Ann Oper Res (2010) 175: 37–76
DOI 10.1007/s10479-009-0654-9

Integrating Operations Research in Constraint Programming

Michela Milano · Mark Wallace

Published online: 27 October 2009

Abstract This paper presents Constraint Programming as a natural formalism for modelling problems, and as a flexible platform for solving them. CP has a range of techniques for handling constraints including several forms of propagation and tailored algorithms for global constraints. It also allows linear programming to be combined with propagation and novel and varied search techniques which can be easily expressed in CP. The paper describes how CP can be used to exploit linear programming within different kinds of hybrid algorithm. In particular it can enhance techniques such as Lagrangian relaxation, Benders decomposition and column generation.

Keywords Constraint Programming · Mathematical Programming · Problem modelling and solving

1 Introduction

Although Operations Research (OR) and Constraint Programming (CP) have different roots, the links between the two communities have grown stronger in recent years. For solving combinatorial optimization problems the techniques of CP and OR will become so interdependent that the two research communities could eventually merge.

Of course, OR being a much larger community, with a much longer history, it seems a bit cheeky to entitle this paper "Integrating OR in CP". Nevertheless the title does reflect the different research objectives of CP and OR. OR primarily studies models and algorithms, aiming to develop more powerful, flexible and especially more scalable solutions to combinatorial optimization problems. A *solution*, in this sense, is ultimately an algorithm to solve

This is an updated version of the paper that appeared in *4OR*, *4*(3), 175–219 (2005).

M. Milano (✉)
DEIS University of Bologna, Via Risorgimento, 2, 40136 Bologna, Italy
e-mail: michela.milano@unibo.it

M. Wallace
Faculty of Information Technology, Monash University, Clayton, Vic 3800, Australia
e-mail: Mark.Wallace@infotech.monash.edu.au

a problem. By contrast CP addresses issues *about* algorithms rather than the algorithms themselves. Such issues include when to invoke an algorithm, how to integrate algorithms into a system, and what their properties are. CP researchers are content to "steal" algorithms from other communities—the research comes from using and combining them in new ways, and widening the use of sophisticated algorithms, by supporting the mapping of problem definitions to appropriate combinations of algorithms.

In this survey we will seek to make clear what it is that CP brings to problem solving, and thus how it can bring benefits to OR researchers. The survey is a slightly extended and updated version of Milano and Wallace (2005).

We explore CP for problem modelling: in particular we distinguish user-oriented conceptual models from solver-oriented design models. We discuss the CP facilities for defining constraints, and study the data types and structures that support precise but clear modelling. We illustrate CP modelling in Sect. 3 with a one-machine scheduling example.

In Sect. 4 we restrict our attention to CP on finite domains and explore the concept of propagation, and the fix-point principle underlying it. We introduce some global constraints and algorithms for handling them. We finish the section by studying global constraints with costs.

Section 5 presents the integration of linear constraint solving into CP. We start with a simple linear problem, and then cover row generation, and logical combinations of constraints, exploiting the combination of constraint propagation and linear programming. The section ends with a description of Lagrangian relaxation, and an example of its use in a CP/LP hybrid.

Section 6 discusses search: search control is an important aspect of problem solving which has received more attention from CP researchers than OR. We describe some forms of heuristically-guided incomplete tree search, and examine some CP heuristics. We discuss variable and value ordering, and heuristics coming from the hybridization of CP and LP. We then explore search and optimization, illustrating it with the one-machine scheduling example introduced earlier. The section concludes with an examination of some decomposition techniques—in particular Benders decomposition.

Section 7 describes various local search techniques which work by improving on non-optimal solutions by making local changes. We introduce the concept of an invariant—key to implementing local moves efficiently. We distinguish driving and driven variables, and discuss feasibility-preserving moves. We look at pivoting as a local move operator, and discuss an example of its use within a hybrid search algorithm. The second half of Sect. 7, presents Column generation as another way to improve on non-optimal solutions. We discuss the contribution of CP to subproblem solving and vice versa the impact of branch-and-price on traditional constraint programming search methods.

In conclusion we discuss the software-engineering role of CP, for model encapsulation and reuse. We conclude with a description of open perspectives and challenging research directions.

2 Modelling

2.1 Conceptual and design models

The holy grail of CP researchers is to build a system which takes *any* precise high-level model of a problem and automatically maps the model to efficient solution methods. For example any precise statement of the TSP would be automatically mapped to the highly sophisticated algorithm of Applegate et al. (1999).

The authors of this survey would agree with most OR researchers that such an automated mapping is currently a pipe-dream, and researching into ways of automating the mapping would be a waste of effort.

However it is a realistic and important research topic to explore the different ways problems, and their subproblems, can be mapped to algorithms. At its simplest we could imagine a parameter setting that maps a problem either to a more-or-less naive integer-linear model and algorithm, or to a meta-heuristic solution, or to a CP propagation and search solution. This is too simple, of course. We seek a much more sophisticated sequence of steps under the control of a (human) problem solver that can start out with a problem statement and end up with an efficient algorithm. Our more restricted objective is to describe each of these steps in detail, so that a less expert problem solver can follow the same path, and even an expert can quickly try multiple paths from problem statement to algorithm in order to find the most efficient one.

The assumption behind this agenda is that even a "bad" model for a problem can be transformed into the "right" one by a sequence of steps.

Consequently we distinguish a high level *conceptual* problem model which is completely independent of the algorithm which best solves it. The *design* model is a model that maps down to an algorithm in a way that is well-defined and fixed. An integer-linear model is in this sense a *design* model, assuming that the particular algorithm (e.g. dual simplex) has been specified in advance.

The CP research agenda is:

- to build a rich and high-level formalism for specifying design models, that incorporates techniques from OR, AI, and the metaheuristics community, graph algorithms and anything algorithmic that contributes to combinatorial optimization;
- to build a rich and natural formalism for specifying conceptual models and flexible methods for mapping conceptual models to design models;
- to build an understanding of what properties of problems influence their efficient solution, so as to guide the mapping of conceptual to design models.

In this survey we shall describe high level CP design models. Whilst these models are motivated by the need for a simple conceptual-design mapping, we shall not discuss conceptual models further in this survey.

2.2 CP constraints

CP built-in constraints. The main advantage of CP models over mathematical modelling is the range of constraints that can be directly expressed.

First in finite domain solvers, all variables range on a finite domain of objects of arbitrary type, though usually integer. Domains are expressed either as a list of objects or as a range. As an example, $X :: [1, 3, 8]$ constrains variable X to assume either the value 1 or 3 or 8, while $X :: [1..10]$ constrains variable X to assume an integer value between 1 and 10. CP systems support all the usual mathematical constraints $LHS \geq RHS$, $LHS \leq RHS$, $LHS = RHS$, but with a much wider range of admissible numerical expressions LHS and RHS. Besides linear expressions, CP can admit polynomials, trigonometric functions, max, min, abs and many other functions. There is, as yet, no standard set of functions for all the different CP systems, but the expressive power comes from the propagation algorithms used to handle these constraints. Additional numeric constraints handled by CP include

$LHS > RHS$, $LHS < RHS$, $LHS \neq RHS$. These constraints are most useful when the expressions are integer.

In addition to numeric variables and constraints, many CP systems handle symbolic variables. As a simple example, consider two variables Emp_1 and Emp_2 ranging on a domain of employees, say $Emp_1 :: [mark, john]$ and $Emp_2 :: [alan, mark, john]$. Suppose now, that Emp_1 performs task t_1 and Emp_2 performs task t_2 that are temporally overlapped. Clearly, t_1 and t_2 cannot be performed by the same employee. We can impose $Emp_1 \neq Emp_2$ ensuring that these two variables would never be assigned to the same value. Note that we can also impose equality on symbolic variables, using the built-in constraint $=$.

Many problems involve logical combinations of constraints. For example we might require that amongst all the employees assigned to carry out task t, each skill required for task t must be covered by at least one employee.

In CP this expressive power is achieved by the facility to associate with any constraint a boolean variable that is set to 1 if the constraint holds and to 0 otherwise. This is termed a *reified* constraint. In the following we represent an arbitrary constraint on a list of variables $\overline{X}$ as $Cons(\overline{X})$ and its reified version as $Cons(\overline{X}, B)$. CP languages either allow logical combinations to be stated explicitly

$$Cons_1(\overline{X_1}) \vee Cons_2(\overline{X_2}) \vee \cdots \vee Cons_n(\overline{X_n})$$

or they require the user to exploit the boolean variables in the usual way

$$Cons_1(\overline{X}_1, B_1) \wedge Cons_2(\overline{X}_2, B_2) \wedge \cdots \wedge Cons_n(\overline{X}_n, B_n) \wedge \left(\sum_{i=1}^{n} B_i \geq 1\right)$$

Notice that reification requires the underlying system to handle both the constraint (since the constraint $Cons(\overline{X})$ is enforced by setting the boolean to 1 in $Cons(\overline{X}, 1)$), and also its negation (which is enforced by $Cons(\overline{X}, 0)$).

CP user-defined constraints. CP also enables the user to define new constraints. CP offers many different ways of enabling users to define application-specific or even problem-specific constraints.

One way in which the user can define specific constraints is the explicit list of all tuples that satisfy the constraint.

A simple example is the constraint between employees and tasks, that only allows employees to perform the tasks for which they have the necessary skills. This constraint can be expressed either explicitly listing the tasks for which each employee is qualified, or implicitly using data about employee skills and skills needed to carry out each task.

A constraint linking employees to the tasks they can carry out, and the time it takes each employee to carry out each of these tasks can be defined straightforwardly in CP by a table of values, e.g.

```
qual(emp_1, t_1, 3).
qual(emp_3, t_1, 4).
qual(emp_1, t_2, 4).
...
```

Suppose *Emp* is a variable denoting the employee who will carry out a given task *t_1*, and *Dur* is the time needed for the employee to carry out this task. Then CP allows the user to invoke the constraint

$$qual(Emp, t, Dur)$$

This constraint is satisfied by assigning to *Emp* and *Dur* values (say *emp_1* and *3*) so that the resulting tuple *qual(emp_1, t_1, 3)* belongs to the table.

More generally CP provides an ideal language for specifying application-specific constraints, because it is a declarative language based on relations. Constraints are also, mathematically, relations. For example the constraint relating the lengths of the sides of a right-angle triangle can be written directly:

```
triangle(SideA, SideB, Hyp) :-
    Hyp > SideA,
    Hyp > SideB,
    Hyp^2 =:= SideA^2 + SideB^2.
```

CP also provides facilities for handling such user-defined constraints, introduced in Sect. 4.2 below.

Global constraints. Of particular importance are *global* constraints representing complex relations between multiple variables. The number of variables need not be fixed: some global constraints can handle lists of variables of any length. These are specific constraints that may represent interesting subproblems and that are encoded into the CP system. For global constraints efficient and powerful filtering algorithms have been written. The most well-known such constraint is the *alldifferent* constraint (Régin 1994). Suppose we want to constrain a list of domain variables to assume different values, then we simply state

$$\mathit{alldifferent}([X_1, \ldots, X_n])$$

This constraint holds iff $X_1 \neq X_2, X_1 \neq X_3, \ldots X_{n-1} \neq X_n$.

We will see in Sect. 4.3 these two formulations will not be equivalent from a propagation perspective.

An extension of the *alldifferent* constraint is the *global cardinality constraint*, hereinafter referred to as *gcc*. Its parameters are: a set of variables $[X_1, \ldots, X_n]$ ranging on domains $[D_1, \ldots, D_n]$, two functions l and u associating two integer values $l(v)$ and $u(v)$ for each $v \in \bigcup_i D_i$

$$\mathit{gcc}([X_1, \ldots, X_n], [D_1, \ldots, D_n], l, u)$$

holds iff the number of times $v \in \bigcup_i D_i$ appears in $[X_1, \ldots, X_n]$ is between $l(v)$ and $u(v)$.

Other global constraints have been introduced to handle scheduling problems (Beldiceanu and Carlsson 2002). For example let us consider the cumulative constraint used for modelling limited resource availability in scheduling problems. Its parameters are: a list of variables $[S_1, \ldots, S_n]$ representing the starting time of all activities sharing the resource, their duration $[D_1, \ldots, D_n]$, the resource consumption for each activity $[R_1, \ldots, R_n]$ and the available resource capacity C. Clearly this constraint holds if in any time step where at least one activity is running the sum of the required resource is less than or equal to the available capacity:

$$\mathit{cumulative}([S_1, \ldots, S_n], [D_1, \ldots, D_n], [R_1, \ldots, R_n], C)$$

holds iff

$$\forall i \sum_{j: S_j \le i < S_j + D_j} R_j \le C$$

An exhaustive list of global constraints implemented in commercial CP solvers can be found in Regin (2004).

2.3 Data types and structures

Mathematical models are powerful enough to describe all kinds of problems. These models can be difficult to maintain and inefficient to evaluate. By encoding any symbol as an integer we can represent symbolic variables numerically. However the resulting model can be littered with integers whose interpretation is difficult or impossible to ascertain for anyone except the original developer of the model. More subtly, integer-linear programming does not always work efficiently on problems with integer variables which represent symbolic values. The confusion of integers which really represent integers and those which represent symbols makes it harder to develop the best combination of algorithms for solving such problems efficiently.

Perhaps the first mathematical modelling language to support data structures is OPL (Van Hentenryck 1999). Indeed OPL can be viewed as a CP modelling language.

Data structures have an important role from the point of view of software engineering. They make models easier to understand, and easier to change.

In a traditional mathematical modelling language the link between a task start time variable and its duration would be implicit in the model. Consequently when a task duration was changed, it would require someone familiar with the model to determine which values in the model needed to be changed.

In a model with data structures, constraints can explicitly refer to the task itself. For example to say that the task must end before its due date we can write: $task_i.start + task_i.duration \leq task_i.duedate$. In this model the due date can be changed unambiguously by one change in the data structure that represents the task.

Data structures offer more than a better way to write design models. The introduction of the *list* data structure makes it possible to build more general constraints and to write models that cannot be expressed without them.

