Pape & Nuijten 2001). Suppose there are $n$ tasks. Task $j$ has starting time $s_j$, duration $d_j$ and needs $r_j$ units of a given resource. The constraint $\texttt{cumulative}([\mathtt{s_1}, \ldots, \mathtt{s_n}], [\mathtt{d_1}, \ldots, \mathtt{d_n}], [\mathtt{r_1}, \ldots, \mathtt{r_n}], \mathtt{l}, \mathtt{e})$ states that the tasks have to be executed in such a way that the global resource limit $l$ is never exceeded and $e$ is the end of the schedule

Figure 1: a) Cumulative constraint b) Diffn constraint

**Diffn** The constraint $\texttt{diffn}([[o_{11}, \ldots, o_{1n}, l_{11}, \ldots, l_{1n}], \ldots, [o_{m1}, \ldots, o_{mn}, l_{m1}, \ldots, l_{mn}]])$ states that $m$ rectangles in $n$-dimensional space should not overlap (Beldiceanu & Contejean 1994, Beldiceanu & Carlsson 2001). Here, $o_{ij}$ gives the origin and $l_{ij}$ the length of the rectangle $i$ in dimension $j$, see Fig. 1b). Applications of this constraint include resource allocation and packing problems. Beldiceanu, Qi & Thiel (2001) consider non-overlapping constraints between convex polytopes.

**Cycle** The `cycle` constraint allows one to define cycles in a directed graph (Beldiceanu & Contejean 1994, Caseau & Laburthe 1997, Bourreau 1999). For each node in the graph, one introduces a variable

$s_i$ whose domain contains the nodes that can be reached from node $i$. The constraint $\texttt{cycle}(\texttt{k}, [\texttt{s}_1, \ldots, \texttt{s}_n])$ holds if the variables $s_i$ are instantiated in such a way that precisely $k$ cycles are obtained. A typical application of this constraint are

vehicle routing problems.

**Cardinality** The `cardinality` constraint restricts the number of times a value is taken by a number of variables (Beldiceanu & Contejean 1994, Régin 1996, Régin & Puget 1997, Régin 1999a). Application areas include personnel planning and sequencing problems. An extension of `cardinality` is the `sequence` constraint that allows one to define complex patterns on the values taken by a sequence of variables (Beldiceanu, Aggoun & Contejean 1996).

**Sortedness** The `sort` constraint $\texttt{sort}(x_1, \ldots, x_n, y_1, \ldots, y_n)$ expresses that the $n$-tuple $(y_1, \ldots, y_n)$ is obtained from the $n$-tuple $(x_1, \ldots, x_n)$ by sorting the elements in non-decreasing order (Bleuzen-Guernalec & Colmerauer 2000, Mehlhorn & Thiel 2000). It was introduced in (Older, Swinkels & van Emden 1995) to model and solve job-shop scheduling problems. Zhou (1997) considered a variant with $3n$ variables that makes explicit the permutation linking the $x$'s and $y$'s.

**Flow** The `flow` constraint can be used to model flows in generalized networks (Bockmayr, Pisaruk & Aggoun 2001). In particular, it can handle conversion nodes that arise when modeling production processes. A typical application area is supply chain optimization.

This list of global constraints is not exhaustive. Various other constraints have been proposed in the literature, e.g. (Régin & Rueher 2000, Beldiceanu 2001). A classification scheme for global constraints that subsumes a variety of the existing constraints (but not all of them) is introduced in Beldiceanu (2000).

## 2.4 Local Consistency

From a declarative point of view, a constraint $c(x_1, \ldots, x_n)$ defines a relation on the Cartesian product $D_1 \times \cdots \times D_n$ of the corresponding domains. In general, it is computationally prohibitive to determine directly the tuples $(v_1, \ldots, v_n)$ that satisfy the constraint. Typically, constraint programming systems try to *filter* the domains $D_j$, i.e., to remove values $v_j$ that cannot occur in a solution.

A constraint $c(x_1, \ldots, x_n)$ is *generalized arc consistent* (Mohr & Masini 1988) if for any variable $x_i, i = 1, \ldots, n$, and any value $v_i \in D_i$, there exist values $v_j \in D_j$, for all $j = 1, \ldots, n$ with $j \neq i$, such that $c(v_1, \ldots, v_n)$ holds.

Generalized arc consistency is a basic concept in constraint reasoning. Stronger notions of consistency have been introduced in the literature, like path consistency, $k$-consistency, or $(i, j)$-consistency. Freuder (1985) introduced $(i, j)$*-consistency* for binary constraints. Given values for $i$ variables, satisfying the constraints on those variables, and given any other $j$ (or fewer) variables, there exist values for those $j$ variables such that the $i + j$ values taken together satisfy all constraints on the $i + j$ variables. With this definition, $k$*-consistency* is the same as $(k - 1, 1)$-consistency. *Path consistency* corresponds to 3- resp. $(2, 1)$-consistency, and *arc consistency* to 2- resp. $(1, 1)$-consistency.

A problem can be made arc consistent by removing inconsistent values from the variable domains, i.e. values that cannot appear in any solution. Achieving $k$-consistency for $k \geq 3$ requires to remove tuples of values (instead of values) from $D_1 \times \cdots \times D_n$. The corresponding algorithms become rather expensive. Therefore, their use in constraint programming is limited. Recently, consistency notions have been introduced that are stronger than arc consistency, but still use only domain filtering (as opposed to filtering the Cartesian product), see (Debruyne & Bessière 2001, Prosser, Stergiou & Walsh 2000).

Bound consistency is a restricted form of generalized arc consistency, where we reason only on the bounds of the variables. Assume that $D_j$ is totally ordered, typically $D_j \subset \mathbb{Z}$. A constraint $c(x_1, \ldots, x_n)$ is *bound consistent* (Puget 1998) if for any variable $x_i, i = 1, \ldots, n$, and each bound value $v_i \in \{\min(D_i), \max(D_i)\}$, there exist values $v_j \in [\min(D_i), \max(D_i)]$, for all $j = 1, \ldots, n$ with $j \neq i$, such that $c(v_1, \ldots, v_n)$ holds.

Most work on constraint satisfaction problems in the artificial intelligence community has been done on binary constraints. However, the non-binary case has been receiving more and more attention during the last years (Bessière 1999, Stergiou & Walsh 1999b, Zhang & Yap 2000). Bacchus, Chen, van Beek & Walsh (2002) study two transformations from non-binary to binary constraints, the dual transformation and the hidden (variable) transformation, and formally compare local consistency techniques applied to the original and the transformed problem.

## 2.5 Constraint Propagation

In general, a constraint problem contains many constraints. When achieving arc consistency for one constraint through filtering, other constraints, which were consistent before, may become inconsistent. Therefore, filtering has to be applied repeatedly to constraints that share common variables, until no further domain reduction is possible. This process is called *constraint propagation.*

The classical method for achieving arc consistency is the algorithm AC 3 (Mackworth 1977b). Consider a constraint satisfaction problem $C$ with unary constraints $c_i(x_i)$ and binary constraints $c_{ij}(x_i, x_j)$, where $i < j$. Let $\text{arc}(C)$ denote the set of all ordered pairs $(i, j)$ and $(j, i)$ such that there is a constraint $c_{ij}(x_i, x_j)$ in $C$.

```
Algorithm AC-3 (Mackworth 77)
    for $i \leftarrow 1$ to $n$ do $D_i \leftarrow \{v \in D_i \mid c_i(v)\}$;
    $Q \leftarrow \{(i,j) \mid (i,j) \in \text{arc}(C)\}$;
    while $Q$ not empty do
        select and delete any arc $(i,j)$ from $Q$;
        if revise$(i,j)$ then $Q \leftarrow Q \cup \{(k,i) \mid (k,i) \in \text{arc}(C), k \neq i, k \neq j\}$;
    end while
    end
```

The procedure `revise`$(i, j)$ removes all values $v \in D_i$ for which there is no corresponding value $w \in D_j$ such that $c_{ij}(v, w)$ holds. It returns `true` if at least one value can be removed from $D_i$ , and `false` otherwise. If $e$ is the number of binary constraints and $d$ a bound on the domain size, the complexity of AC 3 is $O(ed^3)$.

Various extensions and refinements of the original algorithm AC 3 have been proposed. Some of these algorithms achieve the optimal worst case complexity $O(ed^2)$, others have an improved average case complexity.

- AC 4 (Mohr & Henderson 1986),
- AC 5 (van Hentenryck & Graf 1992),
- AC 6 (Bessière 1994),
- AC 7 (Bessière, Freuder & Régin 1999),
- AC 2000 and AC 2001 (Bessière & Régin 2001), see also (Zhang & Yap 2001).

Again these papers focus on binary constraints. Extensions to the non-binary case, i.e. generalized arc consistency, are discussed in (Mackworth 1977a, Mohr & Masini 1988, Bessière & Régin 1997, Bessière & Régin 2001).

## 2.6 Filtering Algorithms for Global Constraints

Local consistency for linear arithmetic constraints looks similar to preprocessing in integer programming. Symbolic constraints in constraint programming, however, come with their own filtering algorithms. These are specific to the constraint and therefore can be much more efficient than the general techniques presented in the previous section. Efficient filtering algorithms are a key reason for the success of constraint programming. They make it possible to embed problem-specific algorithms, e.g. from graph theory or scheduling, into a general purpose solver. The goal of this section is to illustrate this on two examples.

### 2.6.1 Alldifferent

First we discuss a filtering algorithm for the `alldifferent` constraint (Régin 1994). Let $x_1, \ldots, x_n$ be the variables and $D_1, \ldots, D_n$ be the corresponding domains. We construct a bipartite graph $G$ to represent the problem in graph-theoretic terms. For each variable $x_j$ we introduce a node on the left, and for each value $v_j \in D_1 \cup \cdots \cup D_n$ a node on the right. There is an edge between $x_i$ and $v_j$ iff $v_j \in D_i$. Then the constraint `alldifferent`$([x_1, \ldots, x_n])$ is satisfiable iff the graph $G$ has a matching covering all the variables.

Our goal is to remove redundant edges from $G$. Suppose we are given a matching $M$ in $G$ covering all the variables. Matching theory tells us that an edge $(x, v) \notin M$ belongs to some maximum matching iff it belongs either to an even alternating cycle or an even alternating path starting in a free node. A node is free if it is not covered by $M$. An alternating path or cycle is a simple path or cycle whose edges alternately belong to $M$ and its complement. We orient the graph by directing all edges in $M$ from right to left, and all edges not in $M$ from left to right. In the directed version of $G$, the first kind of edge is an edge in some strongly connected component, and the second kind of edge is an edge that is reachable from a free node. This yields a linear-time algorithm for removing redundant edges. If no matching $M$ is known, the complexity becomes $O(m\sqrt{n})$, where $m$ is the number of edges in $G$. Assuming that the domain size is bounded by $d$, this can also be stated as $O(dn\sqrt{n})$. If filtering has to be applied several times due to constraint propagation, the overall complexity can be bounded by $O(m^2)$ or $O(d^2n^2)$.

