Principles of Constraint Programming

Krzysztof R. Apt

February 19, 1999

Chapter 1

Introduction

1.1 Preliminaries

Constraint programming is an alternative approach to programming in which the programming process is limited to a generation of requirements (constraints) and a solution of these requirements by means of general or domain specific methods.

The general methods are usually concerned with techniques of reducing the search space and with specific search methods. In contrast, the domain specific methods are usually provided in the form of special purpose algorithms or specialised packages, usually called *constraint solvers*. Typical examples of constraint solvers are:

- a program that solves systems of linear equations,
- a package for linear programming,
- an implementation of the unification algorithm, a cornerstone of automated theorem proving.

Problems that can be solved in a natural way by means of constraint programming are usually those for which efficient algorithms are lacking (for example computationally intractable problems) or for which formalization in terms of laws (for example electrical engineering) leads to a more flexible style of programming in which the dependencies between the relevant variables can be expressed in the most general form.

In fact, many problems that need to be solved by means of computing are not precisely defined or their precise specification may depend on the quality of a solution (such as the speed with which it is computed) to an initial version of the problem. When solving such problems one needs to proceed by several iterations. Modelling these problems by means of constraints can be often beneficial. Indeed, the appropriate modification of the program can then be often taken care of by modification of some constraints or by altering an appropriate fixed program component (for example the one that defines the adopted search method).

An additional aspect brought in by constraint programming is that modelling by means of constraints leads to a representation by means of relations. In a number of circumstances this representation allows us to use the same program for different purposes.

This can be useful in several cases, for instance when trying to figure out which input led to a given output. In conventional programming languages relations have to be converted first to functions and this possibility is then lost.

The use of relations as a basis for problem formulation bears some resemblance to database systems, for instance relational databases. The difference is that in database systems such relations (for instance tables) are usually explicitly given and the task consists of efficiently querying them, while in constraint programming these relations (for instance equations) are given implicitly and the task consists of solving them.

We can summarize these characteristics of constraint programming as follows.

- The programing process consists of two phases: a generation of a problem representation by means of constraints and a solution of it. In practice, both phases can consist of several smaller steps and can be interleaved.
- The representation of a problem by means of constraints is very flexible because the constraints can be added, removed or modified.
- The use of relations to represent constraints blurs the difference between input and output. This makes it possible to use the same program for a number of purposes.

Constraint programming has already been successfully applied in numerous domains including interactive graphic systems (to express geometric coherence in the case of scene analysis), operations research problems (various scheduling problems), molecular biology (DNA sequencing), business applications (option trading), electrical engineering (location of faults in the circuits), circuit design (computing the layouts and verification of the design), numerical computation (solving of algebraic equations with guaranteed precision), natural language processing (construction of efficient parsers), computer algebra (solving and/or simplifying equations over various algebraic structures), etc.

The growing importance of this area can be witnessed by the fact that there are now separate annual conferences on constraint programming and its applications and that a new journal called "Constraints" has been launched (by Kluwer). But the field is still young and only a couple of books on this subject appeared so far. In contrast, several special issues of computer science journals devoted to the subject of constraints have appeared.

1.2 Constraint Satisfaction Problems

To be more specific about this programming style we need to introduce a central notion of this area: a constraint satisfaction problem. Intuitively, a constraint satisfaction problem consists of a finite set of relations over some domains.

In practice we use a finite sequence of variables with respective domains associated with them. A constraint over a sequence of variables is then a subset of the Cartesian product of their domains. A constraint satisfaction problem is then a finite sequence of variables with respective domains together with a finite set of constraints, each on a subsequence of these variables.

A more formal definition is as follows. Consider a finite sequence of variables $X := x_1, \ldots, x_n$ where $n \geq 0$, with respective domains $\mathcal{D} := D_1, \ldots, D_n$ associated with them. So each variable x_i ranges over the domain D_i. By a *constraint* C on X we mean a subset of $D_1 \times \ldots \times D_n$. If C equals $D_1 \times \ldots \times D_n$ then we say that C is *solved*.

In the boundary case when $n = 0$ we admit two constraints, denoted by $\top$ and $\bot$, that denote respectively the *true constraint* (for example $0 = 0$) and the *false constraint* (for example $0 = 1$).

Now, by a *constraint satisfaction problem*, CSP in short, we mean a finite sequence of variables $X := x_1, \ldots, x_n$ with respective domains $\mathcal{D} := D_1, \ldots, D_n$, together with a finite set $\mathcal{C}$ of constraints, each on a subsequence of X. We write such a CSP as $\langle \mathcal{C} \ ; \ \mathcal{DE} \rangle$, where $\mathcal{DE} := x_1 \in D_1, \ldots, x_n \in D_n$ and call each construct of the form $x \in D$ a *domain expression*. To simplify the notation from now on we omit the "{ }" brackets when presenting specific sets of constraints $\mathcal{C}$.

As an example consider the sequence of four integer variables x, y, z, u and the following two constraints on them: $x + y = z$ and $x = u + 1$.

According to the above notation we write this CSP as

$$\langle x + y = z, \ x = u + 1 \ ; \ x \in \mathcal{Z}, y \in \mathcal{Z}, z \in \mathcal{Z}, u \in \mathcal{Z} \rangle,$$

where $\mathcal{Z}$ denotes the set of integers.

Before we proceed we need to clarify one simple matter. When defining constraints and CSP's we refer to sequences (respectively subsequences) of variables and *not* to the sets (respectively subsets) of variables. Namely, given a CSP each of its constraints is defined on a *subsequence* and not on a *subset* of its variables. In particular, the above constraint $x + y = z$ is defined on the subsequence x, y, z of the sequence x, y, z, u and not on the subset $\{x, y, z\}$ of it.

In particular, the sequence y, x is not a subsequence x, y, z, u, so if we add to the above CSP the constraint $y < x$ we cannot consider it as a constraint on y, x. But we can view it as a constraint on x, y and if we wish we can rewrite it as $x > y$. The reliance on sequences and subsequences of variables instead on sets and subsets of will allow us to analyze in a simple way variable orderings when searching for solutions to a given CSP. Each variable ordering uniquely determines a CSP that is derived from the given one in a straightforward manner. In short, this presentation does not introduce any restrictions and simplifies some considerations.

In what follows we assume that the constraints and the domain expressions are defined in some specific, further unspecified, language. In this representation it is implicit that each constraint is a subset of the Cartesian product of the associated variable domains. For example, if we consider the CSP $\langle x < y \ ; \ x \in [0..10], y \in [5..10] \rangle$, then we view the constraint $x < y$ as the set of all pairs (a, b) with $a \in [0..10]$ and $b \in [5..10]$ such that $a < b$.

We now define the crucial notion of a solution to a CSP. Intuitively, a solution to a CSP is a sequence of values for all of its variables such that all its constraints are satisfied.