– **More general global constraints.** The *alldifferent* constraint introduced above is a constraint that can be applied to a list of variables *of any length*. This means we can use the same constraint for problems with any number of variables. Similarly the global scheduling constraint can be used for problems with any number of tasks.
– **More expressive design models.** Lists also enable us to express problems with an initially unspecified number of decision variables. For example in Artificial Intelligence planning problems we have to find a set of actions characterized by preconditions and effects that starting from an initial state of the world, change it to achieve a goal. Clearly it is not know in advance how many actions will be required in any solution. Such problems cannot be expressed without lists, because without lists only answers with a given number of output values can be returned.

3 One-machine scheduling: the model

In this section we shall present a simple CP solution for a class of single machine scheduling problems. This example is intended to demonstrate the brevity of CP; its power to solve scheduling problems, without artificially discretising time and imposing constraints on each timepoint. In Sect. 4.3.1 we show an example of propagation for the resource constraint, while in Sect. 6.2 we describe the search control available to the modeller.

The example CP programs throughout this paper are expressed in the syntax of the ECLiPSe language (Apt and Wallace 2006).

The problem is expressed in terms of a list of tasks. Each task is a structure containing its release date, duration, due date and resource requirement. The objective is to minimize lateness. The cost of a late task increases quadratically with its lateness.

This program uses the expressive power of the global *cumulative* constraint, introduced in Sect. 2.2.

We first define the required data structure where each task has the required attributes, and then define the scheduling procedure.

```
:- local struct(task(release,duration,due,resource)).
```

This declares a `task` to be a record structure, with fields `release` (the release date of the task), `duration`, `due` (its due date) and `resource` (how much machine resource is used by the task—its value in this example is always 1 because one machine is (completely) used to run each task).

The main predicate which returns a feasible schedule is called `sched1`. It calls `setup` and `single_cumulative` to set up the problem and constraints. These predicates are defined below. It then constrains the optimization variable `OptVar` to be equal to the sum of the costs associated with each task. Finally it sorts the list of task start time variables `Starts` into the best order for the branch-and-bound search. The list `KeyStarts` is a list of `Key-Start` pairs, where the key `Key` is used by the sort routine. The `minimize` routine explores a depth-first branch-and-bound search tree to solve the problem. We will focus on search in Sect. 6.2.

```
sched1(Tasks, Starts, OptVar) :-
   setup(Tasks,Starts,KeyStarts,Costs),
   single_cumulative(Starts,Tasks),
   OptVar $= sum(Costs),
   sort(KeyStarts,SortedStarts),
   minimize(search(SortedStarts),OptVar).
```

We now describe the predicate `setup`. It initializes the start time variables `Starts` and their associated cost expressions `Costs`. Instead of iterating over an index, as in a traditional MIP model, the CP program iterates implicitly using the `foreach ... do ...` construct.

For each task `Task` in the list of tasks `Tasks`, a new variable `Start` is introduced and constrained to be greater than the task's release date `Rel`. The `foreach ... do ...` syntax is used not only for iterating over a list, but also for constructing a new list. Using this syntax, the task start time variables are accumulated in a new list called `Starts`.

A key `Key` is computed for the task: this is simply its due date `Due` minus its duration `Dur`. From the keys and start time variable, a list of Key-Start pairs are created and accumulated in the list `KeyStarts`. Finally a cost for each task is computed by the predicate `ic_cost_cons`. The costs are accumulated in another list called, appropriately, `Costs`.

```
setup(Tasks, Starts, KeyStarts, Costs) :-
   (
     foreach(Task, Tasks),
     foreach(Start,Starts),
     foreach(Key-Start,KeyStarts),
     foreach(Cost,Costs)
   do
      Task = task{release:Rel,duration:Dur,due:Due},
      Start $>= Rel,
      Key is Due - Dur,
      ic_cost_cons(Start, Dur, Due, Cost)
   ).

ic_cost_cons(Start, Dur, Due, Cost) :-
      SimpCost $= (Start + Dur - Due),
      '$>='(SimpCost,0,B),
      Cost $= B*SimpCost^2.
```

In the definition of the predicate `ic_cost_cons`, note that the syntactic expression (`'$>='(Expr1, Expr2, Bool)`) is a reified constraint that holds if either $Expr1 \geq Expr2$ and $Bool = 1$ or $Expr1 < Expr2$ and $Bool = 0$. We introduced reified constraints in Sect. 2.2. The cost of a task is the square of its lateness, or 0 if it is not late.

There is no upper bound on the start time. Instead late starting is penalized by the cost expression. The heuristic `Key`, which will be used in the search routine, will be described in Sect. 6.2.

The other predicate `single_cumulative` imposes a simplified version of the `cumulative` constraint for one-machine scheduling.

```
single_cumulative(Starts,Tasks) :-
   ( foreach(Task, Tasks),
     foreach(Dur, Durs),
     foreach(Resource, Resources)
   do
      Task = task{duration:Dur,resource:Resource}
   ),
   cumulative(Starts,Durs,Resources,1).
```

The total resource available is just 1, because this is a single-machine scheduling problem. The cumulative constraint enforces that only one task can run on the machine at a time.

4 Propagation

4.1 Standard propagation and fix-pointing

One of the main ingredients of the Constraint Programming solving process is constraint propagation. This process is aimed at removing those values that are provably inconsistent with the problem constraints and therefore cannot lead to any consistent solution. If, after propagation, a variable domain becomes empty, a failure occurs. Every constraint has an embedded filtering algorithm which works locally and propagates tighter bounds on its variables, until no more propagation can be inferred from the constraints. The state when the constraint cannot propagate further is termed a fix-point. For example the constraint `X>Y` reaches a fix-point when the lower bound of X becomes greater than the lower bound of Y, and the upper bound of X becomes greater than the upper bound of Y, e.g. $X :: [3..10]$ and $Y :: [2..5]$. Notice, however, that there remain pairs of values for X and Y, such as $X = 3$ and $Y = 5$, that do not satisfy the constraint.

When the fix-point is reached the constraint is suspended until either the program finishes (and the constraint is provided as conditional answer) or a triggering event happens on the involved variables. Triggering events are mainly three: the instantiation of a variable, the reduction of a domain bound, the removal of a domain value.

Unlike an integer-linear solver, CP propagation handles each constraint independently from the others. It uses the mathematical constraint to reason about the upper and lower bounds of the variables occurring in it, and each time a bound is changed either by propagation on another constraint, or during search, the constraint is woken and propagates again. One consequence is that integer constraints can be enforced very easily by simply rounding up the lower bound and rounding down the upper bound.

One important aspect to consider is the fix-point of the filtering algorithm. For this purpose, some properties have been proposed to define the notion of fix-point. These properties are local to the constraint. If the constraint is binary we have the notion of arc consistency, while if the constraint is n-ary then we should extend this concept to the notion of Hyper-arc Consistency or Generalized Arc Consistency GAC.

Arc consistency: A binary constraint C on the variables x_1 and x_2 whose domains are D_1 and D_2 is arc consistent if and only if for all values $v_1 \in D_1$ (resp. $v_2 \in D_2$) there exists a value $v_2 \in D_2$ (resp. $v_1 \in D_1$) such that $(v_1, v_2) \in C$.

A problem is arc consistent if and only if all its constraints are. As an example, let us suppose that we have three variables X, Y, Z ranging on the domain $[1..10]$, and the constraint $X < Y$ which produces the propagation: $X :: [1..9]$ and $Y :: [2..10]$ and the constraint $Y < Z$ which produces the propagation: $Y :: [2..9]$ and $Z :: [3..10]$.

The fact that D_Y has changed produces the final propagation of constraint which leads to the reduction of the domains of X, Y, Z to:

$$X :: [1..8], \qquad Y :: [2..9] \qquad Z :: [3..10]$$

Note that if a domain becomes empty, it means that the problem has no solution, but if domains are not empty it does not mean that the problem has a solution. In fact, arc consistency is local to a binary constraint. When more constraints are present in the problem being each of them arc consistent does not ensure that the problem has a solution. Here is an example: suppose we have three variables $X :: [1..2]$, $Y :: [1..2]$, $Z :: [1..2]$ subject to $X \neq Y$, $Y \neq Z$ and $X \neq Z$. The problem constraints are clearly inconsistent. However. for each value there is a support in the domain of other variables, therefore the problem is arc consistent.

To overcome this problem, other degrees of consistency have been defined.

Hyper-arc consistency or **Generalized Arc consistency**. A constraint C on the variables $x_1, \ldots, x_m$ with respective domains $D_1, \ldots, D_m$ is hyper-arc consistent if and only if for each variable x_i and each value $v_i \in D_i$, there exists a value $v_j \in D_j$ for all $j \neq i$ such that $(v_1, \ldots, v_m) \in C$.

The example above shows that the constraint *alldifferent*$([X, Y, Z])$ where $X :: [1..2]$, $Y :: [1..2]$, $Z :: [1..2]$ is not generalized arc consistent.

Note that arc consistency is a specialization of the hyper-arc consistency when applied to binary constraints. Both arc consistency and hyper-arc consistency check whether all values in every domain belong to a tuple that satisfies the constraint, with respect to the current variable domains.

In some applications, it is not needed to check all values in the domain, but only bounds are considered. For a finite, ordered domain D_i, we define $\min_{D_i}$ and $\max_{D_i}$ to be their minimum and maximum values.

Bound consistency. A constraint C on the variables $x_1, \dots, x_m$ with respective domains $D_1, \dots, D_m$ is bound consistent if for each variable x_i and each value $v_i \in \{\min_{D_i}, \max_{D_i}\}$ there exists a value $v_j \in \min_{D_j} .. \max_{D_j}$ for all $j \neq i$ such that $(v_1, \dots, v_m) \in C$.

A CSP is bound consistent or hyper-arc consistent if and only if all its constraints are.

Clearly how to achieve these properties is very important as far as efficiency is concerned. There are two general ways of achieving these properties: with general purpose methods, i.e., methods that are not linked to the semantics of the constraint and with specialized methods, exploiting the semantics of the constraint.

General purpose approaches for achieving hyper-arc consistency have been studied in Bèssiere and Régin (1997), Freuder and Régin (1999). The interesting aspect of these algorithms is their generality and problem-independency. The drawback is that their efficiency is indeed quite poor for large problems and they are rarely used in practice. Bound consistency is, by contrast, widely used (Choi et al. 2004; Davis 1987; Benhamou et al. 1999; Schulte and Stuckey 2005).

4.2 User-defined constraints

Constraints can be defined by a table, such as the example given earlier:

```
qual(emp_1, t_1, 3).
qual(emp_3, t_1, 4).
qual(emp_1, t_2, 4).
...
```

If the constraint is imposed on three variables,

$$qual(Emp, Task, Dur)$$

it can be handled using general purpose approaches for handling hyper-arc consistency, as described in the previous paragraph. Alternatively it could be delayed until one or more of the variables becomes instantiated, perhaps as a result of propagation on other constraints. At this point the constraint has (at most) two variables, and can be handled by maintaining arc consistency as described on page 45.

For the most sophisticated user, new constraints can be written in any programming language and linked to the CP system by an interface. The interface requires the user to specify

- Failure or success reflecting whether the constraint is unsatisfied in the current state, or not.
- Elimination or re-suspension, depending on whether the constraint is entailed in the current state or not.
- New constraints inferred (propagated) by the user-defined constraint. These constraints are usually in the form of tightened bounds or reduced domains for the variables, but they can be any constraint defined elsewhere in the CP system.
- Waking conditions—events in the CP system which will cause this constraint to be evaluated or reevaluated.

Some CP systems support facilities which make it easier for the user to define constraints. These include indexicals (Codognet and Diaz 1996) which allow bound and domain reduction behaviour to be specified easily, Constraint Handling Rules (CHR's) (Fruhwirth 1998) which enable the user to describe constraint rewriting rules, generalised propagation (Le Provost and Wallace 1993) which allow constraints to be written simply as logic programs, and action rules (Zhou 2005) which give the user access to an efficient implementation of waking conditions.

4.3 Global constraints without costs

Feasibility-based pruning One of the most important aspects of Constraint Programming languages is represented by global constraints. Global constraints, also called symbolic constraints, are n-ary constraints that represent suitable abstractions embedding very sophisticated, semantic-aware filtering algorithms.

Some constraints represent a polynomial problem. In this case, hyper-arc consistency can be achieved in polynomial time with a special purpose algorithm. For example, one of the most widely used and successful global constraints is the *alldifferent* constraint: it is defined on a set of variables $[X_1, \dots, X_n]$ ranging on domains $[D_1, \dots, D_n]$ and it holds iff all variables are assigned to different values. This constraint has the structure of an assignment problem (with no objective function). This constraint has a corresponding Integer Linear Programming model: we have x_{ij} decision variables that take the value 1 if the CP variable X_i is assigned to value j, 0 otherwise.

$$\sum_i x_{ij} = 1 \quad \forall j,$$

$$\sum_j x_{ij} \leq 1 \quad \forall i,$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j$$

Despite the integrality constraint is part of this model, the coefficient matrix is totally unimodular meaning that linear programming provides an integer solution. Clearly, the addition of side constraints, or multiple *alldifferent* constraints, would break this structure. However, the *alldifferent* constraint is treated locally in Constraint Programming, thus exploiting the unimodularity property leading to a polynomial propagation algorithm.

If, instead the constraint represents an NP-complete problem, then hyper-arc consistency cannot be achieved in polynomial time. Therefore, one aims at reaching approximation of hyper arc consistency.