Puget (1998) devised an $O(n \log n)$ algorithm for *bound* consistency of `alldifferent`, a simplified and faster version was obtained in (Mehlhorn & Thiel 2000). Stergiou & Walsh (1999a) compare different notions of consistency for `alldifferent`, see also (van Hoeve 2001).

### 2.6.2 Cumulative

Next we give a short introduction to constraint propagation techniques for resource constraints in scheduling. There is an extensive literature on this subject. We consider here only the simplest example of a one-machine resource constraint in the non-preemptive case. For a more detailed treatment and a guide to the literature, we refer to the recent monograph by Baptiste et al. (2001).

We are given a set of *activities* $\{A_1, \ldots, A_n\}$ that have to be executed on a single *resource* $R$. For each activity, we introduce three domain variables, $\mathit{start}(A_i), \mathit{end}(A_i), \mathit{proc}(A_i)$, that represent the start time, the end time, and the processing time, respectively. The processing time is the difference between the end and the start time, $\mathit{proc}(A_i) = \mathit{end}(A_i) - \mathit{start}(A_i)$. Given an initial release date $r_i$ and a deadline $d_i$, activity $A_i$ has to be performed in the time interval $[r_i, d_i - 1]$. During propagation, these bounds will be updated so that they always denote the current *earliest starting time* and *latest end time* of activity $A_i$.

Different techniques can be applied to filter the domains of the variables $\mathit{start}(A_i)$ and $\mathit{end}(A_i)$ (Baptiste et al. 2001):

**Time tables.** Maintain bound consistency on the formula $\sum_{i=1}^{n} x(A_i, t) \leq 1$, for any time $t$. Here $x(A_i, t)$ is a 0-1 variable indicating whether or not activity $A_i$ executes at time $t$.

**Disjunctive constraint propagation.** Maintain bound consistency on the formula

$$[\mathit{end}(A_i) \leq \mathit{start}(A_j)] \vee [\mathit{end}(A_j) \leq \mathit{start}(A_i)]$$

**Edge finding.** This is one of the key techniques for resource constraints. Given a set of activities $\Omega$, let $r_\Omega$, $d_\Omega$, and $p_\Omega$, respectively, denote the smallest earliest starting time, the largest latest end time, and the sum of the minimal processing times of the activities in $\Omega$. Let $A_i \ll A_j$ mean that $A_i$ executes before $A_j$, and $A_i \ll \Omega$ (resp. $A_i \gg \Omega$) that $A_i$ executes before (resp. after) all activities in $\Omega$. Then the following inferences can be performed:

$$\begin{aligned}
&\forall \Omega, \forall A_i \notin \Omega \quad [d_{\Omega \cup \{A_i\}} - r_\Omega < p_\Omega + p_i] \Rightarrow [A_i \ll \Omega] \\
&\forall \Omega, \forall A_i \notin \Omega \quad [d_\Omega - r_{\Omega \cup \{A_i\}} < p_\Omega + p_i] \Rightarrow [A_i \gg \Omega] \\
&\forall \Omega, \forall A_i \notin \Omega \quad [A_i \ll \Omega] \;\Rightarrow\; [\,\mathit{end}(A_i) \;\leq\; \min_{\emptyset \neq \Omega' \subseteq \Omega}(d_{\Omega'} - p_{\Omega'})\,] \\
&\forall \Omega, \forall A_i \notin \Omega \quad [A_i \gg \Omega] \;\Rightarrow\; [\,\mathit{start}(A_i) \;\geq\; \max_{\emptyset \neq \Omega' \subseteq \Omega}(r_{\Omega'} + p_{\Omega'})\,]
\end{aligned}$$

Edge-finding reasons on sets of activities. Given $n$ activities, a priori $O(n2^n)$ pairs $(A_i, \Omega)$ have to be considered. Carlier & Pinson (1990) present an algorithm that improves the time bounds in $O(n^2)$.

**Not-first, not-last.** The previous techniques try to determine whether an activity $A_i$ *must* be the first (or the last) within a set of activities $\Omega \cup \{A_i\}$. Alternatively, one may try to find out whether $A_i$ *can* be the first (or last) activity in $\Omega \cup \{A_i\}$. If this is not the case, one may deduce that $A_i$ cannot start before the end of at least one activity in $\Omega$ (or that $A_i$ cannot end after the start of at least one activity in $\Omega$) which leads to another set of inference rules.

## 2.7 Modeling in Constraint Programming : An Illustrating Example

To illustrate the variety of models that may exist in constraint programming, we consider the reconstruction of pictures in discrete tomography (Bockmayr, Kasper & Zajac 1998). A *two-dimensional binary picture* is given by a binary matrix $X \in \{0,1\}^{m \times n}$. Intuitively, a pixel is black iff the corresponding matrix element is 1. A binary picture $X$ is

- *horizontally convex*, if the set of 1's in each row is convex, i.e. $x_{ij_1} = x_{ij_2} = 1$ implies $x_{ij} = 1$, for all $1 \leq i \leq m, 1 \leq j_1 < j < j_2 \leq n$.
- *vertically convex*, if the set of 1's in each column is convex, i.e. $x_{i_1 j} = x_{i_2 j} = 1$ implies $x_{ij} = 1$, for all $1 \leq i_1 < i < i_2 \leq m, 1 \leq j \leq n$.
- *connected* or a *polyomino*, if the set of 1's in the matrix is connected with respect to the adjacency relation where each matrix element is adjacent to its two vertical and horizontal neighbours.

Given two vectors $h = (h_1 \ldots, h_m) \in \mathbb{N}^m, v = (v_1, \ldots, v_n) \in \mathbb{N}^n$, the *reconstruction problem* of a binary picture from *orthogonal projections* consists in finding $X \in \{0,1\}^{m \times n}$ such that

- $\sum_{j=1}^{n} x_{ij} = h_i$, for $i = 1, \ldots, m$ (horizontal projections)
- $\sum_{i=1}^{n} x_{ij} = v_j$, for $j = 1, \ldots, n$ (vertical projections)

The complexity of the reconstruction problem depends on the additional properties that are required for the picture (Woeginger 2001).

| | v + h convex | v convex | h convex | no restriction |
|---|---|---|---|---|
| connected | P | NP-complete | NP-complete | NP-complete |
| no restriction | NP-complete | NP-complete | NP-complete | P |

### 2.7.1 0-1 Models

The above properties may be modeled in many different ways. In integer linear programming, one typically uses 0-1 variables $x_{ij}$. The binary picture $X \subseteq \{0,1\}^{m \times n}$ with horizontal and vertical projections $h \in \mathbb{N}^m, v \in \mathbb{N}^n$ is horizontally convex iff the following set of linear inequalities is satisfied:

$$h_i \cdot x_{ik} + \sum_{j=k+h_i}^{n} x_{ij} \leq h_i, \text{ for all } 1 \leq i \leq m, 1 \leq k \leq n.$$

$X$ is vertically convex iff

$$v_j \cdot x_{kj} + \sum_{i=k+v_j}^{m} x_{ij} \leq v_j, \text{ for all } 1 \leq k \leq m, 1 \leq j \leq n.$$

The connectivity of a horizontally convex picture can be expressed as follows:

$$\sum_{j=k}^{k+h_i-1} x_{ij} - \sum_{j=k}^{k+h_i-1} x_{i+1j} \leq h_i - 1, \text{ for all } 1 \leq i \leq m-1, 1 \leq k \leq n - h_i + 1.$$

This leads to $O(mn)$ variables and constraints.

### 2.7.2 Finite Domain Models

In finite domain constraint programming, 0-1 variables are usually avoided. For each row resp. column in the given $m \times n$-matrix, we introduce a finite domain variable

- $x_i \in \{1, \ldots, n\}$, for all $i = 1, \ldots, m$, resp.
- $y_j \in \{1, \ldots, m\}$, for all $j = 1, \ldots, n$.

If $h = (h_1, \ldots, h_n)$ and $v = (v_1, \ldots, v_m)$ are the horizontal and vertical projections, then $x_i = j$ says that the block of $h_i$ 1's for row $i$ starts at column $j$. Analogously, $y_j = i$ expresses that the block of $v_j$ 1's for column $j$ starts in row $i$.

**Conditional propagation.** To ensure that the values of the variables $x_i$ and $y_j$ are compatible with each other, we impose the constraints

$$x_i \leq j < x_i + h_i \iff y_j \leq i < y_j + v_j, \text{ for all } i = 1, \ldots, m, \ j = 1, \ldots, n.$$

Such constraints may be realized by *conditional propagation* rules of the form `if` $C$ `then` $P$, saying that, as soon as the remaining values for the variables satisfy the condition $C$, the constraints $P$ become active. This models horizontal/vertical projections and convexity. To ensure connectivity, we have to forbid that the block in row $i+1$ ends left of the block in row $i$ or that the block in row $i+1$ starts right of the block in row $i$. Negating this disjunction yields the linear inequalities

$$x_i \leq x_{i+1} + h_{i+1} - 1 \text{ and } x_{i+1} \leq x_i + h_i, \text{ for all } i = 1, \ldots, m-1.$$

The above constraints are sufficient to model the reconstruction problem. However, we may try to improve propagation by adding further constraints, which are redundant from the declarative point of view, but provide additional filtering techniques on the procedural side. *Adding redundant constraints* is a standard technique in constraint programming. Again, there is a problem-dependent tradeoff between the cost of the
filtering algorithm and the domain reductions that are obtained.

**Cumulative.** For example, we may use the `cumulative` constraint. We identify each horizontal block in the image with a task $(x_i, h_i, 1)$, which starts at time $x_i$, has duration $h_i$, and requires 1 resource unit. For each column $j$, we introduce an additional task $(j, 1, m - v_j + 1)$, which starts at time $j$, has duration 1, and uses $m - v_j + 1$ resource units. These complementary tasks model vertical projections numbers. The capacity of the resource is $m+1$ and all the tasks end before time $n+1$. Thus, the constraint

$$\begin{array}{llllllll} \texttt{cumulative(} & [\ x_1, & \ldots, & x_m, & 1, & \ldots, & n & ], \\ & [\ h_1, & \ldots, & h_m, & 1, & \ldots, & 1 & ], \\ & [\ 1, & \ldots, & 1, & m - v_1 + 1, & \ldots, & m - v_n + 1 & ], \\ & \quad m+1 & , & & n+1 & & & ) \end{array}$$

models horizontal/vertical projection numbers, and horizontal convexity, see Fig. 2.

