Constraint Programming

Chapter · March 2006

DOI: 10.1007/0-387-28356-0_9

Mark G. Wallace
Monash University (Australia)

Chapter 8

CONSTRAINT PROGRAMMING

Eugene C. Freuder
University College Cork, Ireland

Mark Wallace
Monash University, Australia

1. INTRODUCTION

Constraint satisfaction problems are ubiquitous. A simple example that we will use throughout the first half of this chapter is the following scheduling problem: Choose employees A or B for each of three tasks, X, Y, Z, subject to the work rules that the same employee cannot carry out both tasks X and Y, the same employee cannot carry out both tasks Y and Z, and only employee B is allowed to carry out task Z. (Many readers will recognize this as a simple *coloring problem.*)

Affiliations OK?

Title differs from kluwer's 'constraint based reasoning'. Which is correct?

This is an example of a class of problems known as constraint satisfaction problems (CSPs). CSPs consist of a set of *variables* (e.g. tasks), a *domain* of *values* (e.g. employees) for each variable, and *constraints* (e.g. work rules) among sets of variables. The constraints specify which combinations of value assignments are allowed (e.g. employee A for task X and employee B for task Y); these allowed combinations *satisfy* the constraints. A *solution* is an assignment of values to each variable such that all the constraints are satisfied (Dechter, 2003; Tsang, 1993).

We stress that the basic CSP paradigm can be *extended* in many directions: for example, variables can be added dynamically, domains of values can be continuous, constraints can have priorities, and solutions can be *optimal*, not merely satisfactory.

Some examples of constraints are

- The meeting must start at 6:30.
- The separation between the soldermasks and nets should be at least 0.15 mm.

- This model only comes in blue and green.
- This cable will not handle that much traffic.
- These sequences should align optimally.
- John prefers not to work on weekends.
- The demand will probably be for more than 5 thousand units in August.

Some examples of constraint satisfaction or optimization problems are

- Schedule these employees to cover all the shifts.
- Optimize the productivity of this manufacturing process.
- Configure this product to meet my needs.
- Find any violations of these design criteria.
- Optimize the use of this satellite camera.
- Align these amino acid sequences.

Many application domains (e.g. design) naturally lend themselves to modeling as CSPs. Many forms of reasoning (e.g. temporal reasoning) can be viewed as constraint reasoning. Many disciplines (e.g. operations research) have been brought to bear on these problems. Many computational "architectures" (e.g. neural networks) have been utilized for these problems. Constraint programming can solve problems in telecommunications, internet commerce, electronics, bioinformatics, transportation, network management, supply chain management, and many other fields.

Here are just a few examples of commercial application of constraint technology:

- Staff planning: BanqueBuxelles Lambert.
- Vehicle production optimization: Chrysler Corporation.
- Planning medical appointments: FREMAP.
- Task scheduling: Optichrome Computer Systems.
- Resource allocation: SNCF (French Railways).
- From push to pull manufacturing: Whirlpool.
- Utility service optimization: Long Island Lighting Company.

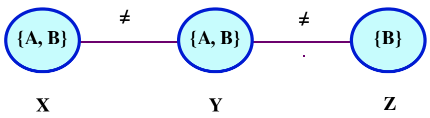


Figure 8.1. A constraint network representation of a sample constraint satisfaction problem.

- Intelligent cabling of big buildings: France Telecom.
- Financial decision support system: Caisse des Dépôts.
- Load capacity constraint regulation: Eurocontrol.
- Planning of satellites missions: Alcatel Espace.
- Optimization of configuration of telecom equipment: Alcatel CIT.
- Production scheduling of herbicides: Monsanto.
- "Just in Time" transport and logistics in Food industry: Sun Valley.
- Supply Chain Management in Petroleum Industry: ERG Petroli.

CSPs can be represented as *constraint networks*, where the variables correspond to nodes and the constraints to arcs. The constraint network for our sample problem appears below. Constraints involving more than two variables can be modeled with hypergraphs, but most basic CSP concepts can be introduced with binary constraints involving two variables, and that is the route we will begin with in this chapter. We will say that a value for one variable is *consistent with* a value for another if the pair of values satisfies the binary constraint between them. (This constraint could be the trivial constraint that allows all pairs of values; such constraints are not represented by arcs in the constraint network.) Note that specifying a domain of values for a variable can be viewed as providing a unary constraint on that single variable.

This chapter will focus on the methods developed in artificial intelligence and the approaches embodied in constraint programming languages. Of course, this brief chapter can only suggest some of the developments in these fields; it is not intended as a survey, only as an introduction. Rather than beginning with formal definitions, algorithms, and theorems, we will focus on introducing concepts through examples.

The constraint programming ideal is this: the programming is declarative; we simply state the problem as a CSP and powerful algorithms,

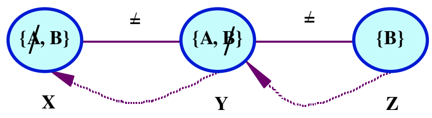


Figure 8.2. Arc consistency propagation.

provided by a constraint library or language, solve the problem. In practice, this ideal has, of course, been only partially realized, and expert constraint programmers are needed to refine modeling and solving methods for difficult problems.

2. INFERENCE

Inference methods make implicit constraint information explicit. Inference can reduce the effort involved in searching for solutions or even synthesize solutions without search. The most common form of inference is known as *arc consistency*. In our sample problem, we can infer that B is not a possible value for Y because there is *no* value for Z that, together with B, satisfies the constraint between Y and Z. This can be viewed as making explicit the fact that the unary constraint on the variable Y does not allow B.

This inference process can *propagate*: after deleting B from the domain of Y, there is no value remaining for Y that together with A for X will satisfy the constraint between X and Y, therefore we can delete A from the domain of X. (See Fig. 8.2.) If we repeatedly eliminate inconsistent values in this fashion until any value for any variable is consistent with some value for all other variables, we have achieved arc consistency. Many algorithms have been developed to achieve arc consistency efficiently (Bessière et al., 1999; Macworth, 1977).

Eliminating inconsistent values by achieving arc consistency can greatly reduce the space we must search through for a solution. Arc consistency methods can also be interleaved with search to dynamically reduce the search space, as we shall see in the next section.

Beyond arc consistency lies a broad taxonomy of consistency methods. Many of these can be viewed as some form of (i,j)-*consistency*. A CSP is (i,j)-consistent if, given any consistent set of i values for i variables, we can find j values for any other j variables, such that the $i+j$ values together satisfy all the constraints on the $i+j$ variables. Arc consistency is (1,1)-consistency. ($k-1$,1)-consistency, or

k-consistency, for successive values of k constitutes an important constraint hierarchy (Freuder, 1978).

More advanced forms of consistency processing often prove impractical either because of the processing time involved or because of the space requirements. For example, 3-consistency, otherwise known as *path consistency*, is elegant because it can be shown to ensure that given values for any two variables one can find values that satisfy all the constraints forming any given path between these variables in the constraint network. However, achieving path consistency means making implicit binary constraint information explicit, and storing this information can become too costly for large problems.

For this reason variations on *inverse consistency*, or $(1,j-1)$-consistency, which can be achieved simply by domain reductions, have attracted some interest (Debruyne and Bessière, 2001). Various forms of *learning* achieve *partial* k-consistency during search (Dechter, 1990). For example, if we modified our sample problem to allow only A for Z, and we tried assigning B to X and A to Y during a search for a solution to this problem, we would run into a "dead end": no value would be possible for Z. From that we could learn that the constraint between X and Y should be extended to rule out the pair (B, A), achieving partial path consistency.

Interchangeability provides another form of inference, which can also eliminate values from consideration. Suppose we modify our sample problem to add employees C and D who can carry out task X. Values C and D would be interchangeable for variable X because in any solution using one we can substitute the other. Thus we can eliminate one in our search for solutions (and if we want to, just substitute it back into any solutions we find). Just as with consistency processing there is a local form of interchangeability that can be efficiently computed. In a sense, inconsistency is an extreme form of interchangeability; all inconsistent values are interchangeable in the null set of solutions that utilize them (Freuder, 1991).

3. MODELING

Modeling is a critical aspect of constraint satisfaction. Given a user's understanding of a problem, we must determine how to model the problem as a constraint satisfaction problem. Some models may be better suited for efficient solution than others (Régin, 2001).