Many global constraints exploit graph-theory results and algorithms for pruning inconsistent values. The filtering algorithm for the *alldifferent* constraint achieves hyper arc consistency by applying a maximum matching algorithm on a bipartite graph $G = (X \cup V, E)$, called *value graph* (Laurière 1978) built as follows: X is a set of nodes representing variables involved in the constraint, V is a set of nodes representing values contained in the union of the domains, and E represents the set of edges, $(X_i, v) \in E$ iff $v \in D_i$. Hyper arc consistency of an *alldifferent* constraint is established by computing a maximum matching M which covers X and identifying all edges that belong to a maximum matching. The complexity of computing a maximum matching (or prove there is none) is $O(m\sqrt{n})$ where m is the number of edges and n the number of variables. The complexity of finding all edges belonging to a maximum matching and thus also achieving hyper arc consistency can be

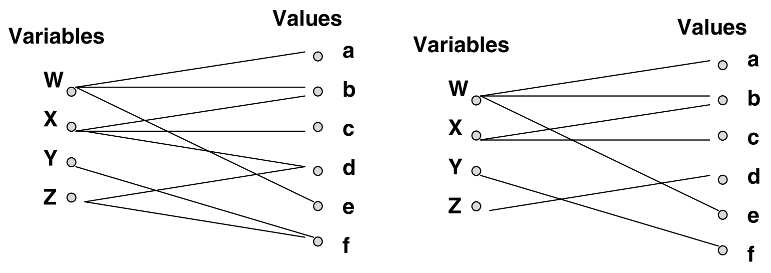

Fig. 1 A bipartite graph for the *alldifferent* constraint

done in $O(m)$ (van Hoeve 2006). For example the following set of variables and domains subject to an *alldifferent* constraint:

```
W :: [a,b,e]
X :: [b,c,d]
Y :: [f]
Z :: [d,f]
```

is represented in the left-hand side of Fig. 1.

The filtering algorithm embedded in the *alldifferent* that achieves hyper arc consistency produces the following propagation:

```
W :: [a,b,e]
X :: [b,c]
Y :: [f]
Z :: [d]
```

as shown in the right-hand side of Fig. 1.

In this specific case, the same propagation could be reached also if we propagate up to arc consistency the corresponding set of binary constraints. The advantage of the global constraint here is none. This is not always the case. Consider the following example:

```
W :: [a,b,f]
X :: [a,b,d]
Y :: [f]
Z :: [d,f]
K :: [a,b,e]
```

the filtering algorithm based on the maximum matching produces the following propagation:

```
W :: [a,b]
X :: [a,b]
Y :: [f]
Z :: [d]
K :: [e]
```

while using the binary constraint propagation, propagation on variable `K :: [a,b,e]` yields no domain reduction.

As another example, a second constraint exploiting graph theory algorithms for achieving hyper-arc consistency is a generalization of the *alldifferent* constraint, the global cardinality constraint *gcc*, introduced above on page 41.

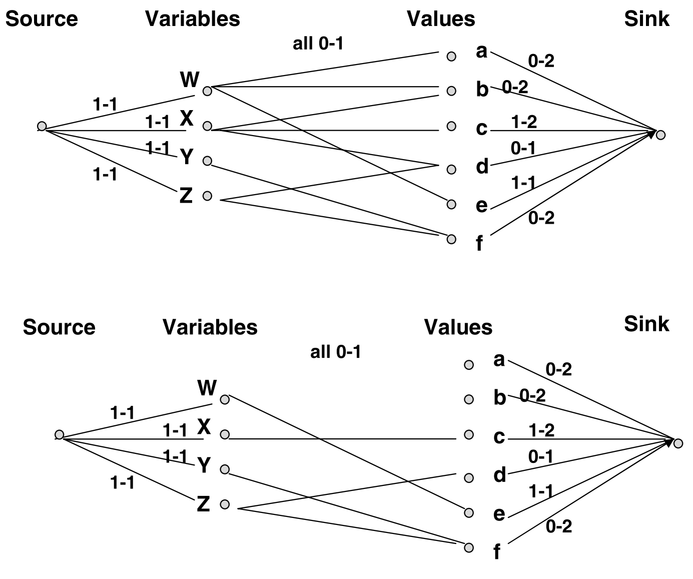

Fig. 2 A flow graph for the *gcc* constraint

We can adapt one of the previous examples to represent the constraint

```
gcc([W,X,Y,Z],
    [[a,b,e],[b,c,d],[f],[d,f]],
    [a-[0,2],b-[0,2],c-[1-2],d-[0,1],e-[1,1],f-[0,2]])
```

Again, graph theory is used for achieving hyper arc consistency and for pruning all infeasible values. Let us start with the value graph (bipartite graph) presented for the *alldifferent* constraint. We have two sets of nodes: variable nodes and value nodes. Here we add a source node s and a sink node t, as shown in the upper part of Fig. 2. In addition arcs are associated with a demand and a capacity and are divided in three sets:

- those starting from s and ending in a variable node. Their demand is 1 and their capacity is 1.
- those connecting a variable and a value node (that are the same for the alldifferent constraint). Their demand is 0 and their capacity is 1.
- those starting from a value node and ending in t. Their demand is $l(v)$ and their capacity is $u(v)$.

A solution of the above mentioned *gcc* corresponds to a feasible $s-t$ flow in the above mentioned graph, which can be found in $O(nm)$, where again n is the number of variables, and m the number of edges in the graph. Again generalized arc consistency can be achieved in $O(m+n)$. In addition, flow algorithms are incremental and each time a value is removed from a domain of a variable, the filtering algorithm should not start from scratch, see (Régin 1999).

The propagation of the gcc, illustrated in the lower part of Fig. 2, leads to

```
W :: [e]
X :: [c]
```

```
Y :: [f]
Z :: [d,f]
```

Beldiceanu's *Global Constraints Catalogue* (Beldiceanu et al. 2005) lists some 250 constraints which are explicitly described in terms of graph properties and/or automata.

As we have seen, the structure of the global constraint enables the definition of efficient filtering algorithms for removing all inconsistent values, i.e., achieving hyper arc consistency. However, hyper arc consistency can be achieved in polynomial time only if the constraint represents a problem which can be solved with a polynomial time algorithm. In some cases, global constraints represent NP-complete or NP-hard problems and therefore, hyper-arc consistency cannot as far as we know be achieved in polynomial time. In these cases, the constraint embeds a filtering algorithm that enforces a weaker notion of consistency.

As an example, let us consider the cumulative constraint seen in Sect. 2.2 whose parameters are: a list of variables $[S_1, \dots, S_n]$ representing the starting time of all activities sharing the resource, their duration $[D_1, \dots, D_n]$, the resource consumption for each activity $[R_1, \dots, R_n]$ and the available resource capacity C:

$$\mathit{cumulative}([S_1, \dots, S_n], [D_1, \dots, D_n], [R_1, \dots, R_n], C)$$

Many filtering algorithms have been implemented for this constraint, none achieving generalized arc consistency of course, see (Baptiste et al. 2003). A successful algorithm that can be applied if $C = 1$, i.e., if the resource is unary, is the edge finder, introduced by Carlier and Pinson (1995), and embedded as filtering algorithm in constraint programming by Baptiste et al. (1995), see Sect. 4.3.1. Variants of the cumulative constraint for preemptive activities, for unary and cumulative resources are presented in Baptiste et al. (2003).

4.3.1 One-machine scheduling: propagation example

As an example, we show a couple of propagation algorithms which can be applied to the single machine version of the *cumulative* constraint. Suppose that three activities whose start time variables are $Start_1$, $Start_2$ and $Start_3$ have to be scheduled on the same single capacity resource. The durations of these activities are respectively 10, 4 and 5. Suppose now that, due to release date and deadline constraints, the domains of variables *Start* are the following:

$Start_1 :: [1..10]$
$Start_2 :: [0..10]$
$Start_3 :: [1..13]$

The cumulative constraint imposed is therefore:

$$\mathit{cumulative}([Start_1, Start_2, Start_3], [10, 5, 5], [1, 1, 1], 1)$$

If the cumulative constraint has a filtering algorithm implementing the edge finder algorithm (Carlier and Pinson 1995) then the propagation achieved is the following:

$Start_1 = 10$
$Start_2 = 0$
$Start_3 = 5$

In fact, if activity 1 starts before 9, say at time 1, its earliest completion time is 11. If this is the case, there is no available space for activities 2 and 3. Consequently tasks 2 and 3 must precede task 1, but in this case one of them must start at time 0, in order for them to be both completed before 10. This is only possible if task 2 starts exactly at time 0. Therefore task 3 starts at time 5, and task 1 starts at time 10.

4.4 Global constraints with costs

4.4.1 Optimality-based pruning

So far, we have considered feasibility based pruning, i.e., those values that are pruned are proven not feasible, so they are not part of any consistent solution. When an objective function is present in the problem, we could exploit this information for pruning those values that are not part of any optimal solution. In fact, if we have a reference solution, this solution represents an upper bound (resp. lower bound) for our minimization (resp. maximization) problem. Therefore, all values that are not part of improving solutions could be pruned.

For this purpose, two approaches have been studied. When the problem is polynomially solvable, hyper arc consistency can be achieved also on cost, so as to prune all infeasible and sub-optimal values. If, instead, the problem is NP-hard then we should rely on a relaxation of the constraint for pruning sub-optimal values. In this case, clearly, hyper arc consistency cannot be achieved.

4.4.2 Polynomial time cost based filtering

Let us see the first case: the constraint represents an *alldifferent* with costs, also called weighted *alldifferent*. Suppose that each value has an associated cost and the objective function is the minimization of the sum of costs. In addition to the parameters of the *alldifferent* there is a variable representing the objective function Z whose domain is $[D_{\min}..D_{\max}]$ where $D_{\min}$ is the problem lower bound and $D_{\max}$ is value of the best solution found so far (the upper bound). This constraint represents the assignment problem: considering the mapping shown in the previous Section this constraint has the same Integer Programming model of the *alldifferent* without costs with in addition the following objective function:

$$\min \sum_i \sum_j x_{ij} c_{ij}$$

The coefficient matrix is again totally unimodular. Therefore, the problem is polynomial. In this case achieving hyper-arc consistency is doable and the algorithm is based on graph theory. The graph has the same structure as the one built for the *gcc*. We have variable nodes, value nodes, a source s and a sink t. Each arc has an associated capacity equal to 1 and a cost. The cost is 0 for those arcs that start from the source and end in variable nodes and for those arcs that start from value nodes and end in the sink. The cost is instead c_{ij} on those arcs that start from variable nodes and end in value nodes. The constraint is consistent if:

– a feasible flow f from s to t exists, its value is n corresponding to the number of variables and $cost(f) \le D_{\max}$ and
– the minimum cost $s - t$ flow f has a value n and $cost(f) \ge D_{\min}$

In the same way, also for the global cardinality constraints with costs there is a polynomial time algorithm for achieving hyper-arc consistency based on network flows presented in Régin (1999).

4.4.3 Relaxation and reduced costs

When the constraint represents an NP-hard problem, only an approximation of hyper-arc consistency can be achieved. Therefore, a possibility is to embed in the constraint a relaxation of the constraint itself as proposed in Focacci et al. (1999).

Whatever relaxation we use, the unique requirement concerns its output. The relaxation should return three results:

– the optimal solution of the relaxed problem x^*;
– the optimal solution value *LB*. This value represents a lower bound on the objective function value;
– a gradient function $grad(X_i, j)$ measuring the variable-value assignment cost.

These pieces of information are exploited by the CP solver not only for filtering purposes, but also for guiding the search toward promising (in terms of costs) branches.

As explained in the previous sections, also in this case we need a mapping between the CP variables involved in the constraints and those involved in the relaxation. This mapping depends on the chosen solver and on the chosen relaxation. A mapping that is often used is the one proposed in the previous section. It was first suggested by Rodosek et al. (1997). We have a binary variable x_{ij} corresponding to variable-values pairs in CP. In particular, $x_{ij} = 1$ if $X_i = j$ in CP, while $x_{ij} = 0$ if $X_i \neq j$ in CP.

Constraints $\sum_{j \in D_i} x_{ij} = 1 \ \forall i$ are part of the mapping and impose that exactly one value should be assigned to each CP variable.

In addition, if a variable value assignment in CP has a cost, then the obvious link between the objective function variable of the CP model Z and the lower bound value *LB* computed by solving the relaxation is $LB \leq Z$, or equivalently $LB = D_{\min}$.

Given this mapping, the variable x_{ij} in the relaxation corresponds to the value j in the domain of the CP variable X_i. Thus, reduced costs $\bar{c}_{ij}$ provide information on the cost of CP variable domain values, $grad(X_i, j) = \bar{c}_{ij}$.

Given the results provided by the relaxation we have a first (trivial) propagation based on the optimal solution value *LB* of the relaxation. This value is a lower bound on the objective function Z of the overall problem we are solving. As we mentioned, $LB = D_{\min}$. If *LB* is greater or equal than the upper bound of the domain of Z, a failure occurs since the domain of Z becomes empty.

More interestingly, we can implement the propagation from the gradient function $grad(X_i, j)$ towards the problem decision variables $X_1, \ldots, X_n$. The gradient function provides an optimistic evaluation on the cost of each variable-value assignment. Given this information, we can compute an optimistic evaluation on the optimal solution of a problem where a given variable is assigned to a given value, i.e., $LB_{X_i=j}$ (we can do this computation for each variable and all values belonging to its domain). Thus, if this evaluation is higher than the best solution found so far, the value can be deleted from the domain of the variable. More formally, for each domain value j of each variable X_i, we can compute a lower bound value of the sub-problem generated if value j is assigned to X_i as

$$LB_{X_i=j} = LB + grad(X_i, j)$$

If $LB_{X_i=j}$ is greater or equal to the upper bound of the domain of Z, j can be deleted from the domain of X_i.

This filtering algorithm performs a real back-propagation from Z to X_i. Such a technique is known in Mathematical Programming (MP) as *variable fixing*. However, variable fixing in the MP context does not trigger in general an iterative process like constraint propagation, as happens in CP. Indeed, removing values from variable domains triggers constraints imposed on modified variables, and it appears therefore particularly suited for CP.

An important point which should be stressed is that this filtering algorithm is general and can be applied whenever the relaxation is able to provide these pieces of information. The filtering algorithm is independent on the structure of the relaxation.