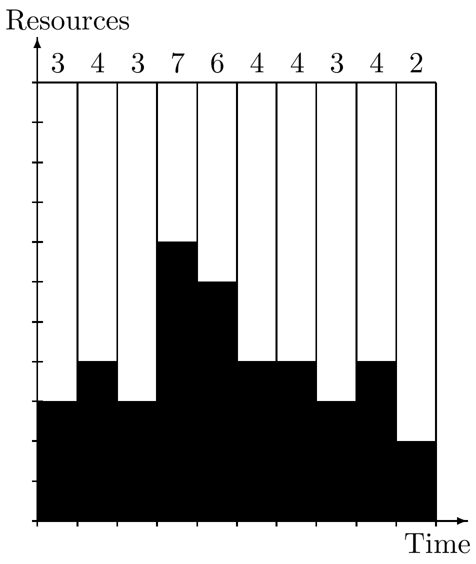

Figure 2: `Cumulative` constraint in discrete tomography

**Diffn.** Another possibility is to use the `diffn` constraint. Here, we look at polyomino reconstruction as packing of two-dimensional rectangles. We model the problem by an extended version of the `diffn` constraint (Beldiceanu & Contejean 1994), involving four arguments. In the first argument, we define the rectangles. For each black horizontal block in the picture, we introduce a rectangle

$$R_i = [x_i, i, h_i, 1],$$

with origins $(x_i, i)$ and lengths $(h_i, 1), i = 1, \ldots, m$. To model vertical convexity, we introduce $2n$ additional rectangles

$$S_{1,j} = [j, 0, 1, l_{j,1}], \quad S_{2,j} = [j, m+1-l_{j,2}, 1, l_{j,2}],$$

which correspond to two white blocks in each column. The variables $l_{jk}$ define the height of these rectangles. To ensure that each white block has a nonzero surface, we introduce two additional rows 0 and $m+1$, see Fig. 3 for an illustration.

The second argument of the `diffn` constraint says that the total number of rows and columns is $m+2$ resp. $n$. In the third argument, we express that the distance between the two white rectangles in column $j$ has to be equal to $v_j$. To model connectivity, we state in the fourth argument that each pair of successive rectangles has a contact in at least one position. This is represented by the list $[\,[1, 2, c_1], ..., [m-1, m, c_{m-1}]\,]$, with domain variables $c_i \geq 1$. Thus, the whole reconstruction problem can be modeled by a single `diffn` constraint:

$$\begin{array}{ll} \texttt{diffn(} & [R_1, \ldots, R_m, S_{1,1}, \ldots, S_{1,n}, S_{2,1}, \ldots, S_{2,n}], \\ & [n, m+2], \\ & [\,[m+1, m+n+1, v_1], \ldots, [m+n, m+2*n, v_n]\,], \\ & [\,[1, 2, c_1], \ldots, [m-1, m, c_{m-1}]\,]\,) \end{array}$$

Note that this model involves only the row variables $x_i$, not the column variables $y_j$. It is also possible to use row and column variables simultaneously. This leads to another model based on a single `diffn` constraint in 3 dimensions, see Fig. 3. Here, the third dimension is used to ensure that row and column variables define the same picture.

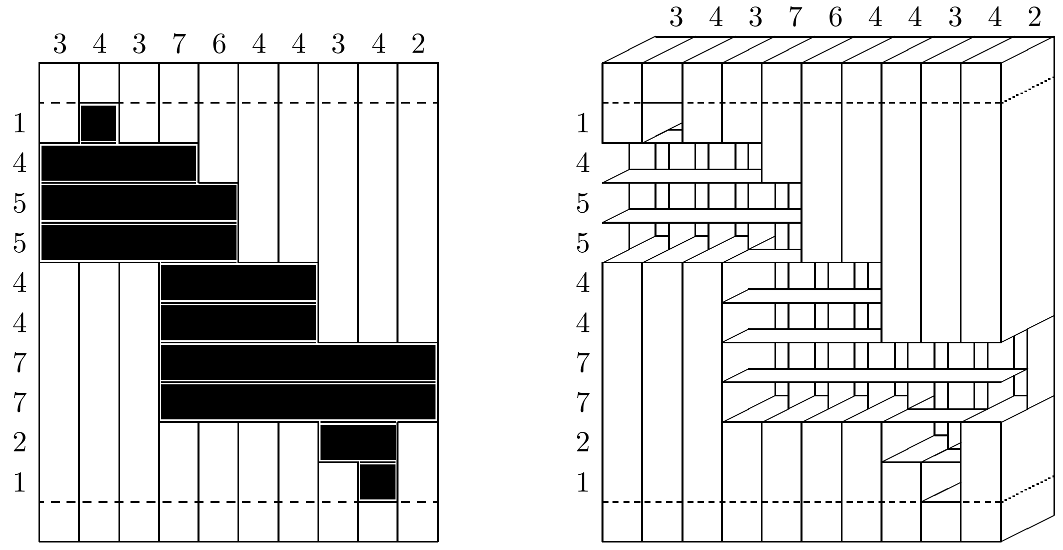

Figure 3: Two- and three-dimensional `diffn` constraint in discrete tomography

## 3 Search

Filtering algorithms reduce the domains of the variables. In general, this is not enough to determine a solution. Therefore, filtering is typically embedded into a search algorithm. Whenever, after filtering, the domain $D$ of a variable $x$ contains more than one value, we may split $D$ into non-empty subdomains $D = D_1 \cup \cdots \cup D_k, k \geq 2$, and consider $k$ new problems $C \cup \{x \in D_1\}, \ldots, C \cup \{x \in D_k\}$. Since $D_i \subsetneq D$, we may apply filtering again in order to get further domain reductions. By repeating this process, we obtain a search tree. There are many different ways to construct and to traverse this tree.

The basic search algorithm in constraint programming is *backtracking*. Variables are instantiated one after the other. As soon as all variables of some constraint have been instantiated, this constraint is evaluated. If it is satisfied, instantiation goes on. Otherwise, at least one variable becomes uninstantiated and a new value is tried.

There are many ways to improve standard backtracking. Following (Dechter 1992), we may distinguish look-ahead and look-back schemes. *Look-ahead* schemes are invoked before extending the current partial solution. The most important techniques are strategies for selecting the next variable or value and maintaining local consistency in order to reduce the search space. *Look-back* schemes are invoked when one has encountered a dead-end and backtracking becomes necessary. This includes heuristics how far to backtrack (back-jumping) or what constraints to record in order to avoid that the same conflict rises again later in the search (no-goods). We focus here on the look-ahead techniques that are widely used in constraint programming. For possible combinations with look-back schemes, we refer to (Jussien, Debruyne & Boizumault 2000, Chen & van Beek 2001).

### 3.1 Variable and Value Ordering

In many cases, the domains $D_1, \ldots, D_k$ are singleton sets that correspond to the different values in the domain $D$. The process of assigning to the variables their possible values and constructing the corresponding search tree is often called *labeling*. During labeling, two important decisions have to be made:

- In which order should the variables be instantiated (variable selection) ?
- In which order should the values be assigned to a selected variable (value selection) ?

These orderings may be defined *statically*, i.e. before starting the search, or *dynamically* by taking into account the current state of the search tree.
Value orderings include:

- Try first the minimal value in the current domain.
- Try first the maximal value in the current domain.
- Try first some value in the middle of the current domain.

Dynamic variable selection strategies may be the following:

- Choose the variable with the smallest domain ("first fail").
- Choose the variable with the smallest domain that occurs in most of the constraints ("most constrained").
- Choose the variable which has the smallest/largest lower/upper bound on its domain.

Variable and value selection strategies have a great impact on the efficiency of the search, see e.g. (Gent, MacIntyre, Prosser, Smith & Walsh 1996, Prosser 1998). Finding good variable or value ordering heuristics is often crucial when solving hard problems.

## 3.2 Complete Search

Whenever we reach a new node of the search tree, typically by assigning a value to a variable, filtering and constraint propagation may be applied again. Depending on the effort we want to spend at the node, we may enforce different levels of consistency.

*Forward checking* (FC) performs arc consistency between the variable that has just been instantiated and the uninstantiated variables. Only those values in the domain of an uninstantiated variable are maintained that are compatible with the current choice for $x$. If the domain of a variable becomes empty, backtracking becomes necessary. Forward-checking for non-binary constraints is described in (Bessière, Meseguer, Freuder & Larrosa 1999), while a general framework for extending forward checking is developed in (Bacchus 2000).

*Full lookahead* or *Maintaining Arc Consistency* (MAC) performs arc consistency for all pairs of uninstantiated variables (in addition to forward checking), see (Sabin & Freuder 1997) for an improved version. *Partial lookahead* is an intermediate form, where only one direction of each edge in the constraint graph is considered.

Again there is a tradeoff between the effort needed to enforce local consistency and the corresponding pruning of the search tree. For a long time, it was believed that FC or FC with Conflict-Directed Backjumping (CBJ) (Prosser 1993), together with the first-fail heuristics, is the most efficient strategy for solving constraint satisfaction problems. (Sabin & Freuder 1994, Bessière & Régin 1996) argued that MAC is more efficient than FC (or FC-CBJ) on hard problems and justified this by a number of empirical results.

### 3.3 Heuristic Search

For many practical problems, complete search methods may be unable to find a solution. In such cases, one may use heuristics in order to guide the search towards regions of the search space that are likely to contain solutions.

*Limited discrepancy search* (LDS) (Harvey & Ginsberg 1995) is based on the idea that a heuristic that normally leads to a solution may fail only because a small number of wrong choices are made. To correct these mistakes, LDS searches paths in the tree that follow the heuristic almost everywhere, except in a limited number of cases where a different choice is made. These are called *discrepancies*. *Depth-bounded discrepancy search* (DDS) is a refinement of LDS that biases search to discrepancies high in the tree (Walsh 1997). It uses an iteratively increasing depth bound. Discrepancies below this bound are forbidden.

*Interleaved depth-first search* (IDFS) (Meseguer 1997) is another strategy to prevent standard depth-first search to fall into mistakes. IDFS searches in parallel several subtrees, called *active*, at certain levels of the trees, called *parallel*. The current active tree is searched depth-first until a leaf is found. If this is a solution, search terminates. Otherwise, the state of the current tree is recorded so that it can be resumed later, and another active subtree is considered. There are two variants of this method. In *Pure* IDFS, all levels are parallel and all subtrees are active. *Limited* IDFS considers a limited number of active subtrees and a limited number of parallel levels, typically at the top of the tree. An experimental comparison of DDS and IDFS can be found in (Meseguer & Walsh 1998).

## 4 Hybrid Methods

Hybrid methods have developed over the last decade in both the constraint programming and optimization communities.

Constraint programmers initially conceived hybrid methods as double modeling approaches, in which some constraints are given both a constraint programming and a mixed integer programming formulation. The two formulations are linked and pass domain reductions and/or infeasibility information to each other. Little & Darby-Dowman (1995) were early proponents of double modeling, along with Rodošek, Wallace & Hajian (1997) and Wallace, Novello & Schimpf (1997), who adapted the constraint logic programming system ECLiPSe so that linear constraints could be dispatched to commercial linear programming solvers (CPLEX and XPRESS-MP). Double modeling requires some knowledge of which formulation is better for a given constraint, an issue studied by Darby-Dowman & Little (1998) and others. The constraints community also began to recognize the parallel between constraint solvers and mixed integer solvers, as evidenced by Bockmayr & Kasper (1998).