Experienced constraint programmers may add constraints that are *redundant* in the sense that they do not change the set of solutions to the problem, in the hope that adding these constraints may still be cost-effective in terms of reducing problem solving effort. Added constraints that do eliminate some, but not all, of the solutions, may also be useful: for example, to break symmetries in the problem.

Specialized constraints can facilitate the process of modeling problems as CSPs, and associated specialized inference methods can again be cost-effective. For example, imagine that we have a problem with four tasks, two employees who can handle each, but three of these tasks must be undertaken simultaneously. This temporal constraint can be modeled by three separate binary inequality constraints between each pair of these tasks; arc consistency processing of these constraints will not eliminate any values from their domains. On the other hand an "all-different" constraint, that can apply to more than two variables at a time, not only simplifies the modeling of the problem, but an associated inference method can eliminate all the values from a variable domain, proving the problem unsolvable. Specialized constraints may be identified for specific problem domains: for example, scheduling problems.

It has even proven useful to maintain multiple complete models for a problem "channeling" the results of constraint processing between the two (Cheng et al., 1999). As has been noted, a variety of approaches have been brought to bear on constraint satisfaction, and it may prove useful to model part of a problem as, for example, an integer programming problem. Insight is emerging into basic modeling issues: for example, binary versus non-binary models (Bacchus et al., 2002).

In practice, modeling can be an iterative process. Users may discover that their original specification of the problem was incomplete or incorrect or simply impossible. The problems themselves may change over time.

4. SEARCH

In order to find solutions we generally need to conduct some form of search. One family of search algorithms attempts to build a solution by *extending* a set of consistent values for a subset of the problem variables, repeatedly adding a consistent value for one more variable, until a complete solution is reached. Another family of algorithms attempts to find a solution by *repairing* an inconsistent set of values for all the variables, repeatedly changing an inconsistent value for one variable, until a complete solution is reached. (Extension and repair techniques can also be combined.)

Often extension methods are systematic and *complete*, they will eventually try all possibilities, and thus find a solution or determine unsolvability, while often repair methods are stochastic and incomplete. The hope is that completeness can be traded off for efficiency.

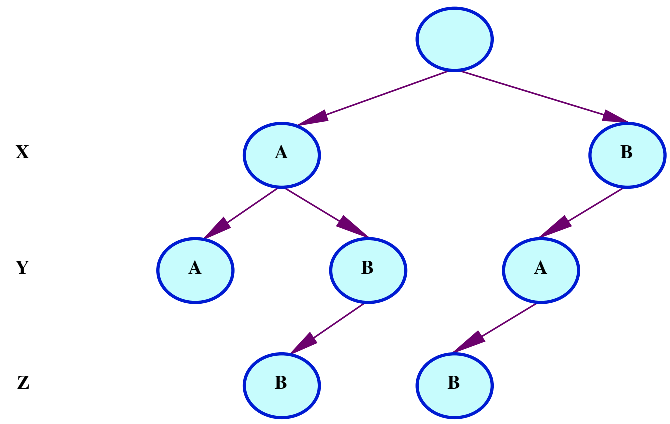


Figure 8.3. Backtrack search tree for example problem.

4.1 Extension

The classic extension algorithm is *backtrack search.* Figure 8.3 shows a backtrack search tree representing a trace of a backtrack algorithm solving our sample problem.

A depth-first traversal of this tree corresponds to the order in which the algorithm tried to fit values into a solution. First the algorithm chose to try A for X, then A for Y. At this point it recognized that the choice of A for Y was inconsistent with the choice of A for X: it failed to satisfy the constraint between X and Y. Thus there was no need to try a choice for Z; instead the choice for Y was changed to B. But then B for Z was found to be inconsistent, and no other choice was available, so the algorithm "backed up" to look for another choice for Y. None was available so it backed up to try B for X. This could be extended to A for Y and finally to B for Z, completing the search.

Backtrack search can prune away many potential combinations of values simply by recognizing when an assignment of values to a subset of the variables is already inconsistent and cannot be extended. However, backtrack search is still prone to "thrashing behavior". A "wrong decision" early on can require an enormous amount of "backing and filling" before it is corrected. Imagine, for example, that there were 100 other variables in our example problem, and, after initially choosing A for X and B for Y, the search algorithm tried assigning consistent values to each of those 100 variables before looking at Z. When it proved impossible to find a consistent value for Z (assuming the search was able to get that far successfully) the algorithm would begin trying different combinations of values for all those 100 variables, all in vain.

A variety of modifications to backtrack search address this problem (Kondrak and van Beek, 1997). They all come with their own overhead, but the search effort savings can make the overhead worthwhile.

Heuristics can guide the search order. For example, the "minimal domain size" heuristic suggests that as we attempt to extend a partial solution we consider the variables in order of increasing domain size; the motivation there is that we are more likely to fail with fewer values to choose from, and it is better to encounter failure higher in the search tree than lower down when it can induce more thrashing behavior. Using this heuristic in our example we would have first chosen B for Z, then proceeded to a solution without having to back up to a prior level in the search tree. While "fail first" makes sense for the order in which to consider the variables, "succeed first" makes sense for the order in which to try the values for the variables.

Various forms of inference can be used prospectively to prune the search space. For example, search choices can be interleaved with arc consistency maintenance. In our example, if we tried to restore arc consistency after choosing A for X, we would eliminate B from the domain of Z, leaving it empty. At this point we would know that A for X was doomed to failure and could immediately move on to B. Even when failure is not immediate, "look ahead" methods that infer implications of search choices can prune the remaining search space. Furthermore, "dynamic" search order heuristics can be informed by this pruning: for example, the minimal domain size heuristic can be based on the size of the domains after look-ahead pruning. Maintaining arc consistency is an extremely effective and widely used technique (Sabin and Freuder, 1997).

Memory can direct various forms of "intelligent backtracking" (Dechter and Frost, 2002). For example, suppose in our example for some reason our search heuristics directed us to start the search by choosing B for Y followed by A for X. Of course B the only choice for Z would then fail. Basic backtrack search would back up "chronologically" to then try B for X. However, if the algorithm "remembers" that failure to find a value for Z was based solely on conflict with the choice for Y, it can "jump back" to try the alternative value A at the Y level in the search tree without unnecessarily trying B for X. The benefits of maintaining arc consistency overlap with those of intelligent backtracking, and the former may make the latter unnecessary.

Search can also be reorganized to try alternatives in a top-down as opposed to bottom-up manner. This responds to the observation that heuristic choices made early in the extension process, when the remaining search space is unconstrained by the implications of many previous choices, may be most prone to failure. For example, “limited discrepancy search” iteratively restarts the search process increasing the number of “discrepancies”, or deviations from heuristic advice, that are allowed, until a solution is found (Harvey and Ginsberg, 1995). (The search effort at the final discrepancy level dominates the upper bound complexity computation, so the redundant search effort is not as significant as it might seem.)

Extensional methods can be used in an incomplete manner. As a simple example, “random restart”, starting the search over as soon as a dead end is reached, with a stochastic element to the search order, can be surprisingly successful (Gomes et al., 1997).

4.2 Repair

Repair methods start with a complete assignment of values to variables, and work by changing the value assigned to a variable in order to improve the solution. Each such change is called a *move*, and the new assignment is termed a *neighbor* of the previous assignment. Genetic algorithms, which create a new assignment by combining two previous assignments, rather than by moving to a neighbor of a single assignment, can be viewed as a form of repair.

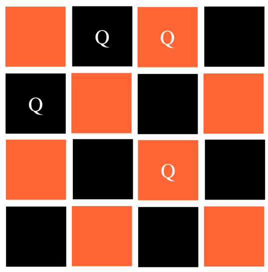
Repair methods utilize a variety of metaphors, physical (hill climbing, simulated annealing) and biological (neural networks, genetic algorithms). For example, we might start a search on our example problem by choosing value A for each variable. Then, seeking to “hill climb” in the search space to an assignment with fewer inconsistencies, we might choose to change the value of Y to B; and we would be done. Hill climbing, is a repair-based algorithm in which each move is required to yield a neighbor with a better cost than before. It cannot, in general, guarantee to produce an optimal solution at the point where the algorithm stops because no neighbor has a better cost than the current assignment.