5 Linear and non-linear programming with CP

5.1 Integer linear programming via CP

CP models allow users to formulate constraints in the same way as mathematical models. Let us model a toy transportation problem with

- Three plants, with capacities 500, 300 and 400
- Four clients, with demands 200, 400, 300, 100
- Transportations costs, per unit load, between each plant and each customer

The requirement is to meet the demands from the plants minimizing demand.

The mathematical model is:

```
minimize
    10 A1 + 7 A2 + 11 A3 +
     8 B1 + 5 B2 + 10 B3 +
     5 C1 + 5 C2 +  8 C3 +
     9 D1 + 3 D2 +  7 D3

subject to
    A1 + A2 + A3 = 200
    B1 + B2 + B3 = 400
    C1 + C2 + C3 = 300
    D1 + D2 + D3 = 100

    A1 + B1 + C1 + D1 =< 500
    A2 + B2 + C2 + D2 =< 300
    A3 + B3 + C3 + D3 =< 400
```

The CP model (in ECLiPSe) is:

```
:- lib(eplex).                                   % Line 1

main(Cost, Vars) :-                              % Line 2
    Vars = [A1, A2, A3, B1, B2, B3, C1, C2, C3, D1, D2, D3],
    ( foreach(Var,Vars) do Var $>= 0.0 ),    % Line 4

    A1 + A2 + A3 $= 200,
    B1 + B2 + B3 $= 400,
    C1 + C2 + C3 $= 300,
    D1 + D2 + D3 $= 100,
    A1 + B1 + C1 + D1 $=< 500,
    A2 + B2 + C2 + D2 $=< 300,
    A3 + B3 + C3 + D3 $=< 400,

    optimize(min(
        10*A1 + 7*A2 + 11*A3 +
         8*B1 + 5*B2 + 10*B3 +
         5*C1 + 5*C2 +  8*C3 +
         9*D1 + 3*D2 +  7*D3), Cost).
```

The main difference is in the first four lines of the CP model. The first line specifies which type of solver will be used for this problem. (*eplex* is just a name given to refer to a linear solver.) The actual package used to implement this solver in ECLiPSe can be CPLEX (ILOG Optimization Team 2008) or Xpress-MP (Guret et al. 2002) or the open source software COIN-OR Foundation (2009). In the following when we wish to refer to an external integer/linear solver, such as CPLEX or Xpress-MP or COIN-OR, we shall call it an *eplex* solver.

Line 2 encapsulates this model as a procedure called *main*. This procedure can be invoked from any other CP model. In this toy example the parameters to the model are all fixed. Naturally they could also be passed as parameters in to the procedure. When the user runs the procedure *main(Cost, Vars)*, the optimal cost with be returned together with a solution giving a specific value for each decision variable $A1, A2, \dots D3$ denoting a quantity transported from a plant to a customer.

Line 3 creates a list of variables called *Vars*. This list is used to return the solution in the procedure *main(Cost, Vars)*.

Line 4 constrains each variable to be a non-negative number. By default a variable can take positive or negative values. This model simply uses all the default settings of the *eplex* solver.

None of the variables in this problem are constrained to be integers. If, for some reason, we required all the transportation variables to be integer, we could add a line `integers(Vars)` and then the solver would run a built-in branch-and-bound search to find an optimum integer solution.

The transportation example is one that is very easy to represent as a mathematical model. We do not need CP to model toy problems such as this. CP is a very powerful modelling language because it allows more complex problems to be modelled naturally, and to be solved by tailored algorithms that may use multiple constraint solvers.

5.2 Row generation via CP

Problem specification. Let us invent a toy non-linear problem to illustrate the possibilities. There are two variables, W and X, with values in the range $0 \dots 10$, and W is an integer. There is just one constraint, that $2 * W + X = 5$.

Suppose we wish to minimize the following non-linear expression: $W + \sqrt{X}$.

To find the true minimum we include a variable *SqrtX* in our model, and use a linear approximation of the constraint $SqrtX = \sqrt{X}$.

Initially we post the linear constraint $2 * W + X = 5$, and a linear approximation to $SqrtX = \sqrt{X}$: $SqrtX \geq 0$, $SqrtX \times \sqrt{10} \geq X$. We minimise $W + SqrtX$.

For this problem we generate new linear constraints during search which are added to the initial linear approximation to make it better and better. Search stops when the optimal values of X and $SqrtX$ which satisfy the current linear approximation are “near enough” in the sense that $abs(SqrtX - \sqrt{X}) \leq \epsilon$.

Search for a sufficiently accurate solution. Our nonlinear search `nlsearch(X, SqrtX)` can halt when the difference between the two is less than this value.

1. Solve the linear relaxation, returning relaxed solution $ValX$ for X and $ValSqrtX$ for $SqrtX$.
2. If $abs(ValSqrtX - \sqrt{ValX}) \leq \epsilon$, then stop
3. Otherwise create a search node. On one branch constrain $X \leq ValX$ and on the other constrain $X \geq ValX$.
4. Tighten the linear approximation to $\sqrt{X}$. Specifically if the bounds of X are now Min and Max, impose $SqrtX \times \sqrt{\text{Min}} \leq X$ and $SqrtX \times \sqrt{\text{Max}} \geq X$. Notice that $ValX$ is now either Min or Max, and therefore search will stop if X again takes the value $ValX$, because the linear approximation to $\sqrt{X}$ is exact at these points.
5. Go to 1

Optimisation. To complete the encoding in CP, the above search routine is posted as a parameter to a minimisation procedure thus

```
minimize( nlsearch(X, SqrtX), Opt )
```

The `minimize` procedure is a generic branch-and-bound procedure that can handle any search routine.

The beauty of CP is that it enables such user-controlled search routines to be combined with a linear solver and a generic branch-and-bound routine. The encoding of this example in the ECLiPSe CP language is presented in detail in Apt and Wallace (2006).

5.3 Logical combinations of constraints via LP and CP

Consider a problem that has logical combinations of linear and non-linear constraints. The logical combinations can be expressed using reified constraints, with extra boolean variables to record their entailment or disentailment. Suppose the problem involves a finite domain variable X which can take integer values in the range `0..3`, and there are logical conditions

$$(X = 1 \leftrightarrow Cons_1(\overline{Y_1})) \wedge (X = 2 \leftrightarrow Cons_2(\overline{Y_2})) \wedge (X = 3 \leftrightarrow Cons_3(\overline{Y_3}))$$

These kinds of constraints can be naturally expressed in CP, and automatically mapped to a form in which the reified linear constraints and their boolean variables can be mapped to integer/linear form and sent to the *eplex* solver.

Indeed, the above compound constraint can be mapped to the following:

$$\begin{aligned} &B_1 + 2 * B_2 + 3 * B_3 = X \wedge B_1 + B_2 + B_3 = 1 \\ &\quad \wedge RB_1 \geq B_1 \wedge RB_2 \geq B_2 \wedge RB_3 \geq B_3 \\ &\quad \wedge Cons_1(\overline{Y_1}, RB_1) \wedge Cons_2(\overline{Y_2}, RB_2) \wedge Cons_3(\overline{Y_3}, RB_3) \end{aligned}$$

If each of the reified constraints $Cons_j(\overline{Y_j}, RB_j)$ can be mapped to integer linear form then the problem can be automatically mapped to an efficient integer/linear model and solved by *eplex*.

More generally the constraints may be sent to both the *eplex* solver and other solvers such as the CP propagation solver. In this case the above constraints provide an automatic communication link between the finite domain variable X in the CP solver, and the boolean variables B_1, B_2, B_3 in the *eplex* solver. If some of the reified constraints are non-linear, then this kind of communication between different solvers is a key part of the algorithm. The constraints which maintain the consistency of the variables in the different solvers are termed *channeling* constraints (Cheng et al. 1996).

CP models allow a great deal of flexibility, not just in the combination of solvers, but also in search. Suppose, even for an integer/linear problem, that the problem solver decides to encode a search algorithm instead of using the *eplex* built-in *optimize*. Then instead of using

```
    optimize(min(Objective), Cost)
```

the problem solver can simply set up the linear constraints, and invoke the specially designed search routine:

```
Cost $=  Objective                       % Line 1
eplex_solver_setup(min(Objective), Cost, [], [bounds]),
<Search>                                 % Line 3
```

In line 1, a new variable *Cost* is introduced, and equated with the objective function which is here called *Objective*. In line 2, the linear constraints (specified earlier in the program) are loaded in to the linear solver. The last argument `[bounds]` states that the linear solver will be rerun every time the bounds of any of the variables are tightened. Such a tightening may result from a search step or from propagation within another solver.[1] Each time the linear solver is rerun, the *Cost* variable lower bound is updated to the new optimum found by the solver. In line 3 the specified search procedure is called.

5.4 Linear solving as a propagation agent

The use of Operations Research techniques in Constraint Programming has shown to be very effective in practice not only when the relaxation is embedded within a CP global constraint, but also using a linear solver as if it was a global constraint. These techniques were proposed first in Beringer and De Backer (1995) and extended by Refalo (2000).

For this purpose the problem model should be stated in the CP solver and also passed to the linear programming solver. For being effective the interaction between the two solvers should last during the overall computation. In fact, the linear programming solver should achieve a tight integration with the standard constraint propagation. Therefore, a communication link between the CP constraints and the linear solver should be established. For this purpose, quite often the linear solver is considered as a software component similar to other constraints, interacting with them and exchanging information. Three pieces of information should be passed back and forth: variable fixing, bound reduction and the optimal solution of the relaxation. Each time the CP propagation infers a new bound, it should be passed as a linear constraint to the linear solver which takes it into account in next re-computations. Also, variable fixing from the LP solver are passed to the CP solver as traditional propagations. The solution of the relaxation, i.e., the lower bound can be used as described in Sect. 4.4.3.

Clearly this procedure is more effective if the bound is tight. Therefore, Refalo (2000) has proposed to tighten the relaxation with cutting planes added during search and coming from

[1]The third argument of *eplex_solver_setup* allows the programmer to specify a list of options. In this example the list is empty.

CP global constraints. In particular, cutting planes can be derived from global constraints representing sub-problems and added to the Linear Programming model so as to tighten the overall relaxation. It is interesting to note that cutting planes can be derived after domain reduction from the CP side, so as to maintain the tightness of the relaxation during the whole solution process. Refalo has provided many examples and shown that it is very effective in practice.

5.5 Constraint Integer Programming—CIP

A recently proposed framework for a tight integration of CP, MIP and SAT methodologies is the Constraint Integer Programming paradigm—CIP presented in Achterberg et al. (2008).

A Constraint Integer Program (CIP) is defined as the optimization problem

$$c^* = \min\{cx \mid \mathcal{C}(x), x \in \mathbb{Z}^I \times \mathbb{R}^{N \setminus I}\}, \tag{1}$$

where $\mathcal{C} = \{C_1, \dots, C_m\}$ is a set of constraints, and $I \subseteq N = \{1, \dots, n\}$. By definition, the constraint set $\mathcal{C}$ has to be such that, once the integer variables have been assigned values, the remaining problem becomes a linear program.

The corresponding solving framework is called SCIP (Solving Constraint Integer Programs) see Achterberg (2009). SCIP allows total control of the solution process and access to detailed information from the solver. SCIP is a framework for branching, cutting, pricing and propagation, and its implementation is based on the idea of plug-ins, which makes it highly flexible and easily extensible. The main types of SCIP plug-ins are:

Constraint handlers: Constraint handlers represent the semantics and check the feasibility of a given constraint class and provides algorithms to handle constraints of the corresponding type.

Domain propagators: Constraint based domain propagation is supported by the constraint handler concept of SCIP. In addition, SCIP features two dual domain reduction methods that are driven by the objective function, namely objective propagation and root reduced-cost strengthening.

Conflict analyzers: Similarly to SAT, SCIP generalizes conflict analysis to CIP and, as a special case, to MIP. This aspect is more detailed in Sect. 6.3.

Cutting plane separators: SCIP features separators for many different types of cuts. For cut selection, SCIP uses *efficacy* and *orthogonality*, and parallelism with respect to the objective function.

Primal heuristics: SCIP relies on different heuristics, which can be classified into four categories: rounding, diving, objective diving, and improvement.

Node selectors and branching rules: SCIP implements most of the well-known branching rules, and it allows the user to implement arbitrary branching schemes.

Pre-solving: SCIP implements a full set of primal and dual pre-solving reductions for MIP problems. It also uses the concept of *restarts*, which are a well-known ingredient of modern SAT solvers.

Beside being a framework for enabling MILP, CP and SAT integration, SCIP can also be used as a pure MILP or as a pure CP solver.

5.6 Lagrangian relaxation

5.6.1 The benefits of Lagrangian relaxation

Lagrangian relaxation is a method of achieving tighter cost bounds for problems which include different kinds of constraints (Lamaréchal 2003). If tight bounds can be achieved for one class of *good* constraints, then the remaining constraints are relaxed, and instead transformed into extra terms in the optimization function. Each extra term includes a coefficient called the Lagrangian multiplier. The resulting problem is called the *Lagrangian subproblem*.

Suppose there are no integrality constraints. If the optimal Lagrangian multipliers are used, then the optimum value of the Lagrangian subproblem is the optimum value of the original problem: there is no duality gap.

Suppose, however, there are integrality constraints. Lagrangian relaxation behaves quite differently from linear relaxation. An optimal solution to the linear relaxation of the integer/linear problem may violate integrality constraints, but it satisfies all the linear constraints. An optimal solution to the Lagrangian subproblem may violate the relaxed constraints. On the other hand, by a judicious choice of which constraints to relax, we can ensure that solutions to the Lagrangian subproblem satisfy the integrality constraints instead.

In this case the optimal solution of the Lagrangian subproblem is feasible for the *good* constraints—including all the integrality constraints—and it also "approximates" the relaxed constraints. The remaining work is to manipulate the solution so as to satisfy the relaxed constraints as well.