More precisely, consider a CSP $\langle \mathcal{C} \ ; \ \mathcal{DE} \rangle$ with $\mathcal{DE} := x_1 \in D_1, \ldots, x_n \in D_n$. We say that an n-tuple $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ *is a solution to* $\langle \mathcal{C} \ ; \ \mathcal{DE} \rangle$ if for every constraint $C \in \mathcal{C}$ on the variables $x_{i_1}, \ldots, x_{i_m}$ we have

$$(d_{i_1}, \ldots, d_{i_m}) \in C.$$

If a CSP has a solution, we say that it is *consistent* and otherwise we say that it is *inconsistent*. For further discussion it is useful to introduce the following simple notion. Take a sequence of variables $X := x_1, \ldots, x_n$ with the corresponding sequence of domains $\mathcal{D} := D_1, \ldots, D_n$ and consider an element $d := (d_1, \ldots, d_n)$ of $D_1 \times \ldots \times D_n$ and a subsequence $Y := x_{i_1}, \ldots, x_{i_\ell}$ of X. Then we denote the sequence $(d_{i_1}, \ldots, d_{i_\ell})$ by $d[Y]$ and call it the *projection* of d on Y.

So given a CSP a tuple of values from the corresponding domains is a solution to it if for every constraint C of it the projection of this tuple on the sequence of the variables of C satisfies C.

Thus the sequence $(1, 3, 4, 0)$ is a solution to the above CSP, because the subsequence $(1, 3, 4)$ satisfies the first constraint, $x + y = z$, and the subsequence $(1, 0)$ satisfies the second constraint, $x = u + 1$.

To solve CSP's one often transforms them in a specific way until all solutions have been found or it is clear that no solution exists. To discuss the outcome of such transformations we need two concepts.

We call a CSP *solved* if it is of the form $\langle \emptyset\ ;\ \mathcal{DE} \rangle$ where no domain in $\mathcal{DE}$ is empty, and *failed* if it either contains the false constraint $\bot$ or some of its domains is empty. Clearly, a failed CSP is inconsistent.

The transformations of CSP's need to be such that their equivalence in an appropriate sense is preserved. This brings us to the notion of equivalence of CSP's. We define it first for a special case.

Consider two CSP's $\mathcal{P}_1$ and $\mathcal{P}_2$ with the same sequence of variables. We say that $\mathcal{P}_1$ and $\mathcal{P}_2$ are *equivalent* if they have the same set of solutions.

So for example the CSP's

$$\langle 3x \perp 5y = 4\ ;\ x \in [0..9], y \in [1..8] \rangle$$

and

$$\langle 3x \perp 5y = 4\ ;\ x \in [3..8], y \in [1..4] \rangle$$

are equivalent, since both of them have $x = 3, y = 1$ and $x = 8, y = 4$ as the only solutions.

Note that the solved constraints can be deleted from any CSP without affecting equivalence.

The above definition is rather limited as it cannot be used to compare CSP's with different sequences of variables. Such situations often arise, for example when new variables are introduced or some variables are eliminated.

Consider for instance the problem of solving the equation

$$2x^5 \perp 5x^4 + 5 = 0 \tag{1.1}$$

over the reals. One way to proceed could be by using a new variable y and transforming (1.1) to the following two equations:

$$2x^5 \perp y + 5 = 0, \tag{1.2}$$

$$y = 5x^4, \tag{1.3}$$

so that in each equation every variable occurs at most once.

Now the CSP formed by the equations (1.2) and (1.3) has one variable more than the one formed by (1.1), so we cannot claim that these two CSP's are equivalent in the sense of the definition just introduced. To deal with such situations we introduce a more general notion of equivalence.

Consider two CSP's $\mathcal{P}_1$ and $\mathcal{P}_2$ and a sequence X of common variables. We say that $\mathcal{P}_1$ and $\mathcal{P}_2$ are *equivalent w.r.t.* X if

- for every solution d to $\mathcal{P}_1$ a solution to $\mathcal{P}_2$ exists that coincides with d on the variables in X,
- for every solution e to $\mathcal{P}_2$ a solution to $\mathcal{P}_1$ exists that coincides with e on the variables in X.

Using the earlier introduced notion of projection, we can define this notion of equivalence in a more succinct way: $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent w.r.t. X iff

$$\{d[X] \mid d \text{ is a solution to } \mathcal{P}_1\} = \{d[X] \mid d \text{ is a solution to } \mathcal{P}_2\}.$$

Clearly, two CSP's with the same sequence of variables X are equivalent iff they are equivalent w.r.t. X, so the latter notion of equivalence is a generalization of the former one. When transforming CSP's one often tries to maintain equivalence w.r.t. to the initial sequence of variables. Note for example that the CSP formed by the equation (1.1) and the one formed by the equations (1.2) and (1.3) are equivalent w.r.t. to $\{x\}$.

Once we represent our problem as a constraint satisfaction problem we need to solve it. In practice we are most often interested in:

- determining whether a CSP has a solution (is consistent),
- finding a solution (respectively all solutions) to a CSP,
- finding an optimal solution to a CSP (respectively all optimal solutions) w.r.t. some quality measure.

1.3 Constraint Programming

As already mentioned, constraint programming deals with solving constraint satisfaction problems. This naturally implies the following characteristics of constraint programming.

To start with, the problem to be tackled can be expressed as a constraint satisfaction problem. Consequently, it can be formulated using

- some variables ranging over specific domains and constraints over these variables;
- some language in which the constraints are expressed.

Then to solve this problem one either uses

- domain specific methods,

or

- general methods,

or a combination of both.

From this point of view constraint programming embodies such diverse areas as Linear Algebra, Global Optimization, Linear and Integer Programming, etc. Therefore we should clarify one essential point. An important part of constraint programming deals with the study of *general* methods and techniques. Clearly, they can be specialised to specific cases. Now, for such specific cases some special purpose methods of solving the constraints have been often developed and are available in the form of packages or libraries. For example, in the case of systems of linear equations well-known algorithms dealing with Linear Algebra are readily available. So it does not make much sense to apply these general methods to the cases for which specialised and efficient methods are known.

Therefore the distinction between the general and domain specific methods is crucial and one needs to be keep it in mind when studying constraint programming.

The general methods are usually useful in cases when no good specialised methods of solving CSP's are known. Just to mention two examples. So-called Boolean constraints deal with the problem of finding variable assignments that satisfy propositional formulas. This problem is in fact the first NP-complete problem that was identified. As another example take the notoriously difficult problem of solving systems of (in)equalities between polynomials over reals with a guaranteed precision. In both cases the general techniques of constraint programming turned out to be beneficial.

The fact that for some specific cases efficient methods for solving CSP's exist and for some others no such methods are known, naturally calls for systems for constraint programming in which both the general and the domain specific methods are available. The former are usually present in the form of specific constructs that support search, while the latter are often present in the form of various built-ins.

Selected References

Apt, K. R. & Kirchner, C., eds (1998), *Fundamenta Informaticae. Special Issue: Foundations of Constraint Programming*, Vol. 34(3), IOS Press.

Codognet, P., ed. (1998), *Science of Computer Programming. Special Issue on Concurrent Constraint Programming*, Vol. 34 (1-2), Elsevier.

Fron, A. (1994), *Programmation par Constraintes*, Addison-Wesley. In French.

Frühwirth, T. & Abdennadher, S. (1997), *Constraint-Programmierung*, Springer-Verlag, Berlin. In German.

Frühwirth, T., Herold, A., Küchenhoff, V., Le Provost, T., Lim, P., Monfroy, E. & Wallace, M. (1992), Constraint logic programming: An informal introduction, *in* G. Comyn, N. E. Fuchs & M. J. Ratcliffe, eds, 'Logic Programming in Action: Proc. 2nd Intl. Logic Programming Summer School, LPSS'92, Zürich, Switzerland, 7–11 Sept 1992', Vol. 636 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin, pp. 3–35.