Repair methods can also use heuristics to guide the search. For example, the *min-conflicts* heuristic suggests finding an inconsistent value and then changing it to the alternative value that minimizes the amount of inconsistency remaining (Minton et al., 1992).

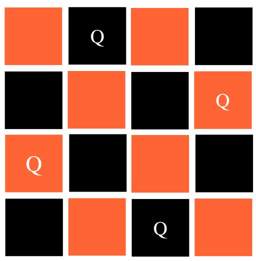
The classic repair process risks getting “stuck” at a “local maximum”, where complete consistency has not been achieved, but any single change will only increase inconsistency, or “cycling” through the same set of inconsistent assignments. There are many schemes to cope. A stochastic element can be helpful. When an algorithm has to choose between equally desirable alternatives it may do so randomly. When no good alternative exists it may start over, or "jump" to a new starting point. Simulated annealing allows moves to neighbors with a worse cost with a given probability. Memory can also be utilized to guide the search and avoid cycling (tabu search).

5. EXAMPLE

We illustrate simple modeling, search and inference now with another example. The Queens Problem involves placing queens on a chessboard such that they do not attack one another. A simple version only uses a four-by-four corner of the chessboard to place four queens:



Queens in chess attack horizontally, vertically, and diagonally. So, for example, the two queens on the dark squares above attack each other diagonally, the two queens on the light squares attack vertically. One solution is



If we model this problem as a CSP where the variables are the four queens and the values for each queen are the 16 squares, we have 65 536 possible combinations to explore, looking for one where the constraints (the queens do not attack each other) are satisfied. If we observe that we can only have one queen per row, and model the problem with a variable corresponding to the queen in each row, each variable having four possible values corresponding to the squares in the row, we have only 256 possibilities to search through.

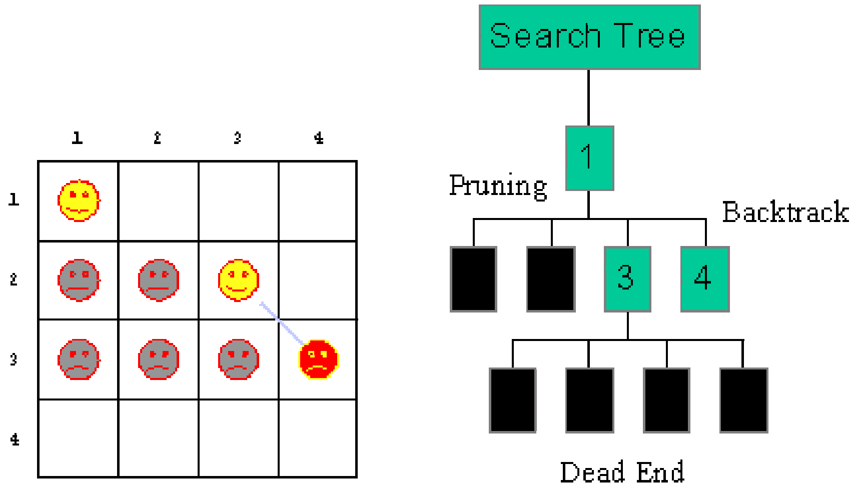


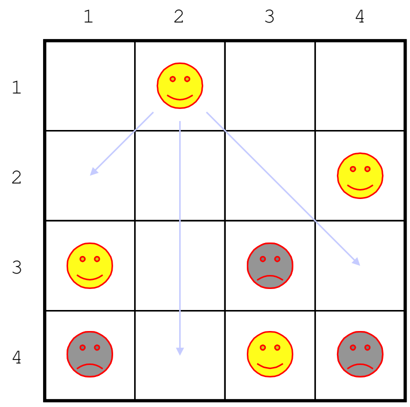
Figure 8.4. The beginning of the example backtrack search tree.

The beginning of the backtrack search tree for this example is shown in Fig. 8.4. After placing the first row queen in the first column, the first successful spot for the second row queen is in the third column. However, that leaves no successful placement for the third row queen, and we need to backtrack.

In fact, there will be quite a lot of backtracking to do here before we find a solution. However, arc consistency inference can reduce the amount of search we do considerably. Consider what happens if we seek arc consistency after placing a queen in the second column of row 1. This placement directly rules out any square that can be attacked by this queen, of course, and, in fact, arc consistency propagation proceeds to rule out additional possibilities until we are left with a solution. The queens in column 3 of row 3 and column 4 of row 4 are ruled out because they are attacked by the only possibility left for row 2. After the queen in column 3 of row 3 is eliminated, the queen in column 1 of row 4 is attacked by the only remaining possibility for row 3, so it too can be eliminated.

6. TRACTABILITY

CSPs are in general NP-hard. Analytical and experimental progress has been made in characterizing tractable and intractable problems. The results have been used to inform algorithmic and heuristic methods.



6.1 Theory

Tractable classes of CSPs have been identified based on the structure of the constraint network, e.g. tree structure, and on closure properties of sets of constraints (Jeavons et al., 1997), e.g. max-closed. Tractability has been associated with sets of constraints defining a specific class of problems, e.g. temporal reasoning problems defined by "simple temporal networks" (Dechter et al., 1991).

If a constraint network is tree-structured, there will be a *width-one* ordering for the variables in which each variable is directly constrained by at most one variable earlier in the ordering. In our sample problem, which has a trivial tree structure, the ordering X, Y, Z is width-one: Y is constrained by X and Z by Y; the ordering X, Z, Y is not width-one: Y is constrained by both X and Z. If we achieve arc consistency and use a width-one ordering as the order in which we consider variables when trying to extend a partial solution, backtrack search will in fact be *backtrack-free*: for each variable we will be able to find a consistent value without backing up to reconsider a previously instantiated variable (Freuder, 1982).

Max-closure means that if $(a\ b)$ and $(c\ d)$ both satisfy the constraint, then $((\max(a\ c)), (\max(b\ d)))$ will also satisfy the constraint. If all the constraints in a problem are max-closed, the problem will be tractable. The "less than" constraint is max-closed, e.g. $4 < 6$, $2 < 9$ and $4 < 9$. Thus if we replaced the inequality constraints in our sample problem by less-than constraints, we would ensure tractability, even if we gave up the tree structure by adding a constraint between X and Z. In fact simple temporal networks are max-closed.

6.2 Experiment

Intuitively it seems natural that many random CSPs would be relatively easy: loosely constrained problems would be easy to solve, highly constrained problems would be easy to prove unsolvable. What is more surprising is the experimental evidence that as we vary the constrainedness of the problems there is a sharp *phase transition* between solvable and unsolvable regions, which corresponds to a sharp spike of "really hard" problems (Cheeseman et al., 1991). ("Phase transition" is a metaphor for physical transitions, such as the one between water and ice.)

7. OPTIMIZATION

Optimization arises in a variety of contexts. If all the constraints in a problem cannot be satisfied, we can seek the "best" partial solution. If there are many solutions to a problem, we may have some criteria for distinguishing the "best" one. "Soft constraints" can allow us to express probabilities or preferences that make one solution "better" than another.

Again we face issues of modeling, inference and search. What does it mean to be the "best" solution; how do we find the "best" solution? We can assign scores to local elements of our model and combine those to obtain a global score for a proposed solution, then compare these to obtain an optimal solution for the problem.

A simple case is the *Max-CSP problem*, where we seek a solution that satisfies as many constraints as possible. Here each satisfied constraint scores 1, the score for a proposed solution is the sum of the satisfied constraints, and an optimal solution is one with a maximal score. However, there are many alternatives, such as fuzzy, probabilistic and possibilistic CSPs, many of which have been captured under the general framework of semiring-based or valued CSPs (Bistarelli et al., 1992).

Backtrack search methods can be generalized to branch and bound methods (see Chapter 1) for seeking optimal solutions, where a partial solution is abandoned when it is clear that it cannot be extended to a better solution than one already found. Inference methods can be generalized. Repair methods can often find close to optimal solutions quickly.

chapter no.?

8. ALGORITHMS

8.1 Handling Constraints

Constraint technology is an approach that solves problems by reasoning on the constraints, and when no further inferences can be drawn, making a search step. Thus inference and search are interleaved.