5.6.2 Lagrangian relaxation and CP/LP hybrids

Of course hybrid CP/LP techniques are typically used for problems involving different kinds of constraints. Some constraints, such as application-specific scheduling constraints like `cumulative` (see Sect. 4.3), are best handled by CP alone. Some constraints are linear and well-handled by LP. Often, despite integrality constraints on the variables, the linear solver meets our needs because the linear constraints are unimodular.

However there are many constraints that are well-handled by CP and MIP. Amongst all the constraints for which there is a linear relaxation, it is often the case that there are both *good* constraints, whose linear relaxation is tight, and other complicating constraints which weaken the relaxation—often so much that it is practically useless.

These complicating constraints are therefore omitted from the LP subproblem, and handled using propagation. This is, perhaps, the typical form of a hybrid CP/LP algorithm. It is very effective in case the complicating constraints do not involve any variables occurring in the cost function. In this case propagation effectively focusses the LP solver on feasible subspaces where it can quickly discover the optimum.

For problems where the complicating constraints are also important for optimization, however, the LP relaxation of the *good* constraints is not so useful. It is for such problems that Lagrangian relaxation plays a key role within a CP/LP hybrid.

In this case the complicating constraints are handled both by CP, using propagation, and by LP, using Lagrangian relaxation. The cooperation between CP and LP, described earlier, results in improved handling of the complicating constraints, and the tighter problem relaxation overall results in better search control. The weakness of Lagrangian relaxation, that the solution of the Lagrangian subproblem may violate the relaxed constraints, is compensated for by CP. The constraints which are relaxed in the Lagrangian subproblem are still enforced by CP. Propagation on these constraints rules out variable assignments that are infeasible, and this information is added to the Lagrangian subproblem in the form of tighter bounds. Thus the Lagrangian subproblem is driven towards a version where the optimal solution also satisfies these relaxed constraints.

5.6.3 An example of Lagrangian relaxation in a CP/LP hybrid

An example of Lagrangian relaxation within a hybrid CP/LP algorithm is presented in Ouaja and Richards (2005). The application is a multi-commodity network flow problem, comprising network flow constraints and edge capacity constraints. There is a set of demands to be placed on the network, each of which must be assigned a single route through the network.

The *good* constraints are the network flow constraints: linear programming quickly returns an optimal path for each flow, whatever cost is associated with each edge. This, incidentally, satisfies the integrality requirement that a single route be assigned to each demand.

The complicating constraints are the capacity constraints: the sum of the demands routed through any edge must not exceed the capacity of the edge. Whilst these are simple linear (knapsack) constraints, the combination of the flow and knapsack constraints make the problem NP-hard. The optimal linear relaxation splits up a single demand through multiple routes.

The capacity constraints are, instead, relaxed into the optimization function. The resulting problem has only the flow constraints, and an extended optimization function. The core problem is therefore a set of (independent) shortest path problems. Unfortunately the optimal Lagrangian multipliers still cannot guarantee that the resulting Lagrangian subproblem has an optimal solution that satisfies the relaxed capacity constraints.

The decision variables for this problem are binary variables that record whether or not a particular demand crosses a particular edge in the network. The CP engine controls the search by choosing at each search node one such variable which is set to 0 or to 1. This decision triggers constraint propagation which, for example, if a boolean is set to 1, sets to 0 all demand booleans which would now cause an overflow. More subtly if a boolean is set to 0 constraint propagation can set other booleans to 1, and by maintaining *integer* bounds for boolean variables it can detect a failing branch earlier than the linear solver.

Subsequently the Lagrangian subproblem is repeatedly evaluated in order to improve the Lagrangian multipliers, until a (near) optimal setting has been discovered for each Lagrangian multiplier. For each intermediate solution of the Lagrangian subproblem the cost lower bound for the original problem is tightened.

Constraint propagation is triggered at each search node, when a decision variable is fixed, and after each solution to the Lagrangian subproblem, when the cost lower bound is tightened.

Two additional features of the algorithm enhance its overall efficiency.

Firstly, the capacity constraints on each edge are large—in principle every decision variable appears in every such edge constraint. The dualisation into the optimization function, would therefore make it unnecessarily complicated. Instead capacity constraints are only added when they are violated. In this case they are both added to the CP solver and relaxed into the optimization function at the next node. Finite domain propagation is thus focussed on constraints and assignments that are violated by Lagrangian subproblem solutions.

Secondly new constraints based on reduced costs, as described in Sect. 4.4.3, are also added after each tightening of the cost lower bound.

The benefit of Lagrangian relaxation—combined with constraint propagation—over linear relaxation is reflected in experimental results on a set of large benchmarks (with between 300 and 600 demands). The hybrid algorithm solved some 93% of the benchmarks whilst MIP could only solve around 53%.

6 Constructive search and optimization

6.1 Exploring alternatives

Constraint programming search is similar to MIP search in that it explores a search tree whose leaves are either solutions or inconsistent nodes. In a MIP search tree, each node has at most two descendants, but in a CP search tree a node may have many descendants. This is a key difference between CP and MIP search.

Secondly in a MIP search tree a child of a node differs from its parent just in one (tighter) bound on one variable. In a CP search tree the child typically has one variable instantiated to a fixed value. This form of search is called "labelling". Nevertheless in a CP search tree the extra constraint added at a child node may instead be a tighter bound on a single variable, or even an arbitrary constraint on any subset of the problem variables. For example in a scheduling application the added constraint may enforce that one task cannot be scheduled until another has finished. This is expressed as a constraint on the start times of two variables, S_1 and S_2 (if D_2 is the duration of the second task, then $S_1 \geq S_2 + D_2$). These are some other forms of CP search that may be used instead of labelling.

Backtrack search is a basic control structure in computer science and finds particular application also in Constraint Programming. Backtracking is a well-defined process for exploring alternatives after a failure. In the case of tree search a failure occurs when a node proves to be inconsistent, or to have no admissible descendants. The basic form of backtracking algorithm is called chronological backtracking. If this algorithm encounters a failure then it always backtracks to the previous decision, therefore it is called chronological.

There are many variants of backtrack search. Credit search (Beldiceanu et al. 1997) is a tree search method where the number of nondeterministic choices is limited a priori. This is achieved by starting the search at the tree root with a certain integral amount of credit. This credit is split between the child nodes, their credit between their child nodes, and so on. A single unit of credit cannot be split any further: subtrees provided with only a single credit unit are not allowed any non deterministic choices, only one path though these subtrees can be explored, i.e. only one leaf in the subtree can be visited. Subtrees for which no credit is left are pruned. Clearly credit search is an incomplete exploration method so the solution to a feasible problem might eventually be missed.

Another widely used search strategy is the Limited Discrepancy Search. Limited Discrepancy Search (LDS) was first introduced by Harvey and Ginsberg (1995). The idea is that one can often find the optimal solution by exploring only a small fraction of the space by relying on tuned (often problem dependent) heuristics. Note that the word heuristics has a different meaning w.r.t. that used in Operations Research. In fact, with heuristics we mean a method that serves as an aid to problem solving. It is sometimes defined as any *rule of thumb*. Technically, a heuristic is a function that takes a state as input and outputs a value for that state, often as a guess of how far away that state is from the solution. However, a perfect heuristic is not always available. LDS addresses the problem of what to do when the heuristic fails.

Thus, at each node of the search tree, the heuristic is supposed to provide the *good* choice (corresponding to the leftmost branch) among possible alternative branches. Any other choice would be *bad* and is called a *discrepancy*. In LDS, one tries to find first the solution with as few discrepancies as possible. In fact, a perfect heuristic would provide us the optimal solution immediately. Since this is not often the case, we have to increase the number of discrepancies so as to make it possible to find the optimal solution after correcting the mistakes made by the heuristic. However, the goal is to use only few discrepancies since in general good solutions are provided soon.

LDS builds a search tree in the following way: the first solution explored is that suggested by the heuristic. Then solutions that follow the heuristic for every variable but one are explored: these solutions are that of discrepancy equal to one. Then, solutions at discrepancy equal to two are explored and so on.

It has been shown that this search strategy achieves a significant cutoff of the total number of nodes with respect to a depth first search with chronological backtracking and iterative sampling (Langley 1992). LDS can be seen both as a complete and incomplete search strategy. If we search up to a maximal discrepancy, it is incomplete. If instead the tree is exhaustively explored with no limitation to the maximal discrepancy, it is a complete, possibly inefficient, method.

6.1.1 CP heuristics

Traditionally CP search is depth-first. The reason for this is the relatively large amount of data necessary to represent a search node. This data includes a compact representation of the domain of each variable, which in many cases reduces to a lower and upper bound, as well as information about the current state of the problem constraints, which are often simplified during search. Increasingly systems are supporting more flexible forms of search, such as depth-first until a solution is found and then best-first. However issues around the number of "open" nodes to maintain are well-understood in the MIP community and CP brings little new in this respect. In this section we will therefore focus on heuristics that are special to CP, and simply assume a depth-first search procedure. These heuristics can be easily adapted to best-first search where it is available.

For pedagogical purposes we will also assume a labelling search, in which a variable is instantiated to a different fixed value for each child of a search node. For labelling search we can distinguish two kinds of heuristics: *variable* choice and *value* choice. The variable choice controls the order in which variables are labelled during search. This ordering dictates the shape of the search tree.

Assuming, firstly, that the variable ordering is chosen before search begins, and remains fixed throughout search. If there are three variables, X, Y and Z with domains $X \in \{a\}$, $Y \in \{b, c\}$ and $Z \in \{d, e, f\}$, then the orderings $\langle X, Y, Z\rangle$ and $\langle Z, Y, X\rangle$ yield two quite different search trees, as shown in Fig. 3.

Both search trees have the same number of leaves (they must have!), but they have a quite different number of internal nodes. It is therefore preferable to explore the left-hand side tree with less internal nodes. Good heuristics are those that chose first the most constrained variables, i.e., the variables with less values in the domain.

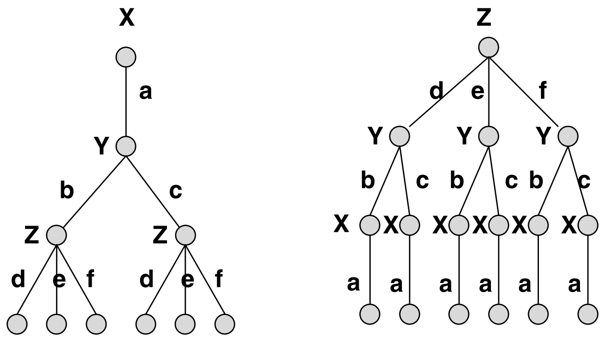

Fig. 3 Two search trees based on different *variable* orderings

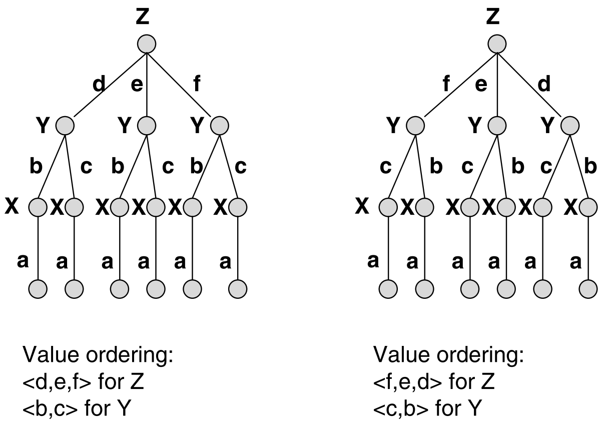

Fig. 4 Two search trees based on different *value* orderings

We now illustrate the effect of different *value* orderings on the same problem. By reversing the value orderings we can obtain the two trees shown in Fig. 4.

These trees illustrate static ordering heuristics. In practice search orderings are computed dynamically during search, so the variable and value orders are different on different branches of the tree. However these simple trees can be used to illustrate two principles.

The first principle is for variable ordering heuristics: choose earlier variables that are more difficult to satisfy. This ordering tends to detect wrong choices as quickly as possible because failure occur early in the search tree. Moreover for "easier" variables, any values that are not part of a feasible, or optimal, solution are eliminated because they are incompatible with the previously chosen values for the other variables. Choosing the most difficult variable to satisfy means for example chose the variable with less values in the domain, or chose the variable appearing in the highest number of constraints.

Examine the two trees illustrated in Fig. 3 with different variable orderings. In the first tree the variables with fewer alternative values are chosen first: this heuristic is termed "first fail". This tree has fewer nodes than the other tree where the variable with the most alternative values, Z, is labelled first. In particular if propagation prunes away nodes lower in the search tree, the difference is even more marked. There are only 4 nodes in the top three layers of the first tree, but in the second tree the top three layers have 10 nodes.

The second principle is for value ordering. In this case we clearly seek to choose the best value first, and only try the less promising values later, if at all. Interestingly, in the case of a feasibility problem, if there are no solutions then it does not matter what value ordering we choose: the number of nodes is the same. However for depth-first branch-and-bound optimization, the value ordering is crucial in order to find good solutions early and thereby prune the search.

Suppose for instance that values in the example of Fig. 4 have a cost: value a has cost 1, value b cost 2, value c cost 3, value d cost 4, value e cost 5 and value f cost 6. We have to minimize the cost of all assignments. Clearly the optimal solution has cost 7. This solution is found in the first branch in the leftmost tree, while it is found in the last but one branch in the other tree. Therefore a good value ordering heuristics is the one choosing first low cost values.

Although we have distinguished variable and value ordering heuristics, there is an important class of heuristics that address variable and value orderings simultaneously. These are relaxation heuristics, which are based on the solution of a relaxed problem at each node of the search tree. The relaxed problem is an "easy" subproblem which, typically, includes all the original problem variables but only a subset of the original problem constraints. If the (optimal) solution to the subproblem is also a feasible solution for the original problem, we have reached a leaf node of the search tree. If, on the other hand, some of the original problem constraints are violated, we choose a variable assignment that conflicts with the constraint. We then assign a different value to the variable in order to satisfy the constraint. For complete search all such alternative assignments to the variable are tried.