Jaffar, J. & Maher, M. (1994), 'Constraint logic programming: A survey', *Journal of Logic Programming* **19, 20**, 503–581.

Jouannaud, J.-P., ed. (1998), *Information and Computation. Special Issue on the 1994 Internationa Conference on Constraints in Computational Logics*, Vol. 142(1), Academic Press.

Marriott, K. & Stuckey, P. (1998*a*), *Programming with Constraints*, The MIT Press, Cambridge, Massachusetts.

Marriott, K. & Stuckey, P., eds (1998*b*), *The Journal of Logic Programming. Special Issue: Constraint Logic Programming*, Vol. 37(1-3), Elsevier.

Montanari, U. & Rossi, F., eds (1997), *Theoretical Computer Science. Special Issue on Principles and Practice of Constraint Programming*, Vol. 173(1), Elsevier.

Tsang, E. (1993), *Foundations of Constraint Satisfaction*, Academic Press.

Van Hentenryck, Saraswat & et al. (1996), 'Strategic directions in constraint programming', *ACM Computing Surveys* **28**(4), 701–726.

Chapter 2

Examples

Let us consider now various, usually well-known, examples of constraint satisfaction problems. When presenting them we shall make clear that a given problem often admits more than one natural representation as a constraint satisfaction problem. We limit ourselves here to the examples of CSP's that are simple to explain and for which the use of general techniques turned out to be beneficial.

In each example we use some self-explanatory language to define the constraints. Later we shall be more precise and shall discuss in detail specific languages in which constraints will be described. This will allow us to classify the constraint satisfaction problems according to the language in which they are defined.

2.1 Constraint Satisfaction Problems on Integers

In this section we discuss CSP's the variables of which range over integers.

Example 2.1 *SEND + MORE = MONEY.*

This is a classical example of a so-called *cryptarithmetic problem*. We are asked to replace each letter by a different digit so that the above sum, that is

$$\begin{array}{r} \mathit{SEND} \\ +\ \ \mathit{MORE} \\ \hline \mathit{MONEY} \end{array}$$

is correct.

Here the variables are S, E, N, D, M, O, R, Y. Because S and M are the leading digits, the domain for each of them consists of the interval [1..9]. The domain of each of the remaining variables consists of the interval [0..9]. This problem can be formulated as the equality constraint

$$\begin{array}{rl} & 1000 \cdot S + 100 \cdot E + 10 \cdot N + D \\ + & 1000 \cdot M + 100 \cdot O + 10 \cdot R + E \\ = & 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y \end{array}$$

combined with 28 disequality constraints $x \neq y$ for $x, y \in \{S, E, N, D, M, O, R, Y\}$ with, say, x preceding y in the alphabetic order.

Another possible representation of this problem as a CSP is obtained by assuming that the domains of all variables are the same, namely the interval [0..9], and by adding to the above constraints two disequality constraints:

$$S \neq 0,\ M \neq 0.$$

Yet another possibility consists of additionally introducing per each column a "carry" variable ranging over [0..1] and using the following four equality constraints, one for each column, instead of the above single one:

$$D + E = 10 \cdot C_1 + Y,$$

$$C_1 + N + R = 10 \cdot C_2 + E,$$

$$C_2 + E + O = 10 \cdot C_3 + N,$$

$$C_3 + S + M = 10 \cdot C_4 + O,$$

$$C_4 = M.$$

Here, $C_1, \ldots, C_4$ are the carry variables.

The above problem has a unique solution depicted by the following sum:

$$\begin{array}{rr} & 9567 \\ + & 1085 \\ \hline & 10652 \end{array}$$

As a consequence, each of the above representations of it as a CSP has a unique solution, as well. Other well-known cryptarithmetic problems that have a unique solution are:

$$\begin{array}{rr} & GERALD \\ + & DONALD \\ \hline & ROBERT \end{array}$$

$$\begin{array}{rr} & CROSS \\ + & ROADS \\ \hline & DANGER \end{array}$$

$$\begin{array}{rr} & TEN \\ + & TEN \\ + & FORTY \\ \hline & SIXTY \end{array}$$

and in French

$$\begin{array}{rr} & LIONNE \\ + & TIGRE \\ \hline & TIGRON \end{array}$$

□

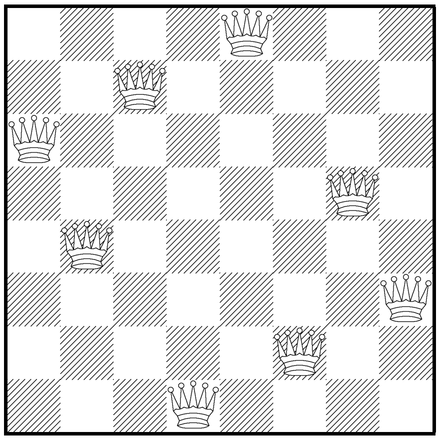
Figure 2.1: One of 92 solutions to the 8 queens problem

Example 2.2 *n Queens Problem.*

This is probably the most known CSP. One is asked to place n queens on the chess board $n \times n$ so that they do not attack each other, where $n \geq 4$. Figure 2.1 shows a solution to the problem for $n = 8$.

To represent this problem as a CSP we use n variables, $x_1, \ldots, x_n$, each with the domain $[1..n]$. The idea is that x_i denotes the position of the queen placed on the i-th row. For example, the solution presented in Figure 2.1 corresponds to the sequence of values (5,3,1,7,2,8,6,4). The appropriate constraints can be formulated as the following disequalities for $i \in [1..n \perp 1]$ and $j \in [i + 1..n]$:

$$x_i \neq x_j,$$

(no two queens in the same column),

$$x_i \perp x_j \neq i \perp j,$$

(no two queens in each North-West – South-East diagonal),

$$x_i \perp x_j \neq j \perp i$$

(no two queens in each South-West – North-East diagonal).

In Section 2.3 we shall discuss another natural representation of this problem. □

Example 2.3 *Zebra Puzzle.*

As another example consider the following famous puzzle of Lewis Carroll. A small street has five differently colored houses on it. Five men of different nationalities live in these five houses. Each man has a different profession, each man likes a different drink, and each has a different pet animal. We have the following information:

The Englishman lives in the red house.
The Spaniard has a dog.
The Japanese is a painter.
The Italian drinks tea.
The Norwegian lives in the first house on the left.
The owner of the green house drinks coffee.
The green house is on the right of the white house.
The sculptor breeds snails.
The diplomat lives in the yellow house.
They drink milk in the middle house.
The Norwegian lives next door to the blue house.
The violinist drinks fruit juice.
The fox is in the house next to the doctor's.
The horse is in the house next to the diplomat's.
The question is who has the zebra and who drinks water?

To formulate this puzzle as a CSP we first try to determine the variables and their domains. Note that this puzzle involves:

- five houses, which we number from left to right: 1, 2, 3, 4, 5,
- five colours, namely red, green, white, yellow, blue,
- five nationalites, namely English, Spanish, Japanese, Italian, Norwegian,
- five pets, namely dog, snails, fox, horse, and (implicitly) zebra,
- five professions, namely painter, sculptor, diplomat, violinist, doctor,
- five drinks, namely tea, coffee, milk, juice, and (implicitly) water.