For problems involving hard constraints—ones that must be satisfied in any solution to the problem—reasoning on the constraints is a very powerful technique.

The secret of much of the success of constraint technology comes from its facility to capitalize on the structure of the problem constraints. This enables "global" reasoning to be supported, which can guide a more "intelligent" search procedure than would otherwise be possible.

In this section we shall introduce some different forms of reasoning and its use in solving problems efficiently

8.2 Domains, and Constraint Propagation

In general domain constraint propagation algorithms take a set of variables and the original domains as input, and either report inconsistency, or output smaller domains for the variables. Since propagation algorithms can extract more information, each time the input domains become smaller, and since the propagation behavior also makes the domains of the variables smaller, the propagation algorithms can co-operate through these domains. The output from one algorithm is input to another, whose output can in turn be input to the first algorithm. Thus many different propagation algorithms can co-operate, together yielding domain reductions which are much stronger than simply pooling the information from the separate algorithms. Consider, for example, the following problem:

```
?- X::1..10, Y::1..10,
fd:(X>Y), fd:(Y>X)
```

If the propagation algorithm for each constraint makes the bounds consistent, then the first constraint will yield new domains

```
X::2..10, Y::1..9
```

and the second constraint will yield new domains:

```
X::1..9, Y::2..10
```

Pooling the deduced information, we get the intersection of the new domains, which is

```
X::2..9, Y::2..9
```

By contrast, if the propagation algorithms communicate through the variable domains, then they will yield new domains, which are then input to the other algorithm until, at the fifth step, the inconsistency between the two constraints is detected.

The interaction of the different algorithms is predictable, even though the algorithms are completely independent, so long as they have certain natural properties. Specifically,

- the output domains must be a subset of the input domains
- if with input domains ID the algorithm produces output domains OD, then with any input domains which are a subset of ID, the output domains must be a subset of OD.

These properties guarantee that the information produced by the propagation algorithms, assuming they are executed until no new information can be deduced, is guaranteed to be the same, independent of the order in which the different algorithms (or *propagation steps*) are executed.

8.3 Constraints and Search

Separating Constraint Handling from Search The constraint programming paradigm supports clarity, correctness and maintainability by separating the statement of the problem as far as possible from the details of the algorithm used to solve it. The problem is stated in terms of its decision variables, the constraints on those variables, and an expression to be optimized.

As a toy example the employee task problem introduced at the beginning of this chapter can be expressed as follows:

```
?- X::[a,b], Y::[a,b], Z::[b], % set up variables
X=\=Y, Y=\=Z % set up constraints
```

This states that the variable `X` can take either of the two values `a` or `b`; similarly Y can take `a` or `b`; but `Z` can only take the value `b`. Additionally, the value taken by `X` must be different from that taken by `Y`; and the values taken by `Y` and `Z` must also be different.

To map this problem statement into an algorithm, the developer must

- choose how to handle each constraint
- specify the search procedure.

For example, assuming **fd** is a solver which the developer chooses to handle the constraints, and **labeling** is a search routine, the whole program is written as follows:

```
?- X::[a,b], Y::[a,b], Z::[b],
fd:(X=$\backslash $=Y), fd:(Y=$\backslash $=Z),
labeling([X,Y,Z])
```

This code sends the constraint $\mathtt{X} = \backslash = \mathtt{Y}$ to the finite domain solver called `fd`; and the constraint $\mathtt{Y} = \backslash = \mathtt{Z}$ is also sent to `fd`. The list of variables `X`, `Y` and `Z` is then passed to the `labeling` routine.

Domain propagation algorithms take as input the domains of the variables, and yield smaller domains. For example, given that the domain of `Z` is [`b`], the `fd` solver for the constraint $\mathtt{Y} = \backslash = \mathtt{Z}$ immediately removes `b` from the domain of `Y`, yielding a new domain [`a`]. Every time the domain of one of those variables is reduced by another propagation algorithm, the algorithm "wakes" and reduces the domains further (possibly waking the other algorithm). In the above example, when the domain of `Y` is reduced to [`a`], the constraint $\mathtt{X} = \backslash = \mathtt{Y}$ wakes, and removes `b` from the domain of `X`, reducing the domain of `X` to [`b`].

The domain of a variable may be reduced either by propagation, or instead by a search decision. In this case propagation starts as before, and continues until no further information can be derived. Thus search and constraint reasoning are automatically interleaved.

Search Heuristics Exploiting Constraint Propagation Most real applications cannot be solved to optimality because they are simply too large and complex. In such cases it is crucial that the algorithm is directed towards areas of the search space where low-cost feasible solutions are most likely to be found.

Constraint propagation can yield very useful information, which can be used to guide the search for a solution. Not only can propagation exclude impossible choices a priori, but it can also yield information about which choices would be optimal in the absence of certain (awkward) constraints.

Because constraint propagation occurs after each search step, the resulting information is used to support *dynamic* heuristics, where the next choice is contingent upon all the information about the problem gathered during the current search.

In short, incomplete extension search techniques can produce high quality solutions to large complex industrial applications in areas such as transportation and logistics. The advantage is that the solutions respect all the hard constraints and are therefore applicable in practice.

8.4 Global Constraints

In this section we introduce a variety of application-specific "global" constraints. These constraints achieve more, and more efficient, propagation behavior than would be possible using combinations of the standard equality and disequality constraints introduced above. We first outline two global constraints, one called `alldifferent` for handling sets of disequality constraints, and one called `schedule` for handling resource usage by a set of tasks.

Alldifferent Consider the disequality constraint "=\=" used in the employee task example at the beginning of this chapter. Perhaps surprisingly, global reasoning on a set of such constraints brings more than local reasoning. Suppose we have a list and want to make local changes until all the elements of the list are distinct. If each element has the same domain (i.e. the same set of possible values), then it suffices to choose any element in conflict (i.e. an element whose value occurs elsewhere in the list), and change it to a new value which does not occur elsewhere. If, however, the domains of different elements of the list are different, then there is no guarantee that local improvement will converge to a list whose elements are all different. However, there is a polynomial time graph-based algorithm (Régin, 1994) which guarantees to detect if there is no solution to this problem, and otherwise it reduces the domains of all the elements until they contain only values that participate in a solution. The constraint that all the elements of a list must be distinct is usually called `alldifferent` in constraint programming. For example, consider the case

```
?- X::[a,b], Y::[a,b], Z::[a,b,c],
alldifferent([X,Y,Z])
```

In this case Régin's algorithm (Régin, 1994) for the `alldifferent` constraint will reduce the domain of `Z` to just [c]. As we saw above, the behavior of reducing the variables domains in this manner is called *constraint propagation.*

Global constraint propagation algorithms work co-operatively within a constraint programming framework. In the above example, the propagation algorithm for the `alldifferent` constraint takes as input the list of variables, and their original domains. As a result it outputs new, smaller, domains for those variables.

Schedule Consider a task scheduling problem comprising a set of tasks with release times (i.e. earliest start times) and due dates (i.e. latest end times), each of which requires a certain amount of resource, running on a set of machines that provide a fixed amount of resource.

The `schedule` constraint works, in principle, by examining each time period within the time horizon:

- First the algorithm calculates how much resource r_i each task i must necessarily take up within this period.
- If the sum $\sum r_i$ exceeds the available resource, then the constraint reports an inconsistency.
- If the sum $\sum r_i$ takes up so much resource that task j cannot be scheduled within the period, and the remaining resource within the period is r_T, then task j is constrained only to use an amount r_T of resource within this period.

For non-preemptive scheduling, this constraint may force such a task j to start a certain amount of time before the period begins, or end after it. The information propagated narrows the bounds on the start times of certain tasks.

Whilst this kind of reasoning is expensive to perform, there are many quicker, but theoretically less complete forms of reasoning, such as "edge-finding", which can be implemented in a time quadratic in the number of tasks.

Further Global Constraints The different global constraints outlined above have proven themselves in practice. Using global constraints such as schedule, constraint programming solves benchmark problems in times competitive with the best complete techniques. The main advantages of global constraints in constraint programming, in addition to their efficiency in solving standard benchmark problems, are that:

- They can be augmented by any number of application-specific "side" constraints. The constraint programming framework allows all kinds of constraints to be thrown in, without requiring any change to the algorithms handling the different constraints.
- They return high-quality information about the problem which can be used to focus the search. Not only do they work with complete search algorithms, but also they guide incomplete algorithms to return good solutions quickly.