In more recent work, Heipcke (1998, 1999) proposed several variations of double modeling. Focacci, Lodi, and Milano (1999a, 1999b, 2000) adapted several optimization ideas to a constraint programming context, such as reduced cost variable fixing, and Refalo (1999) integrated piecewise linear modeling through "tight cooperation" between constraint propagation and a linear relaxation. ILOG's OPL Studio (van Hentenryck 1999) is a commercial modeling language that can invoke both constraint programming (ILOG) and linear programming (CPLEX) solvers and pass a limited amount of information from one to the other.

The mathematical programming community initially conceived hybrid methods as generalizations of branch and cut or a logic-based form of Benders decomposition. Drawing on the work of Beaumont (1990), Hooker (1994) and Hooker & Osorio (1999) proposed mixed logical/linear programming (MLLP) as an extension of mixed integer/linear programming (MILP). Several investigators applied similar hybrid methods to process design and scheduling problems (Cagan, Grossmann & Hooker 1997, Grossmann, Hooker, Raman & Yan 1994, Pinto & Grossmann 1997, Raman & Grossmann 1991, Raman & Grossmann 1993, Raman & Grossmann 1994, Türkay & Grossmann 1996) and a nonlinear version of the method to truss structure design (Bollapragada, Ghattas & Hooker 2001).

The logic-based Benders approach was initially developed for circuit verification by Hooker & Yan (1995) and in general by Hooker 1995, **?** and Hooker & Ottosson (2003). As noted earlier, Jain & Grossmann (2001) found that the Benders approach can dramatically accelerate the solution of a machine scheduling problem. Hooker (2000) observed that the master problem need only be solved once if a Benders cut is generated for each feasible solution found during its solution. Thorsteinsson (2001) obtained an additional order of magnitude speedup for the Jain and Grossmann problem by implementing this idea, which he called *branch and check*. Benders decomposition has recently generated interest on the constraint programming side, as in the work of Eremin & Wallace (2001).

The double modeling and MLLP methods can, by and large, be viewed as special cases of branch-infer-and-relax, which we examine first. We then take up the Benders approach and present Jain and Grossmann's machine scheduling example. Finally, we briefly discuss continuous relaxations of common global constraints and survey some further applications.

## 4.1 Branch, Infer and Relax

Table 2 summarizes the elements of a branch-infer-and-relax (BIR) method. The basic idea is to combine, at each node of the search tree, the filtering and propagation of constraint programming with the relaxation and cutting plane generation of mixed integer programming.

In its simplest form, a BIR method maintains three main data structures: the original set $\mathcal{C}$ of constraints, a constraint store $\mathcal{S}$ that normally contains in-domain constraints, and a relaxation $\mathcal{R}$ that may, for example, contain a linear programming relaxation. The constraint store is itself a relaxation, but for convenience we refer only to $\mathcal{R}$ as the relaxation.

The problem to be solved is to minimize $f(x, y)$ subject to $\mathcal{C}$ and $\mathcal{S}$. The search proceeds by branching on the *search variables* $x$, and the *solution variables* $y$ receive values from the solution of $\mathcal{R}$. The search variables are often discrete, but in a continuous nonlinear problem they may be continuous variables with interval domains, and branching may consist of splitting an interval (van Hentenryck, Michel & Benhamou 1998).

The hybrid algorithm consists of a recursive procedure Search($\mathcal{C}, \mathcal{S}$) and proceeds as follows. Initially one calls Search($\mathcal{C}, \mathcal{S}$) with $\mathcal{C}$ the original set of constraints, and $\mathcal{S}$ containing the initial variable domains. $UB = \infty$ is the initial upper bound on the optimal value. Each call to Search($\mathcal{C}, \mathcal{S}$) executes the following steps.

1. *Infer constraints for the constraint store.* Process each constraint in $\mathcal{C}$ so as to reduce domains in $\mathcal{S}$. Cycle through the constraints of $\mathcal{C}$ using the desired method of constraint propagation (Section 3). If no domains are empty, continue to Step 2.

2. *Infer constraints for the relaxation.* Process each constraint in $\mathcal{C}$ so as to generate a set of constraints to be added to the relaxation $\mathcal{R}$, where $\mathcal{R}$ is initially empty. The constraints in $\mathcal{R}$ contain a subset $x'$ of the variables $x$ and all solution variables $y$, and they may contain new solution variables $u$ that do not appear in $\mathcal{C}$. Constraints in $\mathcal{R}$ that contain no new variables may be added to $\mathcal{C}$ in order to enhance con-

Table 2: Basic elements of branch-infer-and-relax methods.

| | |
|---|---|
| *Constraint store (relaxation)* | Maintain a constraint store (primarily in-domain constraints) and create a relaxation at each node of the search tree. |
| *Branching* | Branch by splitting a non-singleton domain, perhaps using the solution of the relaxation as a guide. |
| *Inference* | Reduce variable domains. Generate cutting planes for the relaxation as well as for constraint propagation. |
| *Bounding* | Solve the relaxation to get a bound. |
| *Feasible solution is obtained at a node...* | When search variables can be assigned values that are consistent with the solution of the relaxation, and all constraints are satisfied. |
| *Node is infeasible...* | When at least one domain is empty or the relaxation is infeasible. |
| *Search backtracks...* | When a node is infeasible, a feasible solution is found at a node, or the tree can be pruned due to bounding. |

straint propagation. Cutting planes, for instance, might be added to both $\mathcal{R}$ and $\mathcal{C}$. Continue to Step 3.

3. *Solve the relaxation.* Minimize the relaxation's objective function $f(x', y, u)$ subject to $\mathcal{R}$. Let $LB$ be the optimal value that results, with $LB = \infty$ if there is no solution. If $LB < UB$ continue to Step 4.

4. *Infer post-relaxation constraints.* If desired, use the solution of the relaxation to generate further constraints for $\mathcal{C}$, such as separating cuts, fixed variables based on reduced costs, and other types of nogoods. Continue to Step 5.

5. *Identify a solution.* If possible, assign some value $\bar{x}$ to $x$ that is consistent with the current domains and the optimal solution $(\bar{x}', \bar{y})$ of the relaxation. If $(x, y) = (\bar{x}, \bar{y})$ is feasible for $\mathcal{C}$, let $UB = LB$, and add the constraint $f(x) < UB$ to $\mathcal{C}$ at all subsequent nodes (to search for a better solution). Otherwise go to Step 6.

6. *Branch.* Branch on some search variable $x_j$ by splitting its domain $D_j$ into smaller domains $D_{j1}, \ldots, D_{jp}$ and calling Search$(\mathcal{C}, \mathcal{S}_k)$ for $k = 1, \ldots, p$, where $\mathcal{S}_k$ is $\mathcal{S}$ augmented with the in-domain constraint $x_j \in D_{jk}$. One can also branch on a violated constraint.

In Step 3, the relaxation $\mathcal{R}$ can depend on the current variable domains. This allows for more flexible modeling. For example, it is often convenient to use conditional constraints of the form $g(x) \rightarrow h(y)$, where $\rightarrow$ means "implies." Such a constraint generates the constraint $h(y)$ for $\mathcal{R}$ when and if the search variable domains become small enough to determine that $g(x)$ is satisfied. If $g(x)$ is not determined to be satisfied, no action is taken.

One common occurrence of conditional constraints is in fixed charge problems, where $cy_1$ is the variable cost of an activity running at level $y_1$, and an additional fixed charge $d$

is incurred when $y_1 > 0$. If $x_1$ is a boolean variable that is true when the fixed charge is incurred, a skeletal fixed charge problem can be written

$$\begin{array}{ll} \text{minimize} & cy_1 + y_2 \\ \text{subject to} & x_1 \rightarrow (y_2 \geq d) \\ & \text{not-}x_1 \rightarrow (y_1 \leq 0) \\ & x_1 \in \{T, F\}, y_1 \in [0, M], y_2 \in [0, \infty) \end{array} \tag{1}$$

where $x_1$ is the only search variable and $y_2$ represents the fixed cost incurred. The constraint $y_2 \geq d$ is added to $\mathcal{R}$ when and if $x_1$ becomes true in the course of the BIR algorithm, and $y_1 \leq 0$ is added when $x_1$ becomes false.

In practice the two conditional constraints of (1) should be written as a single global constraint that will be discussed below in Section 4.4:

$$\text{inequality-or}\left( \begin{bmatrix} x_1 \\ \text{not-}x_1 \end{bmatrix}, \begin{bmatrix} y_2 \geq d \\ y_1 \leq 0 \end{bmatrix} \right)$$

The constraint signals that the two conditional constraints enforce a disjunction $(y_2 \geq d) \vee (y_1 \leq 0)$, which can be given a simple and useful continuous relaxation introduced by Beaumont (1990). (The $\vee$ is an inclusive "or.") In this case the relaxation is $dy_1 \leq My_2$, which the inequality-or constraint generates for $\mathcal{R}$ even before the value of $x_1$ is determined.

## 4.2 Benders Decomposition

Another promising framework for hybrid methods is a logic-based form of Benders decomposition, a well-known optimization technique (Benders 1962; Geoffrion 1972). The problem is written using a partition $[x, y]$ of the variables.

$$\begin{array}{ll} \text{minimize} & f(x, y) \\ \text{subject to} & g_i(x, y), \ \text{all } i \end{array} \tag{2}$$

The basic idea is to search values of $x$ in a *master problem*, and for each value enumerated solve the *subproblem* of finding an optimal $y$. Solution of a subproblem generates a *Benders cut* that is added to the master problem. The cut excludes some values of $x$ that can be no better than the value just tried.

The variable $x$ is initially assigned an arbitrary value $\bar{x}$. This gives rise to a *subproblem* in the $y$ variables:

$$\begin{array}{ll} \text{minimize} & f(\bar{x}, y) \\ \text{subject to} & g_i(\bar{x}, y), \ \text{all } i \end{array} \tag{3}$$

Solution of the subproblem yields a *Benders cut* $z \geq B_{\bar{x}}(x)$ that has two properties:

(a) When $x$ is fixed to any given value $\hat{x}$, the optimal value of (2) is at least $B_{\bar{x}}(\hat{x})$.

(b) When $x$ is fixed to $\bar{x}$, the optimal value of (2) is exactly $B_{\bar{x}}(\bar{x})$.

If the subproblem (3) is infeasible, its optimal value is infinite, and $B_{\bar{x}}(\bar{x}) = \infty$. If the subproblem is unbounded, then (2) is unbounded, and the algorithm terminates. How Benders cuts are generated will be discussed shortly.