To solve this puzzle it suffices to determine for each house its five characteristics:

- colour,
- nationality of the owner,
- pet of the owner,
- profession of the owner,
- favourite drink of the owner.

So we introduce 25 variables, five for each of the above five characteristics. For these variables we use the following mnemonic names:

- "colour" variables: `red`, `green`, `white`, `yellow`, `blue`,
- "nationality" variables: `english`, `spaniard`, `japanese`, `italian`, `norwegian`,

- "pet" variables: `dog, snails, fox, horse, zebra`,
- "profession" variables: `painter, sculptor, diplomat, violinist, doctor`,
- "drink" variables: `tea, coffee, milk, juice, water`.

We assume that each of these variables ranges over [1...5]. If, for example, `violinist` $=$ 3 then we interpret this as the statement that the violinist lives in house no. 3.

After these preparations we can now formalize the given information as the following constraints:

- The Englishman lives in the red house: `english = red`,
- The Spaniard has a dog: `spaniard = dog`,
- The Japanese is a painter: `japanese = painter`,
- The Italian drinks tea: `italian = tea`,
- The Norwegian lives in the first house on the left: `norwegian` $= 1$,
- The owner of the green house drinks coffee: `green = coffee`,
- The green house is on the right of the white house: `green = white` $+ 1$,
- The sculptor breeds snails: `sculptor = snails`,
- The diplomat lives in the yellow house: `diplomat = yellow`,
- They drink milk in the middle house: `milk = 3`,
- The Norwegian lives next door to the blue house: $|\texttt{norwegian} \perp \texttt{blue}| = 1$,
- The violinist drinks fruit juice: `violinist = juice`,
- The fox is in the house next to the doctor's: $|\texttt{fox} \perp \texttt{doctor}| = 1$,
- The horse is in the house next to the diplomat's: $|\texttt{horse} \perp \texttt{diplomat}| = 1$.

Additionally, we need to postulate that for each of the characteristics the corresponding variables are different. This means that we introduce fifty disequality constraints, ten for each characteristics. For example $\texttt{red} \neq \texttt{white}$ is one of such disequalities.

Now, it turns out that there is exactly one assignment of values to all 25 variables for which all constraints are satisfied. Because of this the puzzle has a unique solution. Indeed, the puzzle is solved once we find in this unique assignment for which "profession" variables

$$x, y \in \{\texttt{painter}, \texttt{sculptor}, \texttt{diplomat}, \texttt{violinist}, \texttt{doctor}\}$$

we have

$$x = \texttt{zebra} \text{ and } y = \texttt{water}.$$

□

2.2 Constraint Satisfaction Problems on Reals

Let us move now to the case of CSP's the variables of which range over reals.

Example 2.4 *Spreadsheets.*
Spreadsheet systems, such as Excel or Lotus123, are very popular for various office-like applications. These systems have in general a number of very advanced features but their essence relies on constraints.

To be more specific consider Table 2.1. It represents a spreadsheet, in which the values for the cells D4, D5, E7 and E8 are computed by means of the formulas present in these cells.

	A	B	C	D	E
1	tax	0.17			
2					
3	product	price	quantity	total	
4	tomatoes	3.5	1.5	$B4 \star C4$	
5	potatoes	1.7	4.5	$B5 \star C5$	
6					
7				Grand Total	$D4 + D5$
8				Final Amount	$E7 \star (1 + B1)$

Table 2.1: A spreadsheet

	A	B	C	D	E
1	tax	0.17			
2					
3	product	price	quantity	total	
4	tomatoes	3.5	1.5	5.25	
5	potatoes	1.7	4.5	7.65	
6					
7				Grand Total	12.9
8				Final Amount	15.093

Table 2.2: Solution to the spreadsheet of Table 2.1

Equivalently, we can represent the spreadsheet of Table 2.1 by means of a CSP that consists of nine variables: B1, B4, B5, C4, C5, D4, D5, E7 and E8, each ranging over real numbers, and the following nine constraints:
B1 = 0.17,
B4 = 3.5,

B5 = 1.7,
C4 = 1.5,
C5 = 4.5,
D4 = B4 $\star$ C4,
D5 = B5 $\star$ C5,
E7 = D4 + D5,
E8 = E7 $\star$ (1 + B1).

A spreadsheet system solves these constraints and updates the solution each time a parameter is modified. The latter corresponds to modifying a numeric value in a constraint of the form $x = v$, where x is a variable and v a numeric value.

In the case of the spreadsheet of Table 2.1 the solution is represented by the spreadsheet of Table 2.2. □

Example 2.5 *Finding zero's of polynomials of higher degree.*
Consider the polynomial $2 * x^5 \perp 5 * x^4 + 5$ represented in Figure 2.2.

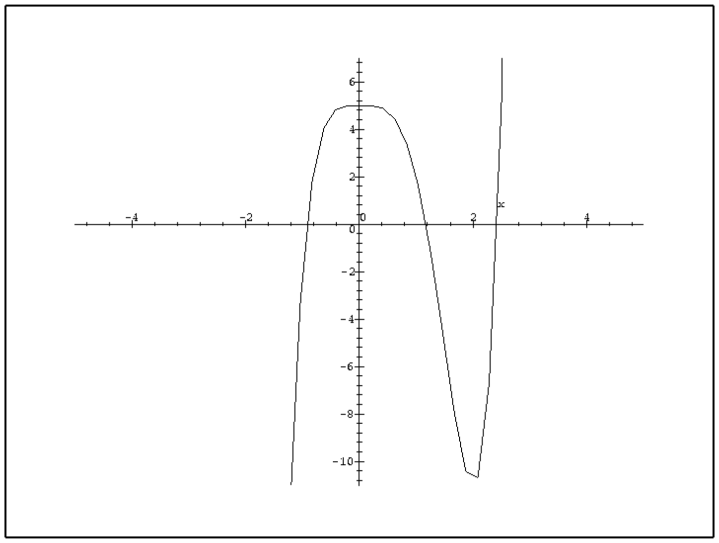

Figure 2.2: The diagram of the polynomial $2 * x^5 \perp 5 * x^4 + 5$

Suppose we wish to find its zero's. The difficulty lies in the fact that in general, by Galois' theorem, the zero's of polynomials of degree higher than four cannot be expressed as radicals, so by means of the four arithmetic operations and the root extraction. In fact, $2 * x^5 \perp 5 * x^4 + 5$ is one of such polynomials. Another complication is that the real numbers cannot be faithfully represented in the computer.

The latter problem is dealt with by using the computer representable real numbers, i.e., the floating point numbers, and by producing solutions in the form of intervals with floating point bounds.

In general, given a polynomial $f(x)$, we consider a CSP with a single constraint $f(x) = 0$ where the variable x ranges over reals. Further, we assume a fixed finite set of floating point numbers augmented with $\perp\infty$ and ∞, and denoted by F. By an *interval CSP* we mean a CSP with the single constraint $f(x) = 0$ and a domain expression of the form $x \in [l, r]$, where $l, r \in F$, so a CSP of the form $\langle f(x) = 0 \; ; \; x \in [l, r] \rangle$.

The original CSP is then transformed into a disjunction of interval CSP's which is equivalent to the original CSP, in the sense that the set of solutions to the original CSP equals the union of the set of solutions to the final interval CSP's.