Constraint programming systems can include a range of propagation algorithms supporting global reasoning for constraints appearing in different kinds of applications such as rostering, transportation, network optimization, and even bioinformatics.

Analysis One of the most important requirements of a programming system is support for reusability. Many complex models developed by Operations Researchers have made very little practical impact, because they are so hard to reuse. The concept of a global constraint is inherently quite simple. It is a constraint that captures a class of subproblems, with any number of variables. Global constraints have built-in algorithms, which are specialized for treating the problem class. Any new algorithm can be easily captured as a global constraint and reused. Global constraints have had a major impact, and are used widely and often as a tool in solving complex real-world problems. They are, arguably, the most important contribution that constraint programming can bring to Operations Research.

8.5 Different Constraint Behaviors

Constraint reasoning may derive other information than domain reductions. Indeed, any valid inference step can be made by a propagation algorithm. For example, from the constraints $\mathtt{X} > \mathtt{Y}, \mathtt{Y} > \mathtt{Z}$ propagation can derive that $\mathtt{X} > \mathtt{Z}$. To achieve co-operation between propagation algorithms, and termination of propagation sequences, constraint programming systems typically require propagation algorithms to behave in certain standard ways. Normally they are required to produce information of a certain kind: for example, domain reductions.

An alternative to propagating domain reductions is to propagate new linear constraints. Just as domain propagation ideally yields the smallest domains which include all values that could satisfy the constraint, so linear propagation ideally yields the convex hull of the set of solutions. In this ideal case, linear propagation is stronger than domain propagation, because the convex hull of the set of solutions is contained in the smallest set of variable domains (termed the *box*) that contains them.

8.6 Extension and Repair Search

Extension search is conservative in that, at every node of the search tree, all the problem constraints are satisfied. Repair search is optimistic, in the sense that a variable assignment at a search node may be, albeit promising, actually inconsistent with one or more problem constraints.

Constraint Reasoning and Extension Search Constraint reasoning, in the context of extension search, corresponds to logical deduction. The domain reductions, or new linear constraints yielded by propagation, are indeed a consequence of the constraint in the input state.

Constraint Reasoning and Repair Search Constraint reasoning can also be applied in the context of repair search. In this case the constraint behavior is typically caused by the constraint being violated in the input state. Like propagation, the behavior yields new information, which is of a standard simple form that can be dealt with actively by the search procedure. We distinguish two forms of behavior: constraint generation and separation. These forms are best illustrated by example.

- *Constraint generation* For an example of constraint generation, consider a travelling salesman problem which is being solved by integer/linear programming. At each search node an integer/linear problem is solved, which only approximates the actual TSP constraint. Consider a search node which represents a route with a detached cycle. This violates the TSP constraint in a way that can be fixed by adding a linear constraint enforcing a unit flow out from the set of "cities" in the detached cycle. This is the *generated* constraint, at the given search node. The search is complete at the first node where the TSP constraint is no longer violated. Constraint generation can be used in case the awkward constraints can be expressed by a conjunction of easy constraints, although the number of such easy constraints in the conjunction may be too large for them all to be imposed.

- *Separation* Separation behavior is required to fix any violated constraint which cannot be expressed as a conjunction of easy constraints (however large). If the awkward constraint can be approximated, arbitrarily closely, by a (conjunction of) disjunction(s) of easy constraints, then separation can be used. Constraint reasoning yields one of the easy constraints—one that is violated by the current search node—and imposes it so that the algorithm which produces the next search node is guaranteed to satisfy this easy constraint. Completeness is maintained by imposing the other easy constraints in the disjunction on other branches of the search tree.

Languages and Systems One drawback of the logical basis is that repair-based search methods have not fitted naturally into the CLP paradigm. Recently, a language has been introduced called Localizer (Michel and Van Hentenryck, 1999) which is designed specifically to support the encoding of repair-based search algorithms such as Simulated Annealing and GSAT (Selman et al., 1992). The fundamental contribution of Localizer is the concept of an "invariant", which is a constraint that retains information used during search. For GSAT, by way of example, an invariant is used to record, for each problem variable, the change in the number of satisfied propositions if the variable's value were to be changed. The invariant is specified as a constraint, but maintained by an efficient incremental algorithm. Other constraint-based languages for specifying search are SALSA (Laburthe and Caseau, 1998) and ToOLS (de Givry and Jeannin, 2003).

9. CONSTRAINT LANGUAGES

9.1 Constraint Logic Programming

The earliest constraint programming languages, such as *Ref-Arf* and *Alice*, were specialized to a particular class of algorithms. The first general purpose constraint programming languages were constraint handling systems embedded in logic programming (Jaffar and Lassez, 1987; Van Hentenryck et al., 1999), called *constraint logic programming* (CLP). Examples are CLP(fd), HAL, SICStus and ECLiPSe. Certainly logic programming is an ideal host programming paradigm for constraints, and CLP systems are widely used in industry and academia.

Logic programming is based on relations. In fact every procedure in a logic program can be read as a relation. However, the definition of a constraint is exactly the same thing—a *relation*. Consequently, the extension of logic programming to CLP is entirely natural. Logic programming also has backtrack search built in, and this is easily modified to accommodate constraint propagation. CLP has been enhanced with some high-level control primitives, allowing active constraint behaviors to be expressed with simplicity and flexibility. The direct representation of the application in terms of constraints, together with the high-level control, results in short simple programs. Since it is easy to change the model and, separately, the behavior of a program, the paradigm supports experimentation with problem solving methods. In the context of a rapid application methodology, it even supports experimentation with the problem (model) itself.

9.2 Modeling Languages

On the other hand, Operations Researchers have introduced a wide range of highly sophisticated specialized algorithms for different classes of problems. For many OR researchers CLP and Localizer are too powerful—they seek a modeling language rather than a computer programming language in which to encode their problems. Traditional mathematical modeling languages used by OR researchers have offered little control over the search and the constraint propagation. OPL (Van

Hentenryck, 1999) is an extension of such a modeling language to give more control to the algorithm developer. It represents a step towards a full constraint programming language.

By contrast, a number of application development environments (e.g. Visual CHIP) have appeared recently that allow the developer to define and apply constraints graphically, rather than by writing a program. This represents a step in the other direction!

10. APPLICATIONS

10.1 Current Areas of Application

Constraint programming is based on logic. Consequently any formal specification of an industrial problem can be directly expressed in a constraint program. The drawbacks of earlier declarative programming paradigms have been

- that the programmer had to encode the problem in a way that was efficient to execute on a computer;

- that the end user of the application could not understand the formal specification.

The first breakthrough of constraint programming has been to separate the logical representation of the problem from the efficient encoding in the underlying constraint solvers. This separation of logic from implementation has opened up a range of applications in the area of control, verification and validation.

The second breakthrough of constraint programming has been in the area of software engineering. The constraint paradigm has proven to accommodate a wide variety of problem solving techniques, and has enabled them to be combined into hybrid techniques and algorithms, suited to whatever problem is being tackled.

As important as the algorithms to the success of constraint technology, has been the facility to link models and solutions to a graphical user interface that makes sense to the end user. Having developers display the solutions in a form intelligible to the end users, forces the developers to put themselves into the shoes of the users.

Moreover, not only are the final solutions displayed to the user: it is also possible to display intermediate solutions found during search, or even partial solutions. The ability to animate the search in a way that is intelligible to the end user means the users can put themselves into the shoes of the developers. In this way the crucial relationship and understanding between developers and end users is supported and users feel themselves involved in the development of the software that will support them in the future.

As a consequence, constraint technology has been applied very successfully in a range of combinatorial problem solving applications, extending those traditionally tackled using operations research.

The two main application areas of constraint programming are, therefore, (i) control, verification and validation, and (ii) combinatorial problem solving.

10.2 Applications in Control, Verification and Validation

Engineering relies increasingly on software, not only at the design stage, but also during operation. Consider the humble photocopier. Photocopiers are not as humble as they used to be—each system comprises a multitude of components, such as feeders, sorters, staplers and so on. The next generation of photocopiers will have orders of magnitude more components than now. The challenge of maintaining compatibility between the different components, and different versions of the components, has become unmanageable.