If we allow more general search than just labelling, then MIP search is an example of this approach. Moreover the constraints omitted from the relaxed problem can be used actively to guide the search heuristics. An algorithm that uses CP to handle the relaxed constraints in this way is called unimodular probing (El Sakkout and Wallace 2000).

6.1.2 How to deal with the optimization function

The way Constraint Programming copes with objective functions is quite naive. CP during search solves a series of feasibility problems imposing for each of them a constraint, called *bounding constraint*, imposing that future solutions should be better than the best one found so far. For instance, suppose we find a solution whose cost is 100. The CP solver imposes in backtracking that the domain variable representing the cost function C should be less than 100, updating the upper bound of its domain. Clearly the proof of optimality is obtained when the problem with the last added constraint fails. The last solution found is optimal.

This method is very easy but inefficient. The reasons are mainly two: the first is that the link between variable C representing the objective function and the problem decision variables often performs a weak propagation. Consider, as an example, a scheduling problem where the objective function to be minimized is the sum of setup times. It does not depend directly on the value of the activities starting times (problem variables), but it depends on their relative positions. In this case, imposing the bounding constraint affects the activities' domains only at the very low levels of the search tree when most decisions have been taken. The second reason concerns the fact that the bounding constraint uses an upper bound for minimization problems, but no use of lower bound is done.

For this reason, many possibilities have been explored for improving this aspect. One possibility is to include a linear relaxation into the CP scheme, as shown in Sect. 5.4, or exploiting cost based information coming from global constraints, as shown in Sect. 4.4. Of course the tightness of the relaxation is important for guiding the search, and accelerating branch-and-bound. The amount of propagation is also critical for search guidance, and early detection of dead branches in the search tree. Let us simplistically divide search control into two components:

– The choice of which variable to constrain at the next search node
– The choice of how to constrain that variable

The search heuristics supported by propagation are of the first type—how to choose which variable to constrain next. The search heuristics supported by the relaxations are of both types, but they are particularly effective for the second type of choice: how to constrain the variable.

Moreover the two approaches, CP and LP, not only complement each other but they also support each other. The linear relaxation yields a bound on the cost variable, which enables more propagation to occur. Propagation, conversely, tightens variable bounds which in turn tightens the linear relaxation. This again yields a tighter bound on the cost variable which enables more propagation. Conversely more propagation supports an even tighter linear relaxation.

6.2 One-machine scheduling: search

Let us consider again the one-machine scheduling example and let us focus on search. Note that the start time variables are then sorted according to an ordering heuristic specified in `KeyStarts`, and the `minimize` routine explores a depth-first branch-and-bound search tree to solve the problem.

The heuristic `KeyStarts` is used in the search routing and is the latest possible start time (`Due - Dur`).

```
search([]).
search(Starts) :-
    remove(Start,Starts, Rest),
    get_min(Start,Min),
    Start $= Min,
    search(Rest).

remove(Start, [_Key-Start|Rest], Rest).
remove(Start, [First | KeyStarts],[First | Rest] ) :-
     remove(Start, KeyStarts, Rest).
```

The search routine simply selects the next task and schedules it as early as possible. Alternative solutions are obtained by selecting a different task to schedule next. Consequently the sorting of start times does not change the search space, but only the order in which alternatives are explored. The selection of the next task is carried out by `remove(Start, Starts, Rest)`. This predicate first selects the next task (the heuristically most promising one), and then, on backtracking, it selects other tasks. (`get_min(Start, Min, Start $= Min)`) sets the start time to the earliest feasible time. Finally `search` calls itself "recursively" on the list of remaining tasks. When there are no tasks left it stops. This is encoded by (`search([])`) which represents the termination condition when the list of activities to be scheduled is finished.

6.3 Learning during search

Recently the idea of learning additional constraints as a result of failures encountered during search has been taken up in the propositional problem solving (SAT) community, with very good results Marques-Silva and Sakallah (1999). The learnt constraints are clauses (disjunctions of propositional literals), which is the same form as all the other constraints handled by SAT solvers. The combination of clause learning and repeatedly restarting from the root of the search tree supports powerful search pruning and search heuristics enabling early discovery of feasible solutions, or accelerated detection of infeasibility in case there are none.

A novel form of hybrid between FD and SAT solvers has also been very successfully applied, in which every propagation step performed by the FD solver yields an additional learned clause for the SAT solver Ohrimenko et al. (2009).

Current state-of-the-art MIP solvers discard infeasible search nodes, and only learn from infeasible linear relaxations. However the SCIP system integrates learning and constraint handling into a MIP framework Achterberg (2009). In case the linear relaxation at a search node is infeasible, SCIP analyses the LP and identifies a subset of the bounds changes occurring previously in the current branch of the search tree that suffice to render the LP infeasible. The elicited conflict constraint comprises a set of bound changes, responsible directly or indirectly for the infeasibility.

6.4 Decomposition

An alternative way to explore the search space is to decompose the problem at hand into multiple subproblems and use the solver which is most suited for each part. Clearly the solvers should interact and exchange information. One of the most successful examples is the CP-based Benders Decomposition.

6.4.1 Benders decomposition

Benders (1962) decomposition has been studied in the 60's and is an effective method for solving a variety of structured problems. It is particularly suited for those problems where fixing a number of *hard* variables makes the problem simpler.

Instead of blindly trying tentative values for the *hard* variables we can solve a master problem (which takes into account constraints on *hard* variables). After fixing *hard* variables, a subproblem can be solved, taking into account the remaining variables. The process iterates and converges to the optimal solution. The iteration between the master and the subproblem is regulated by the so called Benders cuts. In Operations Research, the subproblem should be a linear program. Hooker and Ottosson (2003) propose to remove this restriction, defining the Logic-Based Benders Decomposition framework. In this setting, the subproblem can be expressed as a Constraint Satisfaction Problem.

An interesting and successful application of Logic-based Benders Decomposition is scheduling with alternative resources, where each activity can run on a set of parallel machines of different speeds and costs.

As an example, taken from Hooker and Ottosson (2003), let us consider a scheduling problem where we want to minimize the cost of the schedule. Each task i costs c_{ij} when executed on the machine j. Each task i has a release date R_i, a deadline D_i and a duration p_i.

The problem can be modelled with two sets of variables: the first being the set of variables t_{ij} for the allocation of tasks to machines. In particular, $t_{ij} = 1$ if the task i is allocated on machine j, 0 otherwise. The second set of variables $Start_i$ are posted for the scheduling part and represent the starting time of task i.

The master problem model is:

$$\min \textstyle\sum_{i,j} c_{ij} t_{ij}, \tag{2}$$

$$\text{subject to} \quad \textstyle\sum_j t_{ij} = 1, \quad \forall i, \tag{3}$$

$$t_{ij} \in \{0, 1\}, \quad \forall i, j, \tag{4}$$

$$\text{Benders cuts computed at iteration } h = 2 \text{ to } H, \tag{5}$$

$$\text{Relaxation of the subproblem} \tag{6}$$

The master problem decides the optimal allocation $\bar{t}$ of tasks to resources. At the first iteration the set of Benders cuts is empty, while it is always important to include a relaxation (6) of the subproblem so as to avoid the generation of trivially infeasible solutions. The relaxation avoids the inefficient generate and test mechanism to happen. Now the subproblem is a scheduling problem with a static allocation of resources to tasks and is a feasibility problem whose model is:

$$Start_i \geq R_i \quad \forall i, \tag{7}$$

$$Start_i + p_i \leq D_i \quad \forall j, \tag{8}$$

$$cumulative_j(Start_i | t_{ij} = 1) \quad \forall j \tag{9}$$

If a feasible schedule does not exist, a Benders cut is generated. It simply states that the previous solution should not be computed again. Clearly, tighter cuts produce a faster convergence: they remove not a single solution but a set of solutions. For instance, if we identify the bottleneck resource r provoking a failure, we can produce a cut stating that only those tasks previously assigned to r not to be assigned again all together to r or any other resource homogeneous with r. The cut in both cases is a linear constraint. As soon as the subproblem finds a feasible solution, it is also optimal for the problem overall.

The above example is the simplest case where the objective function of the overall problem depends only on the master problem variables. In general, it can depend on the subproblem variables only or on both. In these cases, Benders cuts produce bounds and the fix point is reached on the basis of optimality reasoning. Examples can be found in Hooker (2004) and Hooker (2005) and Grossmann and Jain (2001).

7 Improving on non-optimal solutions

7.1 Local search

7.1.1 Contrasting constructive and local search

Branch-and-bound search is the staple of MIP and CP. At each node of the search tree, a simple constraint is added, and some inferences are made. Down each branch of the search tree more and more constraints are added until, at the leaf of the tree the inference is enough to elicit a complete, feasible solution to the problem. If at any stage inference reveals that the current set of constraints cannot be satisfied (or if any feasible solution is shown to be non-optimal), the current branch is fathomed, and search continues on another branch.

"Local search" proceeds very differently. Search is initialized by creating a complete assignment of values to variables. At each search step, this assignment is changed in certain ways so as to try and improve it. An improved assignment violates less constraints, or it has a lower cost, or both. Search proceeds using these steps to move from assignment to assignment around the space of complete assignments, in such a way that the assignments tend to improve.

7.1.2 Propagating changes with invariants

CP systems such as COMET (Van Hentenryck and Michel 2005) support local search through facilities to propagate the consequences of change. COMET has the same modeling power of Constraint Programming languages, but it differs in two ways. First it performs local search and second, it differs in how constraints are built from a software engineering viewpoint.

In CP, constraints have an embedded filtering algorithm which works on domain variables and removes those values that are provably inconsistent. Constraints are maintained in a constraint store during computation and are awaked each time an event happens on a variable involved. The events are: the removal of a domain value, the variable fixing and the changing in a domain bound.

In COMET, constraints are again software components that are maintained in a constraint store and awaked each time one or more variable change value (i.e., the local search is moving from one solution to another).

Constraints are called *differentiable objects* and they maintain properties such as the satisfiability, or the violation degree, and how much the involved variables contribute to it. Constraints can be queried to evaluate the effect of local moves on the properties they preserve.

As an example consider a problem with decision variables $X_1, X_2, \dots X_n$ and cost function $X_1 + X_2 + \cdots + X_n$. When performing local search it is necessary to compute the cost of one, or sometimes many, complete assignments when choosing which changes to make at each search step. The computation required to compute the cost $c = val_1 + val_2 + \cdots + val_n$ of an assignment $X_1 = val_1, X_2 = val_2, \dots, X_n = val_n$ is proportional to the number of variables n. Though each computation is very fast, the computation must be done so often during local search that it can govern the overall performance of the local search algorithm.

However after changing the value of X_i to val'_i, the cost of the new complete assignment is $c + val'_i - val_i$. The computational effort of computing the new cost based on the change from the old cost is therefore independent of the size of n. In COMET the cost expression $Cost = X_1 + X_2 + \cdots + X_n$ is handled by this more efficient computation, so that a change in the value of any of the variables X_i is propagated to the cost very efficiently and incrementally.

The facility to propagate changes efficiently was originally proposed for constraint-based graphics (Sannella et al. 1993), but has proven key to efficient local search algorithms in CP. The efficient computation of changes can be applied to any functional expression involving many variables, for example *sum, maximum, minimum*, and *average*. Functional expressions whose values are efficiently updated in this way are termed *invariants* (Michel and Van Hentenryck 2000).

7.1.3 Driving and driven variables

Many local search algorithms make local changes to a set of core "driving" variables. The remaining "driven" variables' values are then inferred from the values of the driving variables. One example of this is in scheduling where efficient local search algorithms only choose an order for the tasks: a local change just swaps the order of two tasks on the same machine (Nowicki and Smutnicki 1996). The start times of all the tasks on all the machines - and the resulting cost of the schedule—are then derived from the order in which the tasks are executed on each machine. Again these values can be efficiently updated after a local change using invariants (Van Hentenryck and Michel 2005).

A familiar case of this is the Simplex algorithm, though there are no explicit driving variables. In the Simplex a solution is defined by the basis. The values of the problem variables are computed from the choice of basis. The problem variables are the driven variables in this case, and the choice of basis corresponds to the driving variables.

The pivoting is, in this sense, an efficient scalable procedure for propagating changes: it propagates a change in the basis to the problem variables.

7.1.4 Feasibility-preserving moves

The simplest local move just changes the value of a single variable, and this local move suffices to reach any point in the search space. The drawback is that this move often transforms a feasible assignment into an infeasible one. The penalty function associated with the violated constraints should then force another move to a feasible state. However if several moves are needed to reach another feasible state the penalty will tend to force the second move back to the first assignment.

To avoid the overhead of restoring feasibility by extra local moves, it is often more efficient to design a move operator that transforms feasible assignments to feasible assignments.

In this case the penalty function is doing the job it is designed to do—moving towards solutions with a better cost. A generic local search technique which preserves both feasibility and optimality was described in Pesant and Gendreau (1999). They used a branch-and-bound search over the neighbourhood to prune away infeasible unpromising neighbours and quickly find feasible high-quality moves. The technique was applied to personnel scheduling and vehicle routing and achieved a performance competitive with specialised algorithms. The advantage of this approach is that the problem constraints can be modelled in the usual fashion for mathematical, or constraint programming, and the standard branch-and-bound algorithm automatically delivers the same moves as a tailored neighbourhood operator.

An approach that falls somewhere between constructive and local search has been called "Large Neighbourhood Search" (LNS). The algorithm iteratively finds a solution; it then throws away the part of the solution associated with a large subproblem; and it then employs a branch-and-bound search to extend the remaining partial solution to a complete solution again. This is a very natural algorithm in which operational research can be exploited for the branch-and-bound, while constraint programming looks after the move from a complete solution to the next partial solution. One of the key algorithm design issues that are necessary to make LNS competitive is the choice of large neighbourhood. This issue can also be tackled using constraint programming (Perron et al. 2004).

Because most problems have many constraints, a local move that preserves feasibility with respect to *all* the constraints can be both difficult to design, and inefficient to implement. Worse still, the resulting search space topology may have regions which are disconnected—so there may be no sequence of moves that transform a given assignment into an optimal one. Finally the moves may be so complicated that there is little or no correlation between the costs of neighbouring solutions.