In the case of the above polynomial, choosing the accuracy of 16 digits after the decimal comma, we get the disjunction of thre interval CSP's based on the following intervals:

$$x \in [\perp.9243580149260359080,$$
$$\perp.9243580149260359031],$$

$$x \in [1.171162068483181786,$$
$$1.171162068483181791],$$

$$x \in [2.428072792707314923,$$
$$2.428072792707314924],$$

An additional argument is needed to prove that each interval contains a zero. These considerations generalize to polynomials in an arbitrary number of variables and to *constrained optimization problems* according to which we are asked to optimize some real-valued function subject to some polynomial constraints over the reals. □

2.3 Boolean Constraint Satisfaction Problems

Boolean CSP's form a special case of numeric CSP's in which the variables range over the domain [0..1] and the constraints are expressed by means of Boolean formulas.

Example 2.6 *Full adder circuit.*

This example deals with circuits built out of the so-called AND, OR and XOR gates. These gates generate an output value given two input values. The possible values are drawn from [0..1], so we deal here with bits or, alternatively, Boolean variables. The behaviour of these gates is defined by the following tables:

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

So the AND gate corresponds to the conjunction, the OR gate to the disjunction and the XOR gate to the exclusive disjunction. Therefore the circuits built out these gates can be naturally represented by equations between Boolean expressions involving conjunction, written as $\wedge$, disjunction, written as $\vee$, and exclusive disjunction, written as $\oplus$.

In particular, the circuit depicted in Figure 2.3 can be represented by the following two formulas:

$$(i_1 \oplus i_2) \oplus i_3 = o_1,$$

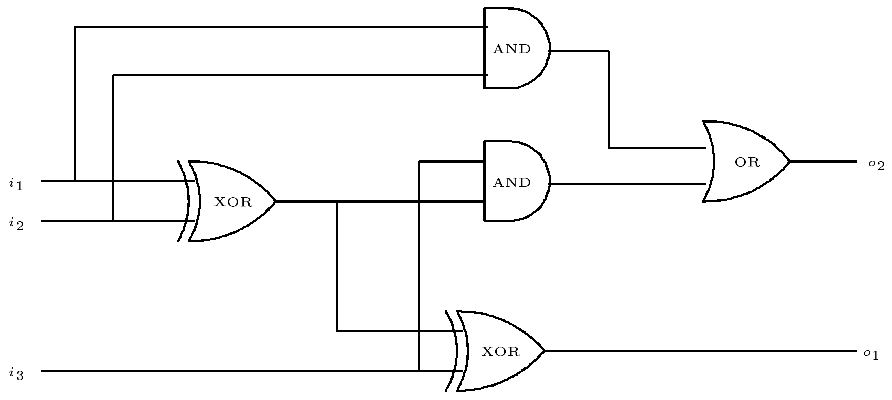

Figure 2.3: Full adder circuit

$$(i_1 \wedge i_2) \vee (i_3 \wedge (i_1 \oplus i_2)) = o_2.$$

This circuit is called *full adder* as it computes the binary sum $i_1 + i_2 + i_3$ in the binary word $o_2 o_1$. For example $0 + 1 + 1$ yields 10. To verify the correctness of this circuit it suffices to use the tables above and to calculate the outputs o_1, o_2 for all combinations of the inputs i_1, i_2, i_3.

The fact that we represent this circuit as a Boolean CSP, so as relations between the Boolean variables i_1, i_2, i_3, o_1 and o_2 determined by the above equations instead of as a function from (i_1, i_2, i_3) to (o_1, o_2) allow us to draw in a systematic way more complex conclusions such as that $i_3 = 0$ and $o_2 = 1$ implies that $i_1 = 1, i_2 = 1$ and $o_1 = 0$. $\Box$

Example 2.7 *n Queens Problem, again.*

When discussing in Example 2.2 the n queens problem we chose a representation involving n variables, each associated with one row. A different natural representation involves n^2 Boolean variables $x_{i,j}$, where $i \in [1..n]$ and $j \in [1..n]$, each of them representing one field of the chess board. The appropriate constraints can then be written as Boolean constraints.

To this end we introduce the following abbreviation. Given k Boolean expressions $s_1, \ldots, s_k$ we denote by $one(s_1, \ldots, s_k)$ the Boolean expression that states that exactly one of the expressions $s_1, \ldots, s_k$ is true. So $one(s_1, \ldots, s_k)$ is a disjunction of k expressions, each of them being a conjunction of the form $\neg s_1 \wedge \ldots \neg s_{i-1} \wedge s_i \wedge \neg s_{i+1} \ldots \wedge \neg s_n$, where $i \in [1..k]$.

Then the following constraints formalize the problem:

$$one(x_{i,1}, \ldots, x_{i,n}),$$

for $i \in [1..n]$ (exactly one queen per row),

$$one(x_{1,i}, \ldots, x_{n,i}),$$

for $i \in [1..n]$ (exactly one queen per column),

$$x_{i,j} \rightarrow \neg x_{k,\ell}$$

for $i, j, k, \ell \in [1..n]$ such that $|i \perp k| = |j \perp \ell|$ (at most one queen per diagonal). $\Box$

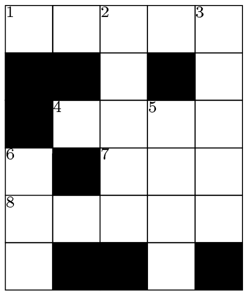

Figure 2.4: A crossword grid

2.4 Symbolic Constraint Satisfaction Problems

By a symbolic constraint satisfaction problem we mean a CSP the variables of which range over non-numeric domains.

Example 2.8 *Crossword Puzzle.*

Consider the crossword grid of Figure 2.4 and suppose that we are to fill it with the words from the following list using each word at most once:

- HOSES, LASER, SAILS, SHEET, STEER,
- HEEL, HIKE, KEEL, KNOT, LINE,
- AFT, ALE, EEL, LEE, TIE.

This problem can be easily formulated as a CSP as follows. First, associate with each position $i \in [1..8]$ in this grid a variable. Then associate with each variable the domain that consists of the set of words of that can be used to fill this position. For example, position 6 needs to be filled with a three letter word, so the domain of the variable associated with position 6 consists of the above set of five 3 letter words.

Finally, there are two types of constraints. The first group deals with the restrictions arising from the fact the words that cross share a letter. For example, the crossing of the positions 1 and 2 contributes the following constraint:

$$\begin{aligned} C_{1,2} := \{&(\text{HOSES, SAILS}), (\text{HOSES, SHEET}), (\text{HOSES, STEER}),\\ &(\text{LASER, SAILS}), (\text{LASER, SHEET}), (\text{LASER, STEER})\}\ . \end{aligned}$$

This constraint formalizes the fact that the third letter of position 1 needs to be the same as the first letter of position 2. In total there are 12 of such constraints.

The other group of constraints deals with the restriction that each word is to be used once. This means that we impose the disequality constraints between each pair of the variables associated with the positions of the same length. We have thus disequality constraints between the variables associated with

- the positions of length five, so 1,2,3 and 8,

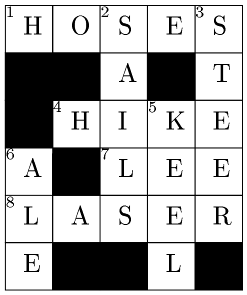

Figure 2.5: A solution to the crossword puzzle

- the positions of length four, so 4 and 5,
- the positions of length three, so 6 and 7.