Xerox has turned to constraint technology to specify the behavior of the different components in terms of constraints. If a set of components are to be combined in a system, constraint technology is applied to determine whether the components will function correctly and coherently. The facility to specify behavior in terms of constraints has enabled engineers at Xerox not only to simulate complex systems in software but also to revise their specifications before constructing anything and achieve compatibility first time.

Control software has traditionally been expressed in terms of finite state machines. Proofs of safety and reachability are necessary to ensure that the system only moves between safe states (e.g. the lift never moves while the door is open) and that required states are reached (the lift eventually answers every request). Siemens has applied constraint technology to validate control software, using techniques such as Boolean unification to detect any errors. Similar techniques are also used by Siemens to verify integrated circuits.

Constraint technology is also used to prove properties of software. For example, abstract interpretation benefits from constraint technology in achieving the performance necessary to extract precise information about concrete program behavior.

Finally, constraints are being used not only to verify software but to monitor and restrict its behavior at runtime. *Guardian Agents* ensure that complex software, in medical applications for example, never behaves in a way that contravenes the certain safety and correctness requirements.

For applications in control, validation and verification, the role of constraints is to model properties of complex systems in terms of logic, and then to prove theorems about the systems. The main constraint reasoning used in this area is propositional theorem proving. For many applications, even temporal properties are represented in a form such that they can be proved using propositional satisfiability.

Nevertheless, the direct application of abstract interpretation to concurrent constraint programs offers another way to prove properties of complex dynamic systems.

10.3 Combinatorial Problem Solving

Commercially, constraint technology has made a huge impact in problem solving areas such as transportation, logistics, network optimization, scheduling and timetabling, production control, and design, and it is also showing tremendous potential in new application areas such as bioinformatics and virtual reality systems.

Starting with applications to transportation, constraint technology is used by airline, bus and railway companies, all over the world. Applications include timetabling, fleet scheduling, crew scheduling and rostering, stand, slot and platform allocation.

Constraints have been applied in the logistics area for parcel delivery, food, chilled goods, and even nuclear waste. As in other application areas, the major IT system suppliers (such as SAP and I2) are increasingly adopting constraint technology.

Constraints have been applied for Internet service planning and scheduling, for minimizing traffic in banking networks, and for optimization and control of distribution and maintenance in water and gas pipe networks. Constraints are used for network planning (bandwidth, routing, peering points), optimizing network flow and pumping energy (for gas and water), and assessing user requirements.

Constraint technology appears to have established itself as the technology of choice in the areas of short-term scheduling, timetabling and rostering. The flexibility and scalability of constraints was proven tested in the European market (for example at Dassault and Monsanto), but is now used worldwide.

It has been used for timetabling activities in schools and universities, for rostering staff at hospitals, call centers, banks and even radio stations.

An interesting and successful application is the scheduling of satellite operations.

The chemical industry has an enormous range of complex production processes whose scheduling and control is a major challenge, currently being tackled with constraints. Oil refineries and steel plants also use constraints in controlling their production processes. Indeed, many applications of constraints to production scheduling also include production monitoring and control.

The majority of commercial applications of constraint technology have, to date, used finite domain propagation. Finite domains are a very natural way to represent the set of machines that can carry out a task, the set of vehicles that can perform a delivery, or the set of rooms/stands/platforms where an activity can be carried out. Making a choice for one task, precludes the use of the same resource for any other task which overlaps with it, and propagation captures this easily and efficiently.

Naturally, most applications involve many groups of tasks and resources with possibly complex constraints on their availability (for example personnel regulations may require that staff have two weekends off in three, that they must have a day off after each sequence of night-shifts, and that they must not work more than 40 hours a week). For complex constraints like this a number of special constraints have been introduced which not only enable these constraints to be expressed quite naturally, but also associate highly efficient specialized forms of finite domain propagation with each constraint.

10.4 Other Applications

- *Constraints and graphics.* An early use of constraints was for building graphical user interfaces. Now these interfaces are highly efficient and scalable, allowing a diagram to be specified in terms of constraints so that it still carries the same impact and meaning whatever the size or shape of the display hardware. The importance of this functionality in the context of the Internet, and mobile computing, is very clear, and constrained-based graphics is likely to make a major impact in the near future. Constraints are also used in design, involving both spatial constraints and, in the case of real-time systems design, temporal constraints.

- *Constraint databases.* Constraint databases have not yet made a commercial impact, but it is a good bet that future information systems will store constraints as well as data values. The first envisaged application of constraint databases is to geographical information systems. Environmental monitoring will follow, and subsequently design databases supporting both the design and maintenance of complex artifacts such as airplanes.

11. SOME PROMISING AREAS FOR FUTURE APPLICATIONS

There are many topics that could be addressed in additional detail. This section briefly samples a few of these.

11.1 Dynamic Constraints, Soft Constraints

Constraint technology was originally designed to handle "hard" constraints that had to be satisfied in every solution. Moreover, each problem had to be solved from scratch, finding values for a set of previously unassigned decision variables.

What has emerged over the years, particularly in the light of practical applications of the technology, is a need to handle "soft" constraints which should be satisfied if possible, but may be violated if necessary.

Another practical requirement is the need to handle dynamic problems, which may change while their solution is being executed (for example due to machine breakdown; newly placed priority orders; or late running).

These requirements have led to the development of new theoretical (Bistarelli et al., 1992) and practical requirements.

11.2 Hybrid Techniques

As constraint technology has matured, the community has recognized that it is not a standalone technology, but a weapon in an armory of mathematical tools for tackling complex problems. Indeed, an emerging role for constraint programming is as a framework for combining techniques such as constraint propagation; integer/linear and quadratic programming; interval reasoning; global optimization and metaheuristics. This role has become a focus of research in a new conference series called CPAIOR, in a current European project (Coconut) and at recent INFORMS meetings (see respective websites).

11.3 Knowledge Acquisition and Explanation

As constraint programming becomes increasingly commercialized, increasing attention is drawn to "human factors". Issues faced by earlier "knowledge-engineering" technologies must be faced by constraint technology.

Acquiring domain-specific knowledge is obviously a key application issue. Provision needs to be made for interactive acquisition, e.g. in electronic commerce applications. Many problems, e.g. many configuration problems, change over time. While constraint programmers tout the advantages of their declarative paradigm for maintaining programs in the face of such change, acquiring and implementing new knowledge on a large scale still presents challenges.

Users may feel more comfortable when an "explanation" can accompany a solution. Explanation is particularly important when a problem is unsolvable. The user wants to know why, and can use advice on modifying the problem to permit a solution (Amilhastre et al., 2002).

A related set of problems confronts the need constraint programmers have to better understand the solution process. Explanation and visualization of this process can assist in debugging constraint programs, computing solutions more quickly, and finding solutions closer to optimal (Deransart et al., 2000).

11.4 Synthesizing Models and Algorithms

Ideally, people with constraints to satisfy or optimize would simply state their problems, in a form congenial to the problem domain, and from this statement a representation suited to efficient processing and an appropriate algorithm to do the processing would be synthesized automatically. In practice, considerable human expertise is often needed to perform this synthesis. Less ambitiously, tools might be provided to assist the constraint programmer in this regard. Initial experiments with simple learning methods have proven surprisingly effective at producing efficient algorithms for specific problem classes (Minton, 1996).

11.5 Distributed Processing

Distributed constraint processing arises in many contexts. There are parallel algorithms for constraint satisfaction and concurrent constraint programming languages. There are applications where the problem itself is distributed in some manner. There are computing architectures that are naturally "distributed": for example, neural networks.

There is considerable interest in the synergy between constraint processing and software agents. Agents have issues that are naturally viewed in constraint-based terms, e.g. negotiation. Agents can be used to solve constraint satisfaction problems (Yokoo et al., 1998).

11.6 Uncertainty

Real world problems may contain elements of uncertainty. Data may be problematic. The future may not be known. For example, decisions about fuel purchases may need to be made based on uncertain demand dependent on future weather patterns. We want to model and compute with constraints in the presence of such uncertainty (Walsh, 2002).