In the $K$th iteration, the *master problem* minimizes $z$ subject to all Benders cuts that have been generated so far.

$$\begin{array}{ll} \text{minimize} & z \\ \text{subject to} & z \geq B_{x^k}(x), \ \ k = 1, \ldots, K-1 \end{array} \tag{4}$$

A solution $\bar{x}$ of the master problem is labeled $x^K$, and it gives rise to the next subproblem. The procedure terminates when master problem has the same optimal value as the previous subproblem (infinite if the original problem is infeasible), or when the subproblem is unbounded. The computation can sometimes be accelerated by observing that (b) need not hold until the last iteration.

To obtain a Benders cut from the subproblem (3), one solves the *inference dual* of (3):

$$\begin{array}{ll} \text{maximize} & v \\ \text{subject to} & (g_i(\bar{x}, y), \text{ all } i) \rightarrow (f(\bar{x}, y) \geq v) \end{array} \tag{5}$$

The inference dual seeks the largest lower bound on the subproblem's objective function that can be inferred from its constraints. If the subproblem has a finite optimal value, clearly its dual has the same optimal value. If the subproblem is unbounded (infeasible), then the dual is infeasible (unbounded).

Suppose that $\bar{v}$ is the optimal value of the subproblem dual ($\bar{v} = -\infty$ if the dual is infeasible). A solution of the dual takes the form of a proof that deduces $f(\bar{x}, y) \geq \bar{v}$ from the constraints $g_i(\bar{x}, y)$. The dual solution proves that $\bar{v}$ is a lower bound on the value of the subproblem (3), and therefore a lower bound on the value $z$ of the original problem (2) when $x = \bar{x}$. The key to obtaining a Benders cut is to structure the proof so that it is parameterized by $x$. Thus if $x = \bar{x}$, the proof establishes the lower bound $\bar{v} = B_{\bar{x}}(\bar{x})$ on $z$. If $x$ has some other value $\hat{x}$, the proof establishes a valid lower bound $B_{\bar{x}}(\hat{x})$ on $z$. This yields the Benders cut $z \geq B_{\bar{x}}(x)$.

In classical Benders decomposition, the subproblem is a linear programming problem, and its inference dual is the standard linear programming dual. The Benders cuts take the form of linear inequalities. Benders cuts can also be obtained when the subproblem is a 0-1 programming problem (Hooker 2000, Hooker & Ottosson 2003).

When the subproblem is a constraint programming problem, it is normally checked only for feasibility, and a Benders cut is generated only when the subproblem is infeasible; otherwise the procedure terminates. Constraint programming provides a natural context for generating Benders cuts, because it is already a dual method. It checks infeasibility by trying to construct a proof of infeasibility, which normally takes the form of a proof that some domain must be empty. One can then identify a minimum set $C(x)$ of premises under which the proof is still valid. Thus for any given $x$, the Benders cut $z \geq B_{\bar{x}}(x)$ is defined by setting $B_{\bar{x}}(x) = \infty$ if $C(x)$ is true and $-\infty$ otherwise. Equivalently, one can simply let the Benders cut be not-$C(x)$.

## 4.3 Machine Scheduling Example

A machine assignment and scheduling problem of Jain and Grossmann (2001) illustrates a Benders approach in which the subproblem is solved by constraint programming.

Each job $j$ is assigned to one of several machines $i$ that operate at different speeds. Each assignment results in a processing time $d_{ij}$ and incurs a processing cost $c_{ij}$. There is a release date $r_j$ and a due date $s_j$ for each job $j$. The objective is to minimize processing cost while observing release and due dates.

To formulate the problem, let $x_j$ be the machine to which job $j$ is assigned and $t_j$ the start time for job $j$. It also convenient to let $[t_j \mid x_j = i]$ denote the tuple of start times of jobs assigned to machine $i$, arranged in increasing order of the job number. The problem can be written

$$
\begin{array}{lll}
\text{minimize} & \displaystyle\sum_j c_{x_j j} & (a) \\
\text{subject to} & t_j \geq r_j, \ \ \text{all } j & (b) \\
& t_j + d_{x_j j} \leq S_j, \ \ \text{all } j & (c) \\
& \texttt{cumulative}([t_j \mid x_j = i], [d_{ij} \mid x_j = i], e, 1), \ \ \text{all } i & (d)
\end{array}
\qquad (6)
$$

The objective function (a) measures the total processing cost. Constraints (b) and (c) observe release times and deadlines. The `cumulative` constraint (d) ensures that jobs assigned to each machine are scheduled so that they do not overlap. (Recall that $e$ is a vector of ones.)

The problem has two parts: the assignment of jobs to machines, and the scheduling of jobs on each machine. The assignment problem is treated as the master problem and solved with mixed integer programming methods. Once the assignments are made, the subproblems are dispatched to a constraint programming solver to find a feasible schedule. If there is no feasible schedule, a Benders cut is generated.

Variables $x$ go into the master problem and $t$ into the subproblem. If $x$ has been fixed to $\bar{x}$, the subproblem is

$$
\begin{array}{l}
t_j \geq r_j, \ \ \text{all } j \\
t_j + d_{\bar{x}_j j} \leq S_j, \ \ \text{all } j \\
\texttt{cumulative}([t_j \mid \bar{x}_j = i], [d_{ij} \mid \bar{x}_j = i], e, 1), \ \ \text{all } i
\end{array}
\qquad (7)
$$

The subproblem can be decomposed into smaller problems, one for each machine. If a smaller problem is infeasible for some $i$, then the jobs assigned to machine $i$ cannot all be scheduled on that machine. In fact, going beyond Jain and Grossmann (2001), there may be a subset $J$ of these jobs that cannot be scheduled on machine $i$. This gives rise to a Benders cut stating that at least one of the jobs in $J$ must be assigned to another machine.

$$\bigvee_{j \in J} (x_j \neq i) \qquad (8)$$

Let $x^k$ be the solution of the $k$th master problem, $I^k$ the set of machines $i$ in the resulting subproblem for which the schedule is infeasible, and $J_{ki}$ the infeasible subset for machine $i$. The master problem can now be written,

$$
\begin{array}{ll}
\text{minimize} & \displaystyle\sum_j c_{x_j j} \\
\text{subject to} & \displaystyle\bigvee_{j \in J_{ki}} (x_j \neq i), \ \ i \in I^k, \ k = 1, \ldots, K
\end{array}
\qquad (9)
$$

The master problem can be reformulated for solution with conventional integer programming technology. Let $x_{ij}$ be a 0-1 variable that is 1 when job $j$ is assigned to machine $i$. The master problem (9) can be written

$$
\begin{array}{lll}
\text{minimize} & \displaystyle\sum_{i,j} c_{ij} x_{ij} & (a) \\
\text{subject to} & \displaystyle\sum_{j \in J_{ki}} (1 - x_{ij}) \geq 1, \ \ i \in I^k, \ k = 1, \ldots, K & (b) \\
& \displaystyle\sum_j d_{ij} x_{ij} \leq \max_j \{s_j\} - \min_j \{r_j\}, \ \ \text{all } i & (c) \\
& x_{ij} \in \{0, 1\}, \ \ \text{all } i, j & (d)
\end{array}
$$

Constraints (c) are valid cuts added to strengthen the continuous relaxation. They simply say that the total processing time on each machine must fit between the earliest release time and the latest deadline. Thorsteinsson (2001) reports that (c) is essential to the success of the Benders approach. It therefore seems worthwhile to develop relaxations for other predicates that might appear in the subproblem, such as the full `cumulative` constraint, as opposed to the one-machine constraint used here. Such a relaxation is mentioned in the next section.

## 4.4 Continuous Relaxations for Global Constraints

Continuous relaxations for global constaints can accelerate solution by exploiting substructure in a model. Relaxations have been developed for several constraints, although other constraints have yet to be addressed. Relaxations for many of the constraints discussed below are summarized by Hooker (2000, 2002); see also (Refalo 2000).

The *inequality-or* constraint, discussed above in the context of fixed charge problems, may be written,

$$\texttt{inequality-or}\left(\begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix}, \begin{bmatrix} A^1 y \geq a^1 \\ \vdots \\ A^k y \geq a^k \end{bmatrix}\right)$$

It requires that $x_i$ be true and $A^i y \geq a^i$ be satisfied for at least one $i \in \{1, \ldots, k\}$. A convex hull relaxation can be obtained by introducing new variables, as shown by Balas (1975, 1979). The well-known "big-$M$" lifted relaxation that is weaker than the convex hull relaxation but requires fewer variables. Hooker & Osorio (1999) discuss how to tighten the big-$M$ relaxation.

A *disjunction of single inequalities*

$$(a^1 y \geq \alpha_1) \vee \cdots \vee (a^k \geq \alpha_k)$$

relaxes to a single inequality, as shown by Beaumont (1990). Hooker & Osorio (1999) provide a closed-form expression for a tighter right-hand side.

*Cardinality rules* provide for more complex logical conditions:

If at least $k$ of $x_1, \ldots, x_m$ are true, then at least $\ell$ of $y_1, \ldots, y_n$ are true.

Yan & Hooker (1999) describe a convex hull relaxation for such rules. Their result has been generalized by Balas, Bockmayr, Pisaruk & Wolsey (2002).

*Piecewise linear functions* can easily be given a convex hull relaxation that, when properly used, can result in faster solution than mixed integer programming with specially ordered sets of type 2 (Ottosson, Thorsteinsson & Hooker 1999). Refalo (1999) shows how to use the relaxation in "tight cooperation" with domain reduction to obtain maximum benefit.

The *all-different* constraint can be given a convex hull relaxation described by Hooker (2000) and Williams & Yan (2001).

The *element* constraint is particularly useful for implementing variable indices. An expression of the form $u_y$ can be encoded by replacing it with the variable $z$ and adding the constraint $\texttt{element}(y, (u_1, \ldots, u_n), z)$. Here $u_1, \ldots, u_n$ may be constants or variables. Hooker, Ottosson, Thorsteinsson & Kim (1999) present various relaxations of the element constraint, including a convex hull relaxation when the variables $u_1, \ldots, u_n$ have the same upper bound (Hooker 2000).

The important *cumulative* constraint has been given three relaxations by Hooker & Yan (2001). One relaxation consists of facet defining inequalities in the special case in which some jobs have identical characteristics.

*Lagrangean relaxation* can be employed in a hybrid setting. Sellmann & Fahle (2001) use it to strengthen propagation of knapsack constraints in an automatic recording problem. Benoist, Laburthe & Rottembourg (2001) apply it to a traveling tournament problem. It is unclear whether this work suggests a general method for integrating Lagrangean relaxation with constraint propagation.

### 4.5 Other Applications

Hybrid methods have been applied to a number of problems other than those already mentioned. Transportation applications include vehicle routing with time windows (Caseau, Silverstein & Laburthe 2001, Focacci, Lodi & Milano 1999b), vehicle routing combined with inventory management (Lau & Liu 1999), crew rostering (Caprara & et al. 1998, Junker, Karisch, Kohl, Vaaben, Fahle & Sellmann 1999), the traveling tournament problem (Benoist et al. 2001), and the classical transportation problem with piecewise linear costs (Refalo 1999).