In total there are 8 of such constraints.

A unique solution to this CSP is depicted in Figure 2.5. □

Example 2.9 *Graph Colouring Problem.*

Here the task is to assign to each node of a finite graph a colour in such a way that no two adjacent nodes have the same colour. Such an assignment is called a *colouring* of the graph.

To formulate this problem as a CSP it suffices to associate with each node of the graph a variable the domain of which consists of the set of assumed colours and to assume the inequality constraints for each pair of variables representing a pair of adjacent nodes.

In an optimization variant of this problem we are asked to find a colouring of the graph involving a minimal number of colours. This number is called the *chromatic number* of the graph.

The problem of finding the chromatic number of a graph has several applications, in particular in the fields of scheduling and compiler optimization.

□

Example 2.10 *Temporal Reasoning.*

Consider the following problem.

> The meeting ran non-stop the whole day. Each person stayed at the meeting for a continuous period of time. The meeting began while Mr Jones was present and finished while Ms White was present. Director Smith was also present but either he arrived after Jones had left or they both left at the same time. Mr Brown talked to Ms White in presence of Smith. Could possibly Jones and White have met during this meeting?

To properly analyse such problems we are naturally led to an abstract analysis of activities that take time, such as being present during the meeting, driving to work, taking lunch break, filling in a form, receiving a phone call, etc. In what follows we call such activities *events*.

If we only take into account the duration of these events, then we can identify each such event with a closed non-empty interval of the real line. With such an identification we end up with 13 possible *temporal relations* between a pair of events. They are presented in Figure 2.6.

Intuitively, these relations arise when we consider two intervals, `A` and `B`, and keep moving the interval `A` from left to right and record all its possible relative positions w.r.t. `B`, taking into account their possibly different relative sizes.

Let *TEMP* denote the set formed by the 13 possibilities of Figure 2.6. In what follows we provide two representations of CSP's dealing with temporal reasoning. The first one is conceptually simpler but it involves infinite domains. The second one is at first appearence more involved but it has the advantage that the domains are finite and directly correspond to the set *TEMP*.

First representation.

In this representation the variables are events. Their domains reflect the view that events are identified with closed non-empty intervals of real line. So each domain consists of the set of such intervals, that is the set

$$D := \{(a,b) \mid a,b \in \mathcal{R},\ a < b\},$$

where $\mathcal{R}$ is the set of all reals and each pair (a,b) represents the closed interval $[a,b]$ of reals.

Next, each of the introduced 13 temporal relations is represented as a binary constraint in the way that reflects the intended meaning. For example, the `overlaps` relation is represented as the following subset of $D \times D$:

$$[\![\texttt{overlaps}]\!] := \{((a_{begin}, a_{end}), (b_{begin}, b_{end})) \mid a_{begin} < b_{begin}, b_{begin} < a_{end}, a_{end} < b_{end}\}.$$

Finally, arbitrary constraints are set-theoretic unions of the elementary binary constraints that represent the elementary temporal relations.

Let us illustrate now this representation by formalizing as a CSP the problem we began with. First, we identify the relevant events and associate with each of them a variable. In total, we consider the following five events and variables:

event	variable
the duration of the meeting	M
the period Jones was present	J
the period Brown was present	B
the period Smith was present	S
the period White was present	W

Next, we define the appropriate constraints. The first three formalize information concerning the presence of each person during the meeting. For brevity we denote here the union of all elementary constraints excluding $[\![\texttt{before}]\!]$, $[\![\texttt{after}]\!]$, $[\![\texttt{meets}]\!]$ and $[\![\texttt{met-by}]\!]$ as $[\![\texttt{REAL-OVERLAP}]\!]$. These three constraints are:

$$([\![\texttt{overlaps}]\!] \cup [\![\texttt{contains}]\!] \cup [\![\texttt{finished-by}]\!])(J, M),$$

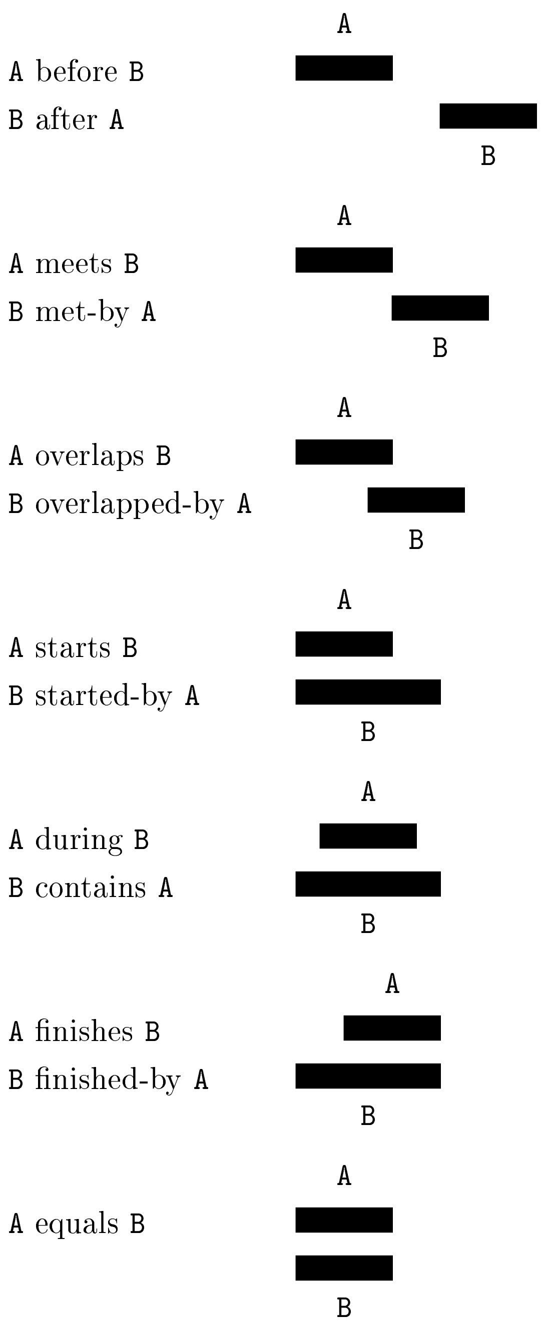

Figure 2.6: Thirteen Temporal Relations

$(\llbracket \texttt{overlaps} \rrbracket \cup \llbracket \texttt{starts} \rrbracket \cup \llbracket \texttt{during} \rrbracket)(M, W)$,

$\llbracket \texttt{REAL-OVERLAP} \rrbracket(M, S)$.

Note that the first constraint formalizes the fact that

- J started strictly earlier than M started,
- M started before J finished.

In turn, the second constraint formalizes the fact that

- W started before M finished,
- M finished strictly earlier than W finished.

Finally, the constraint $\llbracket \texttt{REAL-OVERLAP} \rrbracket(M, S)$ formalizes the fact that M and S "truly" overlap in time, that is, share some time interval of positive length. Note that we do not postulate here

$\llbracket \texttt{REAL-OVERLAP} \rrbracket(M, B)$

because from the problem formulation it is not clear whether Brown was actually present during the meeting.