12. TRICKS OF THE TRADE

The constraints community uses a variety of different tools to solve complex problems. There are a number of constraint programming systems available, which support constraint propagation, search and a variety of other techniques. For pedagogical purposes we will simply show the solution a simple problem, solved using one constraint programming system, ECLiPSe (ECLiPSe, 2005). This system is free for research use, and can be downloaded from the referenced website. If the code developed below is copied into a file, then the reader can load ECLiPSe, compile the file and run the program.

We consider a one-machine scheduling problem. The requirement is to schedule a set of tasks on a machine. Each task has a fixed duration, and each has an earliest start time (the release date) and a latest end time (the due date). How should we schedule the tasks so as to finish soonest?

In constraint programming a problem is handled in three stages:

1 Initialize the problem variables.

2 Constrain the variables.

3 Search for values for the variables that satisfy the constraints.

12.1 Initializing Variables

For the one-machine scheduling problem, a variable is declared to represent the start time of each task. For each task t we introduce a variable St and we declare its lower bound (the release date) and its upper bound (the due date minus the duration). In the code this is achieved by the following lines:

```
EndTime #= StartTime + Duration,
StartTime #>= ReleaseTime,
EndTime #=<DueTime
```

Although `Duration` is written with a capital letter, and therefore is a variable, the program assumes that it will be supplied with a specific input value at runtime.

For the first model of the problem we impose no further constraints on the start time, until search begins.

12.2 Search and Propagation

When the search begins one of the tasks, is chosen to be first. This is done by the following lines of code:

```
choose(Task,Tasks,Rest),
Task = task with [start:StartTime,end:EndTime]
```

Now the following constraints are posted. The start time `StartTime` is constrained to take its lower bound value (in this case, its release date). This is done by the following line of code:

```
fix_to_min(StartTime)
```

The start times of each of the remaining tasks are constrained to be greater than the task end time, `EndTime`. This is achieved by the goal

```
demoPropagate(EndTime, Rest)
```

which is defined as follows:

```
demoPropagate(EndTime, Rest) :-
( foreach( task with start:StartTime, Tasks),
param(EndTime)
do
StartTime #>= EndTime
)
```

As a result, the lower bounds of some or all of the remaining task start times may be increased.

After having "propagated" the new constraints, by computing the new lower bounds for all the start times, search resumes. Another task is chosen to be the first of the remaining tasks, and constraints 1 and 2 are posted as before. This search procedure stops when there are no more tasks. It is defined (recursively) as follows:

```
% If no more tasks are left, then do nothing.
label_starts([]).
% Otherwise, select a task, make it start as early as
% possible and constrain the remaining tasks to start
% after it
label_starts(Tasks) :-
choose(Task,Tasks,Rest),
Task = task with [start:StartTime,end:EndTime],
```

```
fix_to_min(StartTime),
demoPropagate(EndTime,Rest),
label_starts(Rest)
```

12.3 Branch and Bound

When a solution is found the end time of the last task, `FinalEndTime` is recorded. The problem solving process is restarted, but now all tasks are constrained to end before `FinalEndTime`. This is captured by a constraint on each task that its `EndTime` is less than `FinalEndTime`. This behavior is implemented inside the built-in predicate minimize, therefore the code required is simply

```
minimize(label_starts(Tasks),FinalEndTime)
```

If at any time the propagation on a start time makes its lower bound larger than its upper bound, then there is no solution which extends the current task ordering. The system therefore backtracks and chooses a different task to be first at the previous choice point.

12.4 Code

The whole ECLiPSe program for solving the one-machine scheduling problem is as follows. To run it load ECLiPSe, compile the following code and invoke the goal:

```
Tasks = [task with [duration:4, release:1,due:10],
task with [duration:3, release:1, due:15],
{\ldots} % and all the other input tasks!
],
task_schedule(Tasks,FinalEndTime).
```

Code for One-Machine Scheduling Solution

```
% Define a data structure to hold info. about tasks
:- local struct(task(start,duration,end,release,due)).
% Load the finite domain solver
:- lib(fd).

% To solve a problem, first state the constraints and
% then encode the search procedure.
% Names starting with upper-case letters are variables.
task_schedule(Tasks,FinalEndTime) :-
```

```
constrain(Tasks,FinalEndTime),
minimize(label_starts(Tasks),FinalEndTime).
% Constrain the start and end time of each task
constrain(Tasks,FinalEndTime) :-
( foreach(Task,Tasks),
param(FinalEndTime)
do
% Each Task variable holds a data structure
% with the task details
Task = task with [release:ReleaseTime,
due:DueTime,
duration:Duration,
start:StartTime,
end:EndTime
],
% Constrain the start and end times
EndTime #= StartTime + Duration,
StartTime #>= ReleaseTime,
EndTime #=< DueTime,
% Constrain the final end time to follow the end time of
% each task
FinalEndTime #>= EndTime
).

% Stop when there are no more tasks to handle
label_starts([]).
% Select a task, make it start as early as possible
% and constrain the remaining tasks to start after it
label_starts(Tasks) :-
choose(Task,Tasks,Rest),
Task = task with [start:StartTime,end:EndTime],
fix_to_min(StartTime),
demoPropagate(EndTime,Rest),
label_starts(Rest).
% Select any task from a non-empty list.
choose(Task,[TaskTasks],Tasks).
% Alternatively choose a different task
choose(Task,[NotThisTaskTasks],[NotThisTaskRest]) :-
choose(Task,Tasks,Rest).

% Constrain the remaining tasks to start after the given
% previous end time
```

```
demoPropagate(PrevEndTime,Tasks) :-
( foreach( task with start:StartTime, Tasks),
param(PrevEndTime)
do
StartTime {\#}$>$= PrevEndTime
).

% Make the variable Time take its smallest possible
% value
fix_to_min(Time) :-
mindomain(Time,Earliest),
Time #= Earliest
```

12.5 Introducing Redundant Constraints

The first way to enhance this algorithm is by adding a global constraint, specialized for scheduling problems (see Section 8.4). The new constraint does not remove any solutions: it is logically redundant. However, its powerful propagation behavior enables parts of the search space, where no solutions lie, to be pruned. Consequently, the number of search steps is reduced—dramatically for larger problems! The algorithm was devised by operations researchers, but it has been encapsulated by constraint programmers as a single constraint.

12.6 Adding Search Heuristics

The next enhancement is to choose, at each search step, the task with the earliest due date. Whilst this does tend to yield feasible solutions, it does not necessarily produce good solutions, until the end time constraints become tight.

12.7 Using an Incomplete Search Technique

For very large problems, complete search may not be possible. In this case the algorithm may be controlled so as to limit the effort wasted in exploring unpromising parts of the search space. This can be done simply by limiting the number of times a non-preferred ordering of tasks is imposed during search and backtracking, using the LDS algorithm introduced earlier.

The above techniques combine very easily, and the combination is very powerful indeed. As a result constraint programming is currently the technology of choice for operational scheduling problems where task orderings are significant.

The Constraints Archive (http://www.4c.ucc.ie/archive) has pointers to constraint code libraries and constraint programming languages, both freely available software and commercial products.

SOURCES OF ADDITIONAL INFORMATION

Sources of information about constraint programming include:

- Proceedings of the International Conferences on Principles and Practice of Constraint Programming, available in the Springer LNCS series.

- The *Constraints* journal published by Kluwer.

- *Constraint Processing*, by Rina Dechter (published by Morgan Kaufmann, 2003).

- *Programming with Constraints: an Introduction*, by Kim Marriott and Peter Stuckey (MIT Press, 1998).

- On-line Guide to Constraint Programming, maintained by Roman Barták. http://kti.ms.mff.cuni.cz/~bartak/constraints/

- The comp.constraints newsgroup.

- http://carlit.toulouse.inra.fr/cgi-bin/mailman/listinfo/csp (the CSP mailing list).

Many other sources of information can be found on the web at

- Constraint Programming Online: http://slash.math.unipd.it/cp/index.php

- Constraints Archive: http://www.4c.ucc.ie/archive.

these two websites seem unavailable; I haven't checked the others.