Scheduling applications include machine scheduling (Heipcke 1998, Raman & Grossmann 1993), sequencing with setups (Focacci, Lodi & Milano 1999a), hoist scheduling (Rodošek & Wallace 1998), employee scheduling (Partouche 1998), dynamic scheduling (Sakkout, Richards & Wallace 1998), and lesson timetables (Focacci et al. 1999a). Production scheduling applications include scheduling with resource constraints (Pinto & Grossmann 1997) and with labor resource constraints in particular (Heipcke 1999), two-stage process scheduling (Jain & Grossmann 2001), machine allocation and scheduling (Lustig & Puget 1999), production flow planning with machine assignment (Heipcke 1999), scheduling with piecewise linear costs (Ottosson et al. 1999), scheduling with earliness and tardiness costs (Beck 2001), and organization of a boat party (Hooker & Osorio 1999, Smith, Brailsford, Hubbard & Williams 1996).

Other areas of application include inventory management (Rodošek et al. 1997), office cleaning (Heipcke 1999), product configuration (Ottosson & Thorsteinsson 2000), generalized assignment problems (Darby-Dowman, Little, Mitra & Zaffalon 1997), multidimensional knapsack problems (Osorio & Glover 2001), automatic recording of television shows (Sellmann & Fahle 2001), resource allocation in ATM networks (Lauvergne, David & Boizumault 2001), and assembly line balancing (Bockmayr & Pisaruk 2001).

Benders-based hybrid methods provide a natural decomposition for manufacturing and supply chain problems in which resource assignment issues combine with scheduling issues. Recent industrial applications along this line include automobile assembly (Beauseigneur & Noiré 2003), polypropylene manufacture (Timpe 2003), and paint production (Constantino 2003).

## 5 Constraint Programming Languages and Systems

In constraint programming, the term "programming" has two different meanings (Lustig & Puget 1999), see also Sect. 1.1:

- Mathematical programming, i.e. solving mathematical optimization problems.
- Computer programming, i.e. writing computer programs in a programming language.

Constraint programming makes contributions on both sides. On the one hand, it provides a new approach to solving discrete optimization problems. On the other hand, the constraint solving algorithms are integrated into a high-level programming language so that they become easily accessible even to a non-expert user.

There are different ways of integrating constraints into a programming language. Early work in this direction was done by Laurière (1978) in the language ALICE. Constraint programming as it is known today first appeared in the form of *constraint logic programming*, with logic programming as the underlying programming language paradigm (Colmerauer 1987, Jaffar & Lassez 1987). In logic programming (Prolog), search and backtracking are built into the language. This greatly facilitates the development of search algorithms. Constraint satisfaction techniques have been studied in artificial intelligence since the early 70's. They were first introduced into logic programming in the CHIP system (Dincbas, van Hentenryck, Simonis, Aggoun & Graf 1988, van Hentenryck 1989). Puget (1994) showed that the basic concepts of constraint logic programming can also be realized in a `C++` environment, which lead to the development of ILOG Solver. Another possible approach is the *concurrent constraint programming* paradigm (cc) (Saraswat 1993), with systems such as cc(FD) (van Hentenryck, Saraswat & Deville 1998) or Oz (Smolka 1995).

## 5.1 High-level Modeling Languages

The standard way to develop a constraint program is to use the host programming language in order to build the constraint model and to specify the search strategy. In recent years, new declarative languages have been proposed on top of existing constraint programming systems, which allow one to define both the constraints and the search strategy in a very high-level way. Examples include OPL (van Hentenryck 1999), PLAM (Barth & Bockmayr 1998), or more specifically for search SALSA (Laburthe & Caseau 1998).

Both OPL and PLAM support high-level algebraic and set notation, similarly to algebraic modeling languages in mathematical programming. In addition to arithmetic constraints, OPL or PLAM also support the different symbolic constraints that are typical for constraint programming. Furthermore, they allow the user to specify search procedures in a high-level way. While PLAM relies on logic programming, OPL provides a set of high-level constructs to specify complex search strategies.

As an example, we present an OPL model for solving a job-shop scheduling problem (van Hentenryck, Michel, Perron & Régin 1999), see Fig. 4. Part 1of the model contains various declarations concerning machines, jobs, tasks, the duration of the tasks, and the resources they require. Part 2 declares the activities and resources of the problem, which are predefined concepts in OPL. In Part 3, symbolic precedence and resource constraints are stated. Finally, the search strategy is specified in Part 4. It uses limited discrepancy search and a ranking of the resources.

While high-level languages such as OPL provide a very elegant modeling and solution environment, particular problems that require specific solution strategies and heuristics may not be expressible in this high-level framework. In that case, the user has to work directly with the underlying constraint programming system.

## 5.2 Constraint Programming Systems

We finish this section with a short overview of constraint programming systems that are currently available, see Tab. 3. For a more detailed description, we refer to the corresponding web sites.

```
int nbMachines = ...
range Machines 1..nbMachines;
int nbJobs = ...;
range Jobs 1..nbJobs;
int nbTasks = ...;
range Tasks 1..nbTasks;
Machines resource[Jobs,Tasks] = ...;
int+ duration[Jobs,Tasks] = ...;
int totalDuration = sum(j in Jobs, t in Tasks) duration[j,t];

scheduleHorizon = totalDuration;
Activity task[j in Jobs, t in Tasks](duration[j,t]);
Activity makespan(0);
UnaryResource tool[Machines];

minimize
   makespan.end
subject to {
   forall(j in Jobs)
      task[j,nbTasks] precedes makespan;
   forall(j in Jobs)
      forall(t in 1..nbTasks-1)
         task[j,t] precedes task[j,t+1];
   forall(j in Jobs)
       forall(t in Tasks)
          task[j,t] requires tool[resource[j,t]];
};

search {
   LDSearch() {
      forall(r in Machines ordered by increasing localSlack(tool[r]))
         rank(tool[r]);
   }
}
```

Figure 4: A Job-Shop Model in OPL (van Hentenryck et al. 99)

| System | Availability | Constraints | Language | Web site |
|---|---|---|---|---|
| B-prolog | commercial | Finite domain | Prolog | www.probp.com |
| CHIP | commercial | Finite domain,<br>Boolean,<br>Linear rational<br>Hybrid | Prolog,<br>C, C++ | www.cosytec.com |
| Choco | free | Finite domain | Claire | www.choco-constraints.net |
| Eclipse | free for<br>non-profit | Finite domain,<br>Hybrid | Prolog | www.icparc.ic.ac.uk/eclipse/ |
| GNU Prolog | free | Finite domain | Prolog | gnu-prolog.inria.fr |
| IF/Prolog | commercial | Finite domain<br>Boolean,<br>Linear arithmetic | Prolog | www.ifcomputer.co.jp |
| ILOG | commercial | Finite domain,<br>Hybrid | C++,<br>Java | www.ilog.com |
| NCL | commercial | Finite domain |  | www.enginest.com |
| Mozart | free | Finite domain | Oz | www.mozart-oz.org |
| Prolog IV | commercial | Finite domain,<br>Linear/nonlinear<br>interval arithmetic | Prolog | prologianet.univ-mrs.fr |
| Sicstus | commercial | Finite domain,<br>Boolean, linear<br>real/rational | Prolog | www.sics.se/sicstus/ |

Table 3: Constraint programming systems

## References

Aggoun, A. & Beldiceanu, N. 1993. Extending CHIP in order to solve complex scheduling and placement problems, *Mathl. Comput. Modelling* **17**(7): 57 – 73.

Bacchus, F. 2000. Extending forward checking, *Principles and Practice of Constraint Programming, CP'2000, Singapore*, Springer, LNCS 1894, pp. 35–51.

Bacchus, F., Chen, X., van Beek, P. & Walsh, T. 2002. Binary vs. non-binary constraints, *Artificial Intelligence* . To appear.

Balas, E. 1975. Disjunctive programming: Cutting planes from logical conditions, *in* O. L. Mangasarian, R. R. Meyer & S. M. Robinson (eds), *Nonlinear Programming 2*, Academic Press, New York, pp. 279–312.

Balas, E. 1979. Disjunctive programming, *Annals of Discrete Mathematics* **5**: 3 – 51.

Balas, E., Bockmayr, A., Pisaruk, N. & Wolsey, L. 2002. On unions and dominants of polytopes, *Discussion Paper No. 2002/8*, CORE. Also available as Management Science Report #MSRR-669, GSIA, Carnegie-Mellon-University.

Baptiste, P. & Pape, C. L. 2000. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems, *Constraints* **5**(1/2): 119 – 139.

Baptiste, P., Pape, C. L. & Nuijten, W. 2001. *Constraint-Based Scheduling*, Vol. 39 of *International Series in Operations Research and Management Science*, Kluwer.

Barth, P. & Bockmayr, A. 1998. Modelling discrete optimisation problems in constraint logic programming, *Annals of Operations Research* **81**: 467–496.

Beaumont, N. 1990. An algorithm for disjunctive programs, *Europ. J. Oper. Res.* **48**: 362 – 371.

Beauseigneur, M. & Noiré, S. 2003. Solving the car sequencing problem using combined CP/MIP for PSA Peugeot Citroën, LISCOS Project Summary Meeting, Brussels (28 March 2003).

Beck, C. 2001. A hybrid approach to scheduling with earliness and tardiness costs, *Third International Workshop on Integration of AI and OR Techniques (CPAIOR01).*

Beldiceanu, N. 2000. Global constraints as graph properties on a structured network of elementary constraints of the same type, *Principles and Practice of Constraint Programming, CP'2000, Singapore*, Springer, LNCS 1894, pp. 52–66.

Beldiceanu, N. 2001. Pruning for the *minimum* constraint family and for the *number of distinct values* constraint family, *Principles and Practice of Constraint Programming, CP'2001, Paphos, Cyprus*, Springer, LNCS 2239, pp. 211–224.

Beldiceanu, N., Aggoun, A. & Contejean, E. 1996. Introducing constrained sequences in CHIP, *Technical report*, COSYTEC S.A., Orsay, France.

Beldiceanu, N. & Carlsson, M. 2001. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint, *Principles and Practice of Constraint Programming, CP'2001, Paphos, Cyprus*, Springer, LNCS 2239, pp. 377–391.

Beldiceanu, N. & Contejean, E. 1994. Introducing global constraints in CHIP, *Mathl. Comput. Modelling* **20**(12): 97 – 123.

Beldiceanu, N., Qi, G. & Thiel, S. 2001. Non-overlapping constraints between convex polytopes, *Principles and Practice of Constraint Programming, CP'2001, Paphos, Cyprus*, Springer, LNCS 2239, pp. 392–407.

Benoist, T., Laburthe, F. & Rottembourg, B. 2001. Lagrange relaxation and constraint programming collaborative schemes for travelling tournament problems, *Third International Workshop on Integration of AI and OR Techniques (CPAIOR01).*

Bessière, C. 1994. Arc-consistency and arc-consistency again, *Artificial Intelligence* **65**: 179 – 190.