Additionally, the following constraints formalize information concerning the relative presence of the persons in question:

$(\llbracket \texttt{before} \rrbracket \cup \llbracket \texttt{finishes} \rrbracket \cup \llbracket \texttt{finished-by} \rrbracket \cup \llbracket \texttt{equal} \rrbracket)(J, S)$,

$\llbracket \texttt{REAL-OVERLAP} \rrbracket(B, S)$,

$\llbracket \texttt{REAL-OVERLAP} \rrbracket(B, W)$,

$\llbracket \texttt{REAL-OVERLAP} \rrbracket(S, W)$.

The question "Could possibly Jones and White have met during this meeting?" can now be formalized as a problem whether for some solution to this CSP $r(J, W)$ holds for some elementary constraint r different from $\llbracket \texttt{before} \rrbracket$ and $\llbracket \texttt{after} \rrbracket$. In other words, is it true that the above CSP augmented by the constraint $\llbracket \texttt{ENCOUNTER} \rrbracket(J, W)$, where $\llbracket \texttt{ENCOUNTER} \rrbracket$ denotes the union of all elementary constraints excluding $\llbracket \texttt{before} \rrbracket$ and $\llbracket \texttt{after} \rrbracket$, is consistent.

Second representation.

In the first representation the domains, and thus the domain expressions, were uniquely fixed and we had to determine the constraints. In the second representation we first determine the domain expressions that then uniquely determine the constraints.

In this representation a variable is associated with each ordered pair of events. Each domain of such a variable is a subset of the set *TEMP*. All constraints are ternary and are uniquely determined by the domains.

More specifically, consider three events, `A, B` and `C` and suppose that we know the temporal relations between the pairs `A` and `B`, and `B` and `C`. The question is what is the temporal relation between `A` and `C`. For example if `A overlaps B` and `B` is `before C`, then `A` is `before C`. To answer this question we have to examine 169 possibilities. They can be represented by a table that is omitted here.

Consider now all legally possible triples (t_1, t_2, t_3) of temporal relations such that t_1 is the temporal relation between `A` and `B`, t_2 the temporal relation between `B` and `C` and t_3 the temporal relation between `A` and `C`. The set of these triples forms a subset of TEMP^3. Denote it by C_3.

The just mentioned table allows us to compute C_3. For example, we already noticed that

$$(\texttt{overlaps}, \texttt{before}, \texttt{before}) \in C_3.$$

since `A overlaps B` and `B` is `before C` implies that `A` is `before C`.

By a *disjunctive temporal relation* we mean now a disjunction of temporal relations. For example `before` $\vee$ `meets` is a disjunctive temporal relation. The disjunctive temporal relations allow us to model the situations in which the temporal dependencies between the events are only partially known.

Note that each disjunctive temporal relation between the events `A` and `B` uniquely determines the disjunctive temporal relation between `B` and `A`. For example if the former is `before` $\vee$ `meets`, then the latter is `after` $\vee$ `met-by`. So without loss of information we can associate from now on only one disjunctive temporal relation with each pair of events.

We can now define the CSP's. Assume n events $e_1, \ldots, e_n$ with $n > 2$. We consider $\frac{(n-1)n}{2}$ domain expressions, each of the form $x_{i,j} \in D_{i,j}$ where $i, j \in [1..n]$ with $i < j$ and $D_{i,j} \subseteq \mathit{TEMP}$. The intention is that the variable $x_{i,j}$ describes the disjunctive temporal relation associated with the events e_i and e_j.

Finally, we define the constraints. Each constraint links an ordered triple of events. So in total we have $\frac{(n-2)(n-1)n}{6}$ constraints. For $i, j, k \in [1..n]$ such that $i < j < k$ we define the constraint $C_{i,j,k}$ by setting

$$C_{i,j,k} := C_3 \cap (D_{i,j} \times D_{j,k} \times D_{i,k}).$$

So $C_{i,j,k}$ describes the possible entries in C_3 determined by the domains of the variables $x_{i,j}, x_{j,k}$ and $x_{i,k}$.

This completes the description of the CSP's. Because each constraint is determined by the corresponding triple of variable domains, each such CSP is uniquely determined by its domain expressions. Therefore when defining such CSP's it is sufficient to define the appropriate domain expressions.

Let us return now to the problem we started with. It can be formulated as a CSP as follows. We have five events, `M, J, B, S, W`. Here `M` stands for "the duration of the meeting" , `J` stands for "the period Jones was present", and similarly with the events `S` (for Smith), `B` (for Brown) and `W` (for White).

Next, we order the events in some arbitrary way, say

```
J, M, B, S, W.
```

We have in total ten domain expressions, each associated with an ordered pair of events. For brevity we delete below the domain expressions of the form $x \in \mathit{TEMP}$.

Analogously as in the first representation we formalize the fact that two events "truly" overlap in time by excluding from *TEMP* the relations `before`, `after`, `meets` and `met-by`, so by using the domain

$\mathit{REAL\text{-}OVERLAP} := \mathit{TEMP} \perp \{\texttt{before, after, meets, met-by}\}$.

Then the following domain expressions deal with the presence of each person during the meeting:

$x_{\texttt{J,M}} \in \{\texttt{overlaps, contains, finished-by}\}$,

$x_{\texttt{M,W}} \in \{\texttt{overlaps, starts, during}\}$,

$x_{\texttt{M,S}} \in \mathit{REAL\text{-}OVERLAP}$,

and the following domain expressions formalize information concerning the relative presence of the persons in question:

$x_{\texttt{J,S}} \in \{\texttt{before, finishes, finished-by, equal}\}$,

$x_{\texttt{B,S}} \in \mathit{REAL\text{-}OVERLAP}$,

$x_{\texttt{B,W}} \in \mathit{REAL\text{-}OVERLAP}$,

$x_{\texttt{S,W}} \in \mathit{REAL\text{-}OVERLAP}$.

The final question can then be phrased as a question whether in some solution to this CSP we have $x_{\texttt{J,W}} = r$ for some $r \in \mathit{TEMP} \perp \{\texttt{before, after}\}$. In other words, is true that the above CSP augmented by the domain expression

$x_{\texttt{J,W}} \in \mathit{TEMP} \perp \{\texttt{before, after}\}$

is consistent.

Note that in both representations we assumed that events "being present during the meeting" and "talking" are of positive duration. In contrast, we do allow that meeting somebody can be limited to just one point in time. □

2.5 Bibliographic Remarks

Most of the CSP's discussed in this chapter are classics in the field. In particular, the representation of the *SEND + MORE = MONEY* and of the *Zebra* puzzles as a CSP is discussed in Van Hentenryck (1989). According to Russell & Norvig (1995) the eight queens problem was originally published anonymously in the German chess magazine *Schach* in 1848 and its generalization to n queens problem in Netto (1901). Falkowski & Schmitz (1986) show how to construct a solution to this problem for arbitrary $n > 3$.

The representation of spreadsheets as CSP's is taken from Winston (1992) and the crossword puzzle discussed in Example 2.8 is from Mackworth (1992). The 13 temporal relations presented in Figure 2.6 and the ensuing temporal CSP's were introduced in Allen (1983).

F.Bacchus & van Beek (1998) discuss the tradeoffs involved in the conversion of non-binary CSP's to binary ones and provide pointers to the earlier literature on this subject.