Consequently a compromise is to design local moves that preserve feasibility with respect to a certain constraint, or class of constraints, and allow others to be violated. The classic examples of this are local moves for solving the travelling salesman problem. These include 2-swaps and 3-swaps which map circuits to circuits, and the Lin-Kernighan operator which performs a sub-optimization as part of the local move (Lin and Kernighan 1973).

A more general way to preserve feasibility after a local move is to use a Simplex pivot as a move: this is a form of local move which preserves feasibility for all the linear constraints. The choice of pivot can be controlled by, for example, constraining a specific variable to take a new value. A multi-layer hybrid exploiting pivoting in this way is Kamarainen and El Sakkout (2002). In the next section we discuss the use of pivoting as a local search move on an example application.

7.1.5 Pivoting as a local search move

The pivot is typically used as a local move in Simplex, a hill-climbing algorithm whose local optimum is a globally optimal solution for linear problems. However it can also be used in local search algorithms for non-linear problems.

Pivoting as a local move was applied to a network design problem by Crainic et al. (2000). Given a set of network demands (e.g., a set of required origin-destination flows), the capacitated network design problem is how to design the cheapest network that can meet all the demands. The cost of each potential edge in the network is given: the challenge is to minimize the sum of the costs of the edges that are required.

The problem was modelled using path flow variables: for each demand, with origin O and destination D, and each path from O to D, a variable was introduced to represent the flow for that demand on that path. Constraints were imposed that the sum of the flows matched the demand, and the total flows through each network edge did not exceed the capacity of the edge. This subproblem, ignoring the network edge costs, is linear and so pivoting on the flow variables and constraint slacks, preserves feasibility.

To solve the original network design problem it is only necessary to compute from each subproblem solution which edges are used, and therefore the total network cost.

A key requirement for local search algorithms is that the solution space is "connected", so that from wherever the algorithm starts it can reach an optimum solution by a finite sequence of moves.

We confirm that an optimum solution to the network design problem can indeed be reached by enough pivots from any feasible set of flows. Given an optimum solution, we can reach it by pivoting out all flows through edges which do not belong to this optimum network.

7.1.6 Pivoting for local search in CP

As we have shown throughout the paper, modern CP systems provide a tight integration with a linear constraint solver (supporting both Simplex and barrier method variants). The impact of a pivot operation on the variable in the linear solver are exported to the CP system, and can then wake further invariants to compute the consequences on other variables.

The network design algorithm, described above, has a further set of driven "required-edge" variables that do not go into the linear solver. The CP system can use simple invariants to propagate each non-zero flow to the variables associated with the edges on its path: these variables are set to 1 (or, more directly, to the cost of the edge). Flows which have become zero, are also propagated, using a more complex invariant. In this case a "required-edge" variable is only set to zero if the other flows through that edge are also zero. The efficiency of this invariant can be enhanced by using a further invariant to maintain a count of the number of non-zero flows through that edge.

Finally these changes are propagated to the total cost of the network, via an invariant for sum expressions introduced above.

The search can be controlled from the CP system. The algorithm of Crainic et al. (2000) used a Tabu list to prevent cycling, for example, and this can be handled by invariants that check each change against the current Tabu list. The paper by Michel and Van Hentenryck (2000) provides an overview of how different forms of local search—including Tabu search—is specified quite naturally within a CP framework, and implemented very efficiently.

7.2 Column generation

7.2.1 Overview

Column generation is a very different technique for improving on non-optimal solutions. It is a very useful approach for problems for which an integer-linear model would involve too many variables—in some cases the number of variables grows exponentially with the size of the problem. The purpose is to enable an optimal solution to be found, and proven optimal, without ever considering more than a small proportion of these variables—certainly only a number that grows polynomially with the problem size.

Indeed the algorithm of Crainic et al. (2000) discussed in the previous section used column generation because the number of paths in a network grows much faster than the number of edges. In this algorithm, column generation is employed as a sub-algorithm in a local search algorithm.

More typically column generation is the main algorithm and the hybridisation comes through the choice of the sub-algorithm used to solve the subproblem. In the classic application of column generation to crew scheduling, the sub-algorithm is often a shortest path algorithm on a network whose edge costs are dual prices inherited from the master problem.

7.2.2 Modelling column generation in a CP framework

The constraint programming environment can encapsulate column generation so that it is possible just to plug in the master problem constraints and the subproblem algorithm.

The functionality is similar to that offered by column generation libraries and packages such as Abacus (Junger and Thienel 2000) and BC-Opt (Cordier et al. 1999). Column generation has been used in combination with constraint programming for a range of applications including time-tabling and crew scheduling (Fahle et al. 2002; Moura et al. 2005; Demassey 2005).

As an example of the formulation of column generation in CP consider a simple cutting stock problem. In cutting stock problems the subproblem computes different ways of cutting the raw material so as to optimally meet the total customer demand. We suppose the raw material comes in pieces with length 17. The demand is for 25 pieces with length 3, 20 pieces with length 5 and 15 pieces with length 9.

The subproblem constraints are sent to the CP solver. For example using the constraint #= to express CP equality we can encode the subproblem constraints as

```
3 * A1 + 5 * A2 + 9 * A3 + Waste #=  17
```

The master problem, which is here called `cutstock`, is connected to the subproblem through the built-in expression `implicit_sum` constraints:

```
cutstock: implicit_sum(A1) >= 25
cutstock: implicit_sum(A2) >= 20
cutstock: implicit_sum(A3) >= 15
```

The Master problem cost is expressed as `implicit_sum(Waste).`

Each column returned from a subproblem solution is automatically added to the `implicit_sum` expressions. The master problem is solved by a built-in predicate (either a straight optimization or, if integer solutions are required, a branch-and-price search). The search behaviour has parameters enabling the user to control the search.

In a column generation problem, the master problem has constraints $C_1.V_1 + \cdots C_n.V_n \geq \mathit{RHS}$ in which each coefficient C_i is the value of a solution to the subproblem, and each variable V_i registers whether (and to what extent) the subproblem solution contributes to the overall solution of the master problem.

Before the ith subproblem solution is found, and the ith column is generated, the variable V_i takes the value 0 in all master problem solutions. Until this point, therefore, the above master problem constraint could be written $C_1.V_1 + \cdots C_{i-1}.V_{i-1} + \mathit{Expr} \geq \mathit{RHS}$ where the variable *Expr* represents the (currently unknown) expression $C_i.V_i + \cdots + C_n.V_n$. In all master problem solutions up to this point *Expr* takes the value 0.

In the CP hybrid, the term `implicit_sum(Var)` plays the role of *Expr*. The specific variable *Var* is used to link the expression to the relevant component of the subproblem solution. In this example `implict_sum(A1)` links directly to a subproblem variable `A1`. If in solutions $1, \ldots, i-1$, the variable $A1$ takes values $C_1, \ldots, C_{i-1}$ respectively, then this implicit sum represents the expression $C_1.V_1 + \cdots C_{i-1}.V_{i-1} + \mathit{Expr}$. In our example, of course, the ith solution is a way of cutting the raw material, $A1$ is the number of pieces of length 3 in that cutting, and V_i is the number of times that cutting is used to satisfy the customer requirements.

Each time the master problem is solved, the shadow prices are automatically fed back to the subproblem, and included in its cost function.

The benefit of CP in solving the subproblem is that it is easy to choose the method that best meets the needs of the problem. This point was made by Gendron et al. (2005) where different strategies for solving the subproblem were explored, one which returned new columns with the best potential for improving the master problem cost, and one which focussed on master problem integer feasibility. The conclusion was that in devising subproblem solvers it may be best to use hybrid subproblem models which offer flexibility between the two strategies. CP, with its facility to maintain multiple models connected via channeling constraints is an ideal implementation vehicle for such hybrid methods.

7.2.3 Branch-and-price

The remaining aspect of column generation is the branch-and-price search, in the usual case where the master problem includes integrality constraints.

Branch-and-price augments branch-and-bound by adding new columns to the master problem during search.

Traditionally constraint programming employs depth-first branch-and-bound. Depth-first search is used to control memory consumption. The total memory used is linear in the size of the number of problem variables, in contrast to any kind of best-first or breadth-first search whose memory consumption grows rapidly with the problem size (in theory the memory requirement is exponential in the problem size). With the growth in computer memory due to hardware advances, and with very economic representations of the search state, best-first search is being introduced in constraint programming systems.

A second drawback of this traditional search method in constraint programming is that information inferred at lower levels within the search tree is lost when the search backtracks to higher levels of the tree. This architecture cannot accommodate global cuts, which are inferred at lower levels in the search tree but remain valid for all search states. For the same reason it cannot accommodate column generation where columns added on one branch of a search tree need to be kept and reused on other branches.

In fact any branch-and-bound infers a global cut whenever a new optimum is found, so constraint programming has special mechanisms for handling one kind of global cut—a new bound on the cost variable. Branch-and-price has driven CP search forwards from its traditional depth-first framework to a best-first approach where

1. Multiple open search nodes remain at the search frontier
2. New columns added at one node are made available to other nodes when they are expanded. Other columns are dropped.

An example of the integration of branch and price in CP is described in Puchinger et al. (2008).

8 Conclusion

8.1 Model encapsulation and reuse

The greatest benefit of CP is the support it offers for model encapsulation and reuse. When an OR researcher develops an efficient new model for a class of problems, others wishing to take advantage of this technique are required to read the thesis and implement their own version. In addition, quite often a set of cutting planes devised for a specific problem are not so effective when side constraints are added. Therefore, specific cutting planes should be devised for each problem variant.

Software engineering is about making it easy to "stand on the shoulders of giants". When a good model is developed it can be encapsulated as a (user defined or a global) constraint in CP. The recently introduced modelling language Zinc Marriott et al. (2008) extends OPL to support encapsulation of models through user-defined predicates and functions. Future users have many alternative ways of incorporating this constraint into their design models. The simplest way is to associate a separate solver with each such constraint. In this case the special properties of the model are not lost, but the scope for coordination with other solvers is reduced. The most likely communication is via cost lower bounds and infeasibility-driven cuts. The constraint may be linked in many further ways (for example through standard master/subproblem decompositions, through heuristics fed back to the search engine, through local integration with other user-defined constraints).

CP propagation is particularly adapted to encapsulation and reuse because different propagation constraints cooperate automatically. They achieve this by communicating through the domains of their shared variables and waking each other up by tightening these domains.

The fix-point behaviour, detailed in Sect. 4.1 above, provides a guarantee that the cooperation is, in a well-defined sense, as good as it can possibly be.

The ease with which a new technique can be encapsulated as a constraint and then reused by everyone in the field is responsible for the rapid advance of CP in recent years. Even quite complex techniques such as column generation can be encapsulated and reused by other CP researchers who may never have grappled with the issues of tailing off and repetitive entry of the same columns etc. that previously any user of column generation has had to overcome.

8.2 Open issues

This emerging research field of the integration of OR techniques in CP is extremely promising and motivating since there are many open issues and challenges to be addressed and studied. The first challenge concerns the user support for the problem solving process. The aim is to provide a solver with an abstract conceptual model and let the solver choose the best algorithm to solve it. This is too simple and unrealistic of course, but as a first step, research in this direction should be done in the identification of a mapping between problem structures and a corresponding efficient algorithm. Also, languages and formalisms should be studied to support the easy mapping between models and algorithms, possibly encapsulating these languages in meta-solvers that define the solving strategy. Examples of languages and meta-solver are Arbab and Monfroy (1998), Michel and Van Hentenryck (2000), ILOG Optimization Team (2003), Van Hentenryck (1999), Laburthe and Caseau (2002), Bousonville et al. (2005). Research directions are different: some researchers look for a minimal set of algorithm primitives and operators, powerful enough to express combinatorial problems of any kind. Others tend to embed in the language all possible constraints and their variants so that the resulting system offers all the tools necessary for solving combinatorial problems efficiently.

A second challenge concerns the solution of problems whose data are partially unknown or ill-defined. The problem solving process here should focus not only on searching for the optimal solution, which is often meaningless, but also on its robustness to the unknown parameter changing and to external events. In many cases, an interaction with the user or the environment where the algorithm works is mandatory to acquire problem data while the execution is in progress. This interaction opens a number of issues to be addressed and solved such as dynamic changes and problem requirements modification.

In many cases, the problem is defined on probabilistic data. In this case, stochastic optimization should be taken into account. Many approaches have been studied in the field of Operations Research, like sampling (Ahmed and Shapiro 2002) that uses an approximation of the expected value with its average value over a given sample; the *l-shaped* method (Laporte and Louveaux 1993) which faces two phase problems and is based on Benders Decomposition (Benders 1962). A different method is based on the branch-and-bound extended for dealing with stochastic variables (Norkin et al. 1998). The constraint programming community has recently faced stochastic problems: in Walsh (2002) stochastic constraint programming is introduced formally and the concept of solution is replaced with the one of *policy*. This work has been extended in Tarim et al. (2006) where an algorithm based on the concept of scenarios is proposed. In particular, the paper shows how to reduce the number of scenarios, maintaining a good expressiveness. However, up to now, large scale stochastic problems have never been faced with CP.

A third challenge concerns over-constrained problems, i.e., those problems that have no solution. These problems are quite common in an industrial setting where solutions represent compromises and violate some problem constraints. At the state of the art some approaches have been devised to face these problems: (1) rank constraints into classes of importance, (2) add a penalty to constraints and (3) count the violation. Again large scale over-constrained problems have never been faced in CP using these techniques. In addition, a good inconsistency explanation mechanism is missing in CP, while it is a crucial part of the programming development environment (Deransart et al. 2000).