Acknowledgments

Some of this material is based upon works supported by the Science Foundation Ireland under Eugene Freuder's Grant No. 00/PI.1/C075; some of his contribution to this chapter was prepared while he was at the University of New Hampshire. Richard Wallace and Dan Sabin provided some assistance. The contents of this chapter overlap with a chapter by the same authors on Constraint Satisfaction in the *Handbook of Metaheuristics*, edited by Fred W. Glover and Gary A. Kochenberger, and published by Kluwer Academic Press.

References

Amilhastre, J., Fargier, H. and Marquis, P., 2002, Consistency restoration and explanations in dynamic CSPs—Application to configuration, *Artif. Intell.* **135**: 199–234.

Bacchus, F., Chen, X., van Beek, P. and Walsh, T., 2002, Binary vs non-binary constraints, *Artif. Intell.*

Bacchus vol/pp.?

Bessiere, C., Freuder, E. and Regin, J., 1999, Using constraint metaknowledge to reduce arc consistency computation, *Artif. Intell.* **107**: 125–148.

Bistarelli, S., Fargier, H., Montanari, U., Rossi, F., Schiex, T. and Verfaille, G., 1996, Semiring-based CSPs and valued CSPs: Basic properties, in: *Over-Constrained Systems,* Lecture Notes in Computer Science, Vol. 1106, M. Jampel, E. C. Freuder, and M. Maher, eds, Springer, Berlin, pp. 111–150.

Cheng, B., Choi, K., Lee, J. and Wu, J., 1999, Increasing constraint propagation by redundant modeling: an experience report, *Constraints* **4**:167–192.

Cheeseman, P., Kanefsky, B. and Taylor, W., 1991, Where the really hard problems are, in: Proc. 12th Int. Joint Conference in Artificial Intelligence, Morgan Kaufmann, San Mateo, CA, pp. 331–337.

www.mat.univie.ac.at/∼neum/glopt/coconut/

www.informs.org/Conf/CORS-INFORMS2004/

www-sop.inria.fr/coprin/cpaior04/

Debruyne, R. and Bessière, C., 2001, Domain filtering consistencies, *J. Artif. Intell. Res.* **14**:205–230.

Dechter, R., 1990, Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition, *Artif. Intell.* **41**:273–312.

Dechter, R., 2003, *Constraint Processing,* Morgan Kaufmann, San Mateo, CA.

Dechter, R. and Frost, D., 2002, Backjump-based backtracking for constraint satisfaction problems, *Artif. Intell.* **136**: 147–188.

de Givry, S. and Jeannin, L., 2004, *A Library for Partial and Hybrid Search Methods,* www.crt.umontreal.ca/cpaior/

Deransart, P., Hermenegildo, M. and Maluszynski, J., eds, 2000, *Analysis and Visualization Tools for Constraint Programming,* Lecture Notes in Computer Science, Vol. 1870, Springer, Berlin.

Dechter, R., Meiri, I. and Pearl, J., 1991, Temporal constraint networks, *Artif. Intell.* **49**:61–95.

www.icparc.ic.ac.uk/eclipse/

Freuder, E., 1978, Synthesizing constraint expressions, *Commun. ACM* **11**:958–966.

Freuder, E., 1982, A sufficient condition for backtrack-free search, *J. Assoc. Comput. Mach.* **29**:24–32.

Freuder, E., 1991, Eliminating interchangeable values in constraint satisfaction problems, in: *Proc. 9th Natl Conf. on Artificial Intelligence,* AAAI/MIT Press, Menlo Park, CA, pp. 227–233.

Focacci, F., Lodi, A. and Milano, M., 1999, Cost-based domain filtering, in: *Principles and Practice of Constraint Programming,* Lecture Notes in Computer Science, Vol. 1713, J. Jaffar, ed., Springer, Berlin.

ref Focacci 1999 not cited.

Gomes, C., Selman, B. and Crato, N., 1997, Heavy-tailed distributions in combinatorial search, in: *Proc. 3rd Int. Conf. on Principles and Practice of Constraint Programming (CP'97),* Lecture Notes in Computer Science, Vol. 1330, G. Smolka, ed., Springer, Berlin, pp. 121–135.

Harvey, W. and Ginsberg, M., 1995, Limited discrepancy search, in: *Proc. 14th Int. Joint Conf. on Artificial Intelligence (IJCAI-95).* Morgan Kaufmann, San Mateo, CA, pp. 607–615.

Jeavons, P., Cohen, D. and Gyssens, M., 1997, Closure properties of constraints, *J. ACM* **44**:527–548.

Jaffar, J. and Lassez, J.-L., 1987, Constraint logic programming, *Proc. Ann. ACM Symp. on Principles of Programming Languages (POPL),* pp. 111–119.

Kondrak, G. and van Beek, P., 1997, A theoretical evaluation of selected backtracking algorithms, *Artif. Intell.* **89**:365–387.

Laburthe, F. and Caseau, Y., 1998, SALSA: A language for search algorithms, in: *Proc. 4th Int. Conf. on the Principles and Practice of Constraint Programming (CP'98),* Pisa, Italy, Lecture Notes in Computer Science, Vol. 1520, Springer, Berlin, pp. 310–324.

Mackworth, A., 1977, Consistency in networks of relations, *Artif. Intell.* **8**:99–118.

Minton, S., 1996, Automatically configuring constraint satisfaction programs: A case study, *Constraints* **1**:7–43.

Minton, S., Johnston, M. D., Philips, A. B. and Laird, P., 1992, Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling, *Artif. Intell.* **58**:61–205.

Michel, L. and Van Hentenryck, P., 1999, Localizer: a modeling language for local search, *INFORMS, J. Comput.* **11**:1–14.

Regin, J.-C., 1994, A filtering algorithm for constraints of difference in CSPs, in: *Proc. AAAI 12th Natl Conf. on Artificial Intelligence,* AAAI Press, Menlo Park, CA, pp. 362–367.

Régin, J.-C., 2001, Minimization of the number of breaks in sports scheduling problems using constraint programming, in: Constraint

Programming and Large Scale Discrete Optimization, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 57, E. Freuder and R. Wallace, eds, American Mathematical Society, Providence, RI, pp. 115–130.

Sabin, D. and Freuder, E., 1997, Understanding and Improving the MAC Algorithm, in: *Proc. 3rd Int. Conf. on Principles and Practice of Constraint Programming (CP'97),* Lecture Notes in Computer Science, Vol. 1330, G. Smolka, ed., Springer, Berlin, pp. 167–181.

Salkin, H. M., 1970, On the merit of the generalized origin and restarts in implicit enumeration, *Oper. Res.* **18**:549–554.

ref Salkin 1970 not cited.

Selman, B., Levesque, H. and Mitchell, D., 1992, A new method for solving hard satisfiability problems, in: *Proc. 10th Natl Conf. on Artificial Intelligence (AAAI-92),* San Jose, CA, pp. 440–446.

Tsang, E., 1993, *Foundations of Constraint Satisfaction,* Academic, London.

Van Hentenryck, P., 1999, *The OPL optimization programming language,* MIT Press, Cambridge, MA.

Van Hentenryck, P., Simonis, H. and Dincbas, M., 1992, Constraint satisfaction using constraint logic programming, *Artif. Intell.* **58**:113–159.

Verfaillie, G., Lemaitre, M. and Schiex, T., 1996, Russian doll search for solving constraint optimization problems, in: *Proc. 13th AAAI,* pp. 181–187.

ref Verfaillie 1996 not cited.

Veron, A., Schuerman, K., Reeve, M. and Li, L.-L., 1993, Why and How in the ElipSys OR-Parallel CLP System, in: *Proc. 5th Int. PARLE Conf.,* pp. 291–303.

Walsh, T., 2002, Stochastic constraint programming, in: *Proc. 15th ECAI,* F. van HArmelen, ed., pp. 111–115.

Yokoo, M., Durfee, E., Ishida, T. and Kuwabara, K., 1998, The distributed CSP: Formalization and algorithms, *IEEE Trans. on Knowledge and Data Engineering* **10**:673–685.

Yokoo, M., 1994, Weak-commitment search for solving constraint satisfaction problems, in: *Proc. 12th Natl Conf. on Artificial Intelligence (AAAI-94),* Vol. 1, pp. 313–318.

ref Yokoo 1994 not cited.

Zhang, W., 1998, Complete anytime beam search, in: *Proc. 15th Natl Conf. on Artificial Intelligence (AAAI-98),* pp. 425–430.