Bessière, C. 1999. Non-binary constraints, *Principles and Practice of Constraint Programming, CP'99, Alexandria, VA*, Springer, LNCS 1713, pp. 24–27.

Bessière, C., Freuder, E. & Régin, J.-C. 1999. Using constraint metaknowledge to reduce arc consistency computation, *Artificial Intelligence* **107**: 125–148.

Bessière, C., Meseguer, P., Freuder, E. C. & Larrosa, J. 1999. On forward checking for non-binary constraint satisfaction, *Principles and Practice of Constraint Programming, CP'99, Alexandria, VA*, Springer, LNCS 1713, pp. 88–102.

Bessière, C. & Régin, J.-C. 1996. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems, *Principles and Practice of Constraint Programming, CP'96, Cambridge, MA*, Springer, LNCS 1118, pp. 61–75.

Bessière, C. & Régin, J.-C. 1997. Arc consistency for general constraint networks: preliminary results, *15th Intern. Joint Conf. Artificial Intelligence, IJCAI'97, Nagoya, Japan*, Vol. 1, pp. 398–404.

Bessière, C. & Régin, J.-C. 2001. Refining the basic constraint propagation algorithm, *17th Intern. Joint Conf. Artificial Intelligence, IJCAI'01, Seattle*, Vol. 1, pp. 309–315.

Bleuzen-Guernalec, N. & Colmerauer, A. 2000. Optimal narrowing of a block of sortings in optimal time, *Constraints* **5**(1/2): 85–118.

Bockmayr, A. & Kasper, T. 1998. Branch and infer: A unifying framework for integer and finite domain constraint programming, *INFORMS Journal on Computing* **10**: 287–300.

Bockmayr, A., Kasper, T. & Zajac, T. 1998. Reconstructing binary pictures in discrete tomography, *16th European Conference on Operational Research, EURO XVI, Bruxelles.*

Bockmayr, A. & Pisaruk, N. 2001. Solving assembly line balancing problems by combining IP and CP, *Sixth Annual Workshop of the ERCIM Working Group on Constraints, Prague.* http://arXiv.org/abs/cs.DM/0106002.

Bockmayr, A., Pisaruk, N. & Aggoun, A. 2001. Network flow problems in constraint programming, *Principles and Practice of Constraint Programming, CP'2001, Paphos, Cyprus*, Springer, LNCS 2239, pp. 196 – 210.

Bollapragada, S., Ghattas, O. & Hooker, J. N. 2001. Optimal design of truss structures by mixed logical and linear programming, *Operations Research* **49**: 42–51.

Bourreau, E. 1999. *Traitement de contraintes sur les graphes en programmation par contraintes*, PhD thesis, L.I.P.N., Univ. Paris 13.

Cagan, J., Grossmann, I. E. & Hooker, J. N. 1997. A conceptual framework for combining artificial intelligence and optimization in engineering design, *Research in Engineering Design* **49**: 20–34.

Caprara, A. & et al. 1998. Integrating constraint logic programming and operations research techniques for the crew rostering problem, *Software - Practice and Experience* **28**: 49–761.

Carlier, J. & Pinson, E. 1990. A practical use of Jackson's preemptive schedule for solving the job-shop problem, *Annals of Operations Research* **26**: 269–287.

Caseau, Y. & Laburthe, F. 1997. Solving small TSP's with constraints, *14th International Conference on Logic Programming, ICLP'97, Leuven*, MIT Press, pp. 316–330.

Caseau, Y., Silverstein, G. & Laburthe, F. 2001. Learning hybrid algorithms for vehicle routing problems, *Third International Worksho on Integration of AI and OR Techniques (CPAIOR01)*.

Chen, X. & van Beek, P. 2001. Conflict-directed backjumping revisited, *Journal of Artificial Intelligence Research* **14**: 53–81.

Colmerauer, A. 1987. Introduction to PROLOG III, *4th Annual ESPRIT Conference, Bruxelles*, North Holland. See also: Comm. ACM 33 (1990), 69-90.

Constantino, M. 2003. Integrated lot-sizing and scheduling of Barbot's paint production using combined MIP/CP, LISCOS Project Summary Meeting, Brussels (28 March 2003).

Darby-Dowman, K. & Little, J. 1998. Properties of some combinatorial optimization problems and their effect on the performance of integer programming and constraint logic programming, *INFORMS Journal on Computing* **10**: 276–286.

Darby-Dowman, K., Little, J., Mitra, G. & Zaffalon, M. 1997. Constraint logic programming and integer programming approaches and their collaboration in solving an assignment scheduling problem, *Constraints* **1**: 245–264.

Debruyne, R. & Bessière, C. 2001. Domain filtering consistencies, *Journal of Artificial Intelligence Research* **14**: 205 – 230.

Dechter, R. 1992. Constraint networks, *in* S. Shapiro (ed.), *Encyclopedia of artificial intelligence*, Vol. 1, Wiley, pp. 276 – 285.

Dincbas, M., van Hentenryck, P., Simonis, H., Aggoun, A. & Graf, T. 1988. The constraint logic programming language CHIP, *Fifth Generation Computer Systems, Tokyo, 1988*, Springer.

Eremin, A. & Wallace, M. 2001. Hybrid Benders decomposition algorithms in constraint logic programming, *Seventh International Conference on Principles and Practice of Constraint Programming (CP2001)*.

Focacci, F., Lodi, A. & Milano, M. 1999a. Cost-based domain filtering, *Principles and Practice of Constraint Programming*, Vol. 1713 of *Lecture Notes in Computer Science*, pp. 189–203.

Focacci, F., Lodi, A. & Milano, M. 1999b. Solving TSP with time windows with constraints, *16th International Conference on Logic Programming*, Las Cruces, NM.

Focacci, F., Lodi, A. & Milano, M. 2000. Cutting planes in constraint programming: An hybrid approach, *Principles and Practice of Constraint Programming, CP'2000, Singapore*, Springer, LNCS 1894, pp. 187 – 201.

Freuder, E. C. 1985. A sufficient condition for backtrack-bounded search, *Journal of the Association for Computing Machinery* **32**(4): 755 – 761.

Gent, I. P., MacIntyre, E., Prosser, P., Smith, B. M. & Walsh, T. 1996. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem, *Principles and Practice of Constraint Programming, CP'96, Cambridge, MA*, Springer, LNCS 1118, pp. 179–193.

Grossmann, I. E., Hooker, J. N., Raman, R. & Yan, H. 1994. Logic cuts for processing networks with fixed charges, *Computers and Operations Research* **421**: 265–279.

Harvey, W. D. & Ginsberg, M. L. 1995. Limited discrepancy search, *14th Intern. Joint Conf. Artificial Intelligence, IJCAI'95, Montreal*, Vol. 1, pp. 607–615.

Heipcke, S. 1998. Integrating constraint programming techniques into mathematical programming, *Proceedings, 13th European Conference on Artificial Intelligence*, Wiley, New York, pp. 259–260.

Heipcke, S. 1999. *Combined modelling and problem solving in mathematical programming and constraint programming*, PhD thesis, Univ. Buckingham.

Hooker, J. N. 1994. Logic-based methods for optimization, *in* A. Borning (ed.), *Principles and Practice of Constraint Programming*, Vol. 874 of *Lecture Notes in Computer Science*, Springer, pp. 336–349.

Hooker, J. N. 1995. Logic-based Benders decomposition, *INFORMS National Meeting.*

Hooker, J. N. 2000. *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*, John Wiley and Sons.

Hooker, J. N. 2002. Logic, optimization and constraint programming, *INFORMS Journal on Computing* **14**: 295–321.

Hooker, J. N. 2003. A framework for integrating solution methods, in H. K. Bhargava and Mong Ye, eds., *Computational Modeling and Problem Solving in the Networked World (Proceedings of ICS2003)*, Kluwer, pp. 3–30.

Hooker, J. N. & Ottosson, G. 2003. Logic-based Benders decomposition, *Mathematical Programming* **96**: 33–60.

Hooker, J. N., Ottosson, G., Thorsteinsson, E. & Kim, H.-J. 1999. On integrating constraint propagation and linear programming for combinatorial optimization, *Proceeedings, 16th National Conference on Artificial Intelligence*, MIT Press, Cambridge, MA, pp. 136–141.

Hooker, J. N. & Yan, H. 1995. Logic circuit verification by Benders decomposition, *in* V. Saraswat & P. V. Hentenryck (eds), *Principles and Practice of Constraint Programming: The Newport Papers*, MIT Press, Cambridge, MA, pp. 267–288.

Hooker, J. N. & Yan, H. 2001. A continuous relaxation for the cumulative constraint. Manuscript.

Hooker, J. & Osorio, M. A. 1999. Mixed logical/linear programming, *Discrete Applied Mathematics* **96-97**: 395–442.

Jaffar, J. & Lassez, J.-L. 1987. Constraint logic programming, *Proc. 14th ACM Symp. Principles of Programming Languages*, Munich.

Jain, V. & Grossmann, I. E. 2001. Algorithms for hybrid MILP/CP models for a class of optimization problems, *INFORMS J. Computing* **13**(4): 258 – 276.

Junker, U., Karisch, S. E., Kohl, N., Vaaben, B., Fahle, T. & Sellmann, M. 1999. A framework for constraint programming based column generation, *in* J. Jaffar (ed.), *Principles and Practice of Constraint Programming*, Vol. 1713 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 261–274.

Jussien, N., Debruyne, R. & Boizumault, P. 2000. Maintaining arc-consistency within dynamic backtracking, *Principles and Practice of Constraint Programming, CP'2000, Singapore*, Springer, LNCS 1894, pp. 249–261.

Laburthe, F. & Caseau, Y. 1998. SALSA : A language for search algorithms, *Principles and Practice of Constraint Programming, CP'98, Pisa*, Springer, LNCS 1520, pp. 310 – 324.

Lau, H. C. & Liu, Q. Z. 1999. Collaborative model and algorithms for supporting real-time distribution logistics systems, *CP99 post-Conference Workshop on Large Scale Combinatorial Optimization and Constraints*, pp. 30–44.

Laurière, J.-L. 1978. A language and a program for stating and for solving combinatorial problems, *Artificial Intelligence* **10**: 29–127.

Lauvergne, M., David, P. & Boizumault, P. 2001. Resource allocation in ATM networks: A hybrid approach, *Third International Workshop on the Integration of AI and OR Techniques (CPAIOR 2001).*

Little, J. & Darby-Dowman, K. 1995.

The significance of constraint logic programming to operational research, *in* M. Lawrence & C. Wilson (eds), *Operational Research*, pp. 20–45.

Lustig, I. J. & Puget, J.-F. 1999. Program != program: Constraint programming and its relationship to mathematical programming, *Technical report*, ILOG. To appear in *Interfaces.*

Mackworth, A. 1977a. On reading sketch maps, *5th Intern. Joint Conf. Artificial Intelligence, IJCAI'77, Cambridge MA*, pp. 598–606.