A fourth challenge concerns the improvement of each aspect of the constraint programming solving process: propagation and search. As far as propagation is concerned, many problem structures have been studied and filtering algorithms have been devised, but there is space for improvement, both for the definition of new structures and new constraints, and for the improvement of existing algorithms in particular on those structures that represent NP-hard problems as it happens, for instance, in scheduling and routing related constraints. As far as search is concerned, tree search lacks of scalability as the dimension of the problem grows. For large scale problem instances complete tree search is not viable. Indeed, the constraint programming community has recognized the importance of combining tree search with local search and meta-heuristics for instance and yet some steps have been done in this perspective by Pesant and Gendreau (1999), Shaw (1998), Focacci et al. (2003) and Lhomme and Jussien (2002).

References

Achterberg, T. (2009). Scip: solving constraint integer programs. *Mathematical Programming Computation*.

Achterberg, T., Berthold, T., Koch, T., & Wolter, K. (2008). Constraint integer programming: a new approach to integrate cp and mip. In *Proceedings of the intl conference on the integration of AI and OR techniques in constraint programming*, 2008, pp. 6–20.

Ahmed, S., & Shapiro, A. (2002). The sample average approximation method for stochastic programs with integer recourse. In *Optimization on line*, 2002.

Applegate, D., Bixby, R., Chvátal, V., & Cook, W. (1999). Tsp-solver concorde. http://www.keck.caam.rice.edu/concorde.html.

Apt, K. R., & Wallace, M. G. (2006). *Constraint logic programming using ECLiPSe*. Cambridge: Cambridge University Press.

Arbab, F., & Monfroy, E. (1998). Coordination of heterogeneous distributed cooperative constraint solving. *Applied Computing Review*, *6*, 4–17. ACM SIGAPP.

Baptiste, P., Le Pape, C., & Nuijten, W. (1995). Efficient operations research algorithms in constraint-based scheduling. In *1st Joint workshop on artificial intelligence and operational research*.

Baptiste, P., Le Pape, C., & Nuijten, W. (2003). *Constraint-based scheduling*. Dordrecht: Kluwer Academic Publisher.

Beldiceanu, N., & Carlsson, M. (2002). A new multi-resource cumulatives constraint with negative heights. In *LNCS: Vol. 2470. Proc. of the Int. Conference on Principles and Practice of Constraint Programming CP 2002* (pp. 63–79). Berlin: Springer.

Beldiceanu, N., Bourreau, E., Chan, P., & Rivreau, D. (1997). Partial search strategy in CHIP. In *Proceedings of the 2nd international conference on meta-heuristics*, 1997.

Beldiceanu, N., Carlsson, M., & Rampon, J.-X. (2005). *Global constraint catalog* (SICS Technical Report T2005:08).

Benders, J. F. (1962). Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, *4*, 238–252.

Benhamou, F., Granvilliers, L., & Goualard, F. (1999). Interval constraints: results and perspectives. In *New Trends in Constraints* (pp. 1–16).

Beringer, H., & De Backer, B. (1995). Combinatorial problem solving in constraint logic programming with cooperating solvers. In C. Beierle & L. Plümer (Eds.), *Logic programming: formal methods and practical applications* (pp. 245–272). Amsterdam: North Holland.

Bèssiere, C., & Régin, J. C. (1997). Arc consistency for general constraint networks: preliminary results. In M. Pollack (Ed.), *Proceedings of the international joint conference on artificial intelligence, IJCAI* (pp. 398–404). San Mateo: Morgan Kaufmann.

Bousonville, T., Focacci, F., Le Pape, C., Nuijten, E., Paulin, F., Puget, J. F., Robert, A., & Sadeghin, A. (2005). Integration of rules and optimization in plant powerops. In R. Bartak & M. Milano (Eds.), *LNCS: Vol. 3524. Proc. of the international conference in the integration of AI and OR techniques in constraint programming—CPAIOR 2005* (pp. 1–15). Berlin: Springer.

Carlier, J., & Pinson, E. (1995). An algorithm for solving job shop scheduling. *Management Science*, *35*, 164–176.

Cheng, B. M. W., Lee, J. H. M., & Wu, J. C. K. (1996). Speeding up constraint propagation by redundant modeling. In E. C. Freuder (Ed.), *LNCS: Vol. 1118. Proc. of the int. conference on principles and practice of constraint programming* (pp. 91–103). Berlin: Springer.

Choi, C. W., Harvey, W., Ho-Man Lee, J., & Stuckey, P. J. (2004). Finite domain bounds consistency revisited. URL www.citebase.org/cgi-bin/citations?id=oai:arXiv.org:cs/0412021.

Codognet, P., & Diaz, D. (1996). Compiling constraints in clp(fd). *Journal of Logic Programming*, *27*, 1–199.

COIN-OR Foundation (2009). COmputational INfrastructure for Operations Research. http://www.coin-or.org/.

Cordier, C., Marchand, H., Laundy, R., & Wolsey, L. A. (1999). BC-Opt: a branchand-cut code for mixed integer programs. *Mathematical Programming*, *86*(2), 335–354.

Crainic, T. G., Gendreau, M., & Farvolden, J. M. (2000). A simplex-based tabu search method for capacitated network design. *INFORMS Journal of Computing*, *12*(3), 223–236.

Davis, E. (1987). Constraint propogation with interval labels. *Artificial Intelligence*, *32*(3), 281–331.

Demassey, S., Pesant, G., & Rousseau, L.-M. (2005). Constraint programming based column generation for employee timetabling. In R. Bartak & M. Milano (Eds.), *LNCS: Vol. 3524. Proc. of the int. conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems—CPAIOR* (pp. 140–154). Berlin: Springer.

Deransart, P., Hermenegildo, M. V., & Maluszynski, J. (2000). In *LNCS: Vol. 1870. Analysis and visualisation tools for constraint programming*. Berlin: Springer.

El Sakkout, H., & Wallace, M. (2000). Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints*, *5*(4), 359–388.

Fahle, T., Junker, U., Karisch, S. E., Kohl, N., Sellmann, M., & Vaaben, B. (2002). Constraint programming based column generation for crew assignment. *Journal of Heuristics*, *8*(1), 59–81.

Focacci, F., Lodi, A., & Milano, M. (1999). Cost-based domain filtering. In J. Jaffar (Ed.), *LNCS: Vol. 1713. Proceedings of the international conference on principles and practice of constraint programming CP'99* (pp. 189–203). Berlin: Springer.

Focacci, F., Laburthe, F., & Lodi, A. (2003). Local search and constraint programming—ls and cp illustrated on a transportation problem. In M. Milano (Ed.), *Constraint and integer programming—toward a unified methodology, Chap. 9.* Dordrecht: Kluwer Academic Publisher.

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

Fruhwirth, T. (1998). Theory and practice of constraint handling rules. *Journal of Logic Programming—Special Issue on Constraint Logic Programming*, 37.

Gendron, B., Lebbah, H., & Pesant, G. (2005). Improving the cooperation between the master problem and the subproblem in constraint programming based column generation. In R. Bartak & M. Milano (Eds.), *LNCS: Vol. 3524. Proc. of the int. conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems—CPAIOR* (pp. 217–227). Berlin: Springer.

Grossmann, I. E., & Jain, V. (2001). Algorithms for hybrid milp/cp models for a class of optimization problems. *INFORMS Journal on Computing*, *13*, 258–276.

Guret, C., Prins, C., & Sevaux, M. (2002). Application of optimization with Xpress MP. In *Dash optimization*. ISBN 0-9543503-0-8.

Harvey, W. D., & Ginsberg, M. L. (1995). Limited discrepancy search. In C. S. Mellish (Ed.), *Proceedings of the fourteenth international joint conference on artificial intelligence (IJCAI-95)* (Vol. 1, pp. 607–615).

Hooker, J. N. (2004). A hybrid method for planning and scheduling. In M. Wallace (Ed.), *LNCS: Vol. 3258. Proc. of the int. conference on principles and practice of constraint programming—CP 2004* (pp. 305–316). Berlin: Springer.

Hooker, J. N. (2005). Planning and scheduling to minimize tardiness. In P. Van Beek (Ed.), *LNCS: Vol. 3709. Proc. of the int. conference on principles and practice of constraint programming—CP 2005* (pp. 314–327). Berlin: Springer.

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

ILOG Optimization Team (2003). Concert technology.

ILOG Optimization Team (2008). Cplex 11.2 user manual.

Junger, M., & Thienel, S. (2000). The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. *Software: Practice and Experience*, *30*, 1325–1352.

Kamarainen, O., & El Sakkout, H. (2002). Local probing applied to scheduling. In P. Van Hentenryck (Ed.), *LNCS: Vol. 2470. Proc. of the int. conference on principles and practice of constraint programming* (pp. 155–171). Berlin: Springer.

Laburthe, F., & Caseau, Y. (2002). Salsa: a language for search algorithms. *Constraints*, *7*(3–4), 255–288.

Lamaréchal, C. (2003). The omnipresence of Lagrange. *4OR*, *1*, 7–25.

Langley, P. (1992). Systematic and nonsystematic search strategies. In J. Hendler (Ed.), *Proceedings of the 1st international conference on AI planning systems, AIPS* (pp. 145–152). San Mateo: Morgan Kaufmann.

Laporte, G., & Louveaux, F. V. (1993). The integer l-shaped method for stochastic integer programs with complete recourse. *Operations Research Letters*, *13*, 133–142.

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

Le Provost, T., & Wallace, M. (1993). Generalized constraint propagation over the clp scheme. *Journal of Logic Programming*, *16*, 319–359.

Lhomme, O. & Jussien, N. (2002). Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, *139*(1), 21–45.

Lin, S., & Kernighan, B. (1973). An efficient heuristic for the Traveling Salesman Problem. *Operations Research*, *21*(2), 498–516.

Marques-Silva, J. P., & Sakallah, K. A. (1999). Grasp-a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, *48*(5), 506–521.

Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P. J., Garcia De La Banda, M., & Wallace, M. (2008). The design of the zinc modelling language. *Constraints*, *13*(3), 229–267.

Michel, L., & Van Hentenryck, P. (2000). Localizer. *Constraints*, *5*(1–2), 43–84.

Milano, M., & Wallace, M. (2005). Integrating operations research in constraint programming. *4OR*, *4*(3), 175–219.

Moura, A. V., Yunes, T. H., & de Souza, C. C. (2005). Hybrid column generation approaches for urban transit crew management problems. *Transportation Science*, *39*(2), 273–288.

Norkin, V. I., Pflug, G. C., & Ruszczynski, A. (1998). A branch and bound method for stochastic global optimization. *Mathematical Programming*, *83*, 407–423.

Nowicki, E., & Smutnicki, C. (1996). A fast taboo search algorithm for the job shop problem. *Management Science*, *42*(6), 797–813.

Ohrimenko, O., Stuckey, P. J., & Codish, M. (2009). Propagation via lazy clause generation. *Constraints*, *14*(3), 357–391.

Ouaja, W., & Richards, B. (2005). Hybrid Lagrangian relaxation for bandwidth-constrained routing: knapsack decomposition. In *Proceedings of 2005 ACM symposium on applied computing* (pp. 383–387).

Perron, L., Shaw, P., & Furnon, V. (2004). Propagation guided large neighborhood search. In M. Wallace (Ed.), *LNCS: Vol. 3258. Proc. of the int. conference on principles and practice of constraint programming CP2004*. (pp. 468–481). Berlin: Springer.

Pesant, G., & Gendreau, M. (1999). A constraint programming framework for local search methods. *Journal of Heuristics*, *5*(3), 255–279.

Puchinger, J., Stuckey, P. J., Wallace, M., & Brand, S. (2008). From high-level model to branch-and-price solution in g12. In *Proceedings of the intl conference on the integration of artificial intelligence and operations research techniques in CP* (pp. 218–232).

Refalo, P. (2000). Linear formulation of constraint programming models and hybrid solvers. In R. Dechter (Ed.), *LNCS: Vol. 1894. Proceedings of the international conference on principle and practice of constraint programming—CP 2000*. Berlin: Springer.

Régin, J. C. (1994). A filtering algorithm for constraints of difference in CSPs. In B. Hayes-Roth & R. Korf (Eds.), *Proceedings of the twelfth national conference on artificial intelligence—AAAI94* (pp. 362–367).

Régin, J. C. (1999). Arc consistency for global cardinality constraints with costs. In J. Jaffar (Ed.), *LNCS: Vol. 1713. Proceedings of the international conference on principles and practice of constraint programming, CP'99* (pp. 390–404). Berlin: Springer.

Regin, J. C. (2004). Global constraints and filtering algorithms. In M. Milano (Ed.), *Constraint and integer programming*. Dordrecht: Kluwer Academic Publisher.

Rodosek, R., Wallace, M., & Hajian, M. T. (1997). A new approach to integrating mixed integer programming and constraint logic programming. *Annals of Operational Research*. Recent Advances in Combinatorial Optimization.

Sannella, M., Maloney, J., Freeman-Benson, B., & Borning, A. (1993). Multi-way versus one-way constraints in user interfaces: experience with the DeltaBlue algorithm. *Software: Practice and Experience*, *23*(5), 529–566.

Schulte, C., & Stuckey, P. J. (2005). When do bounds and domain propagation lead to the same search space. *ACM Transactions on Programming Languages and Systems*, *27*(3), 388–425.

Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. In M. Maher & J. F. Puget (Eds.), *LNCS: Vol. 1520. Proc. of the int. conference on principles and practice of constraint programming CP1998* (pp. 417–431). Berlin: Springer.

Tarim, A., Manandhar, S., & Walsh, T. (2006). Stochastic constraint programming: a scenario-based approach. *Constraints*, *11*, 53–80.

Van Hentenryck, P. (1999). *The OPL optimization programming language*. Cambridge: MIT Press.

Van Hentenryck, P., & Michel, L. (2005). *Constraint-based local search*. Cambridge: MIT Press.

van Hoeve, W. J. (2006). The alldifferent constraint: a systematic overview. URL www.cs.cornell.edu/~vanhoeve/papers/alldiff.pdf.

Walsh, T. (2002). Stochastic constraint programming. In F. van Harmelen (Ed.), *Proc. of the European conference on artificial intelligence, ECAI*. Amsterdam: IOS Press.

Zhou, N. F. (2005). Finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming* (4–5).