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

Mehlhorn, K. & Thiel, S. 2000. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint, *Principles and Practice of Constraint Programming, CP'2000, Singapore*, Springer, LNCS 1894, pp. 306–319.

Meseguer, P. 1997. Interleaved depth-first search, *15th Intern. Joint Conf. Artificial Intelligence, IJCAI'97, Nagoya, Japan*, Vol. 2, pp. 1382–1387.

Meseguer, P. & Walsh, T. 1998. Interleaved and discrepancy based search, *13th Europ. Conf. Artificial Intelligence, Brighton, UK*, 229-233, p. John Wiley and Sons.

Mohr, R. & Henderson, T. C. 1986. Arc and path consistency revisited, *Artificial Intelligence* **28**: 225 – 233.

Mohr, R. & Masini, G. 1988. Good old discrete relaxation, *Proc. 8th European Conference on Artificial Intelligence*, Pitman Publishers, Munich, FRG, pp. 651–656.

Older, W. J., Swinkels, G. M. & van Emden, M. H. 1995. Getting to the real problem: experience with BNR prolog in OR, *Practical Application of Prolog, PAP'95, Paris.*

Osorio, M. A. & Glover, F. 2001. Logic cuts using surrogate constraint analysis in the multidimensional knapsack problem, *Third International Workshop on Integration of AI and OR Techniques (CPAIOR01).*

Ottosson, G. & Thorsteinsson, E. 2000. Linear relaxations and reduced-cost based propagation of continuous variable subscripts, *Second International Workshop on Integration of AI*

*and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR2000*, University of Paderborn.

Ottosson, G., Thorsteinsson, E. & Hooker, J. N. 1999. Mixed global constraints and inference in hybrid CLP-IP solvers, *CP99 Post-Conference Workshop on Large Scale Combinatorial Optimization and Constraints*, pp. 57–78.

Partouche, A. 1998. *Planification d'horaires de travail*, PhD thesis, Université Paris-Daphiné, U. F. R. Sciences des Organisations.

Pinto, J. M. & Grossmann, I. E. 1997. A logic-based approach to scheduling problems with resource constraints, *Computers and Chemical Engineering* **21**: 801–818.

Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problem, *Computational Intelligence* **9**: 268–299.

Prosser, P. 1998. The dynamics of dynamic variable ordering heuristics, *Principles and Practice of Constraint Programming, CP'98, Pisa*, Springer, LNCS 1520, pp. 17–23.

Prosser, P., Stergiou, K. & Walsh, T. 2000. Singleton consistencies, *Principles and Practice of Constraint Programming, CP'2000, Singapore*, Springer, LNCS 1894, pp. 353–368.

Puget, J. F. 1994. A C++ implementation of CLP, *Technical report*, ILOG S. A. http://www.ilog.com.

Puget, J.-F. 1998. A fast algorithm for the bound consistency of alldiff constraints, *Proc. 15th National Conference on Artificial Intelligence (AAAI'98) and 10th Conference on Innovative Applications of Artificial Intelligence (IAAI'98)*, AAAI Press, pp. 359–366.

Raman, R. & Grossmann, I. 1991. Symbolic integration of logic in mixed-integer linear programming techniques for process synthesis, *Computers and Chemical Engineering* **17**: 909–927.

Raman, R. & Grossmann, I. 1993. Relation between MILP modeling and logical inference for chemical process synthesis, *Computers and Chemical Engineering* **15**: 73–84.

Raman, R. & Grossmann, I. 1994. Modeling and computational techniques for logic based integer programming, *Computers and Chemical Engineering* **18**: 563–578.

Refalo, P. 1999. Tight cooperation and its application in piecewise linear optimization, *Principles and Practice of Constraint Programming, CP'99, Alexandria, VA*, Springer, LNCS 1713, pp. 375–389.

Refalo, P. 2000. Linear formulation of constraint programming models and hybrid solvers, *Principles and Practice of Constraint Programming, CP'2000,*

*Singapore*, Springer, LNCS 1894, pp. 369 – 383.

Régin, J.-C. 1994. A filtering algorithm for constraints of difference in CSPs, *Proc. 12th National Conference on Artificial Intelligence, AAAI'94, Seattle*, Vol. 1, pp. 362 – 367.

Régin, J.-C. 1996. Generalized arc consistency for global cardinality constraint, *Proc. 13th National Conference on Artificial Intelligence, AAAI'96, Protland*, Vol. 1, pp. 209 – 215.

Régin, J.-C. 1999a. Arc consistency for global cardinality constraints with costs, *Principles and Practice of Constraint Programming, CP'99, Alexandria, VA*, Springer, LNCS 1713, pp. 390–404.

Régin, J.-C. 1999b. The symmetric alldiff constraint, *Proc. 16th International Joint Conference on Artificial Intelligence, IJCAI'99, San Francisco*, Vol. 1, pp. 420–425.

Régin, J.-C. & Puget, J.-F. 1997. A filtering algorithm for global cardinality constraints with costs, *Principles and Practice of Constraint Programming, CP'97, Linz, Austria*, Springer, LNCS 1330, pp. 32–46.

Régin, J.-C. & Rueher, M. 2000. A global constraint combining a sum constraint and difference constraint, *Principles and Practice of Constraint Programming, CP'2000, Singapore*, Springer, LNCS 1894, pp. 384–395.

Rodošek, R. & Wallace, M. 1998. A generic model and hybrid algorithm for hoist scheduling problems, *Principles and Practice of Constraint Programming (CP98)*, Vol. 1520 of *Lecture Notes in Computer Science*, Springer, pp. 385–399.

Rodošek, R., Wallace, M. & Hajian, M. 1997. A new approach to integrating mixed integer programming and constraint logic programming, *Annals of Operations Research* **86**: 63–87.

Sabin, D. & Freuder, E. C. 1994. Contradicting conventional wisdom in constraint satisfaction, *Principles and Practice of Constraint Programming, PPCP'94, Rosario*, Springer, LNCS 874, pp. 10–20.

Sabin, D. & Freuder, E. C. 1997. Understanding and improving the MAC algorithm, *Principles and Practice of Constraint Programming, CP'97, Linz, Austria*, Springer, LNCS 1330, pp. 167–181.

Sakkout, L. E., Richards, T. & Wallace, M. 1998. Minimal perturbance in dynamic scheduling, *in* H. Prade (ed.), *Proceedings, 13th European Conference on Artificial Intelligence*, Vol. 48, Wiley, New York, pp. 504–508.

Saraswat, V. A. 1993. *Concurrent constraint programming*, ACM Doctoral Dissertation Awards, MIT Press.

Sellmann, M. & Fahle, T. 2001. CP-based lagrangian relaxation for a multimedia application, *Third International Workshop on the Integration of AI and OR Techniques (CPAIOR 2001)*.

Smith, B. M., Brailsford, S. C., Hubbard, P. M. & Williams, H. P. 1996. The progressive party problem: Integer linear programming and constraint programming compared, *Constraints* **1**: 119–138.

Smolka, G. 1995. The Oz programming model, *in* J. van Leeuwen (ed.), *Computer Science Today: Recent Trends and Developments*, Springer, LNCS 1000.

Stergiou, K. & Walsh, T. 1999a. The difference all-difference makes, *16th Intern. Joint Conf. Artificial Intelligence, IJCAI'99, Stockholm*, pp. 414–419.

Stergiou, K. & Walsh, T. 1999b. Encodings of non-binary constraint satisfaction problems, *Proc. 16th National Conference on Artificial Intelligence (AAAI'99) and 11th Conference on Innovative Applications of Artificial Intelligence (IAAI'99)*, pp. 163–168.

Thorsteinsson, E. S. 2001. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming, *Seventh International Conference on Principles and Practice of Constraint Programming (CP2001)*.

Timpe, C. 2003. Solving BASF's plastics production planning and lot-sizing problem using combined CP/MIP, LISCOS Project Summary Meeting, Brussels (28 March 2003).

Türkay, M. & Grossmann, I. E. 1996. Logic-based MINLP algorithms for the optimal synthesis of process networks, *Computers and Chemical Engineering* **20**: 959–978.

van Hentenryck, P. 1989. *Constraint satisfaction in logic programming*, MIT Press.

van Hentenryck, P. 1999. *The OPL Optimization Programming Language*, MIT Press. (with contributions by I. Lustig, L. Michel, J.-F. Puget).

van Hentenryck, P. & Graf, T. 1992. A generic arc consistency algorithm and its specializations, *Artificial Intelligence* **57**: 291–321.

van Hentenryck, P., Michel, L. & Benhamou, F. 1998. Newton: constraint programming over non-linear constraints, *Science of programming* **30**: 83–118.

van Hentenryck, P., Michel, L., Perron, L. & Régin, J.-C. 1999. Constraint programming in OPL, *Principles and Practice of Declarative Programming, International Conference PPDP'99, Paris*, Springer, LNCS 1702, pp. 98–116.

van Hentenryck, P., Saraswat, V. & Deville, Y. 1998. Design, implementation, and evaluation of the constraint language cc(FD), *Journal of Logic Programming* **37**(1-3): 139–164.

van Hoeve, W. J. 2001. The alldifferent constraint: a survey, *Sixth Annual Workshop of the ERCIM Working Group on Constraints, Prague.*

http://arXiv.org/abs/cs.PL/0105015.

Wallace, M., Novello, S. & Schimpf, J. 1997. ECLiPSe: A platform for constraint logic programming, *ICL Systems Journal* **12**: 159–200.

Walsh, T. 1997. Depth-bounded discrepancy search, *15th Intern. Joint Conf. Artificial Intelligence, IJCAI'97, Nagoya, Japan*, Vol. 2, pp. 1388–1395.

Williams, H. P. & Yan, H. 2001. Representations of the all-different predicate of constraint satisfaction in integer programming, *INFORMS Journal on Computing* **13**: 96–103.

Woeginger, G. J. 2001. The reconstruction of polyominoes from their orthogonal projections, *Information Processing Letters* **77**(5-6): 139–164.

Yan, H. & Hooker, J. N. 1999. Tight representation of logical constraints as cardinality rules, *Mathematical Programming* **85**: 363–377.

Zhang, Y. & Yap, R. H. C. 2000. Arc consistency on $n$-ary monotonic and linear constraints, *Principles and Practice of Constraint Programming, CP'2000, Singapore*, Springer, LNCS 1894, pp. 470–483.

Zhang, Y. & Yap, R. H. C. 2001. Making AC-3 an optimal algorithm, *17th Intern. Joint Conf. Artificial Intelligence, IJCAI'01, Seattle*, Vol. 1, pp. 316–321.

Zhou, J. 1997. *Computing smallest cartesian products of intervals: application to the job-shop scheduling problem*, PhD thesis, Univ. de la Méditerranée Aix-Marseille II.

Zhou, J. 2000. Introduction to the constraint language NCL, *Journal of Logic Programming* **45**(1-3): 71–103.