References

Allen, J. (1983), 'Maintaining knowledge about temporal intervals', *Communications of ACM* **26**(11), 832–843.

Falkowski, B.-J. J. & Schmitz, L. (1986), 'A note on the queens' problem', *Information Processing Letters* **23**(1), 39–46.

F.Bacchus & van Beek, P. (1998), On the conversion between non-binary and binary constraint satisfaction problems, *in* 'AAAI-98: Proceedings of the 15th National Conference on Artificial Intelligence', AAAI Press, Menlo Park.

Mackworth, A. (1992), Constraint satisfaction, *in* S. C. Shapiro, ed., 'Encyclopedia of Artificial Intelligence', Wiley, pp. 285–293. Volume 1.

Netto, E. (1901), *Lehrbuch der Combinatorik*, Teubner, Stuttgart.

Russell, S. & Norvig, P. (1995), *Artifical Intelligence: A Modern Approach,*, Prentice-Hall, Englewood Cliffs, NJ.

Van Hentenryck, P. (1989), *Constraint Satisfaction in Logic Programming*, Logic Programming Series, MIT Press, Cambridge, MA.

Winston, P. (1992), *Artificial Intelligence*, third edn, Addison-Wesley, Reading, Massachusetts.

Chapter 3

Constraint Programming in a Nutshell

At this stage it is useful to get a general feeling what constraint programming is about. It consists of a number of techniques that can be explained informally in a rather simple way. The aim of this chapter is to provide an intuitive introduction to these techniques. In the subsequent chapters we shall discuss these techniques in a more detailed and precise way.

3.1 A General Framework

We begin by formulating a general framework for constraint programming that we shall use to explain its specific aspects.

First, we formulate our initial problem as a CSP. This in itself can be a non-trivial problem. In particular, at this stage we have to take decisions concerning the choice of variables, domains and constraints. This stage of constraint programming is called *modelling* and in contrast to programming in other programming styles it is more time consuming and more involved. Modelling is more an art than science and a number of rules of thumb and various heuristics are useful at this stage.

Subsequently, we apply to the formulated CSP the following procedure:

```
SOLVE:

WHILE NOT DONE DO
  PREPROCESS;
  CONSTRAINT PROPAGATION;
  IF HAPPY
    THEN
      DONE:=true;
    ELSE
      SPLIT;
      PROCEED BY CASES
  END
```

END

where DONE is initially **false** and where PROCEED BY CASES leads to a recursive invocation of SOLVE for each newly formed CSP.

The SOLVE procedure represents the basic loop of constraint programming. In what follows we briefly explain, in an informal way, the meaning of all the subsidiary procedures used in SOLVE. As the notion of constraint propagation is central to constraint programming we defer the discussion of it to the end of the section. At this stage it suffices to know that constraint propagation transforms a given CSP into another one that is equivalent to it.

3.1.1 PREPROCESS

The aim of this procedure is to bring the given CSP into a desired syntactic form. The resulting CSP should be equivalent to the original one w.r.t. to the original set of variables.

To illustrate it we consider two simple examples.

First, consider Boolean constraints as discussed in Section 2.3. Most of the procedures that deal with them assume that these constraints are in a specific syntactic form. A well-known example is *disjunctive normal form* according to which the Boolean constraint is a disjunction of formulas each of which is a conjunction of literals, where in turn a literal is a Boolean variable or its negation. For example, the Boolean constraint

$$(x \wedge \neg y) \vee (\neg x \wedge y \wedge z) \vee (\neg x \wedge \neg z)$$

is in the disjunctive normal form.

In this case the preprocessing consists of of rules that transform an arbitrary formula to an equivalent one that is in disjunctive normal form.

As a second example consider constraints on reals as discussed in Section 2.2. Often, specific procedures that deal with such constraints assume that in each constraint each variable appears at most once. In this case the preprocessing consists of transforming each constraint into such a form by introducing auxiliary variables.

For instance, given an equation

$$ax^7 + bx^5y + cy^{10} = 0$$

we employ an auxiliary variable y and replace it by two equations,

$$ax^7 + z + cy^{10} = 0$$

and

$$bx^5y = z.$$

3.1.2 Happy

Informally Happy means that the goal set for the initial CSP has been achieved. What goal it is depends of course on the applications. The following contingencies are most common:

- a solution has been found,
- all solutions have been found,
- a "normal form" has been reached from which it is straightforward to generate all solutions,
- no solution was found,
- best solution w.r.t. some quality measure was found,
- all best solutions w.r.t. some quality measure were found,
- (in the case of constraints on reals) all interval domains are reduced to sizes smaller than some fixed in advance ϵ.

In general Happy can be viewed as a test applied to the current CSP in which some additional parameters are also taken into account.

3.1.3 Split

If after termination of the constraint propagation the current CSP is not in the desired form, that is the test Happy fails, this CSP is split into two (or more) CSP's the union of which is equivalent to the current CSP. In general, such a split is obtained either by splitting a domain or by splitting a constraint.

In the following two examples a split of a domain is represented by a rule that transforms a domain expression into two domain expressions separated by means of the "|" symbol.

- Labelling.

  Assume that the domain D is non-empty and finite. The following rule can then be used:

  $$\frac{x \in D}{x \in \{a\} \mid x \in D \perp \{a\}}$$

  where $a \in D$.

- Bisection.

  Assume that the domain is a non-empty interval of reals, written as $[a..b]$. We can then employ the following rule:

  $$\frac{x \in [a..b]}{x \in [a..\frac{a+b}{2}] \mid x \in [\frac{a+b}{2}..b]}$$

In turn, the following two examples, also written as rules, illustrate a split of a constraint.

- Disjunctive constraints.

  Suppose that the constraint is a Boolean disjunction. The constituents of this disjunction can be arbitrary constraints. We can then apply the following rule:

  $$\frac{C_1 \vee C_2}{C_1 \mid C_2}$$

- Constraints in "compound" form.

  The idea is that such constraints are split into syntactically simpler compounds that can be dealt with directly. Suppose for example that we know how to deal directly with polynomial equations on reals and that the constraint in question is $p(\bar{x}) = a$, where $p(\bar{x})$ is a polynomial in the variables $\bar{x}$ and a is a constant. Then we can use the following rule:

  $$\frac{|p(\bar{x}) = a|}{p(\bar{x}) = a \mid p(\bar{x}) = \perp a}$$

Each split of a domain or of a constraint leads to a replacement of the current CSP by two CSP's that differ from the current one in that the split domain, respectively the split constraint, is replaced by one of the constituents.

For instance, the above rule dealing with labelling leads to the replacement of a CSP

$$\langle \mathcal{C} \ ; \ \mathcal{DE}, x \in D \rangle$$

by two CSP's,

$$\langle \mathcal{C} \ ; \ \mathcal{DE}, x \in \{a\} \rangle$$

and

$$\langle \mathcal{C} \ ; \ \mathcal{DE}, x \in D \perp \{a\} \rangle.$$

3.1.4 Proceed by Cases

The Split procedure yields two new CSP's. They are then dealt with by means of the Proceed by Cases procedure.

The order in which these new CSP's are considered depends on the adopted *search technique*. In general, due to the repeated use of the Split procedure a binary tree of CSP's is generated. The purpose of the Proceed by Cases procedure is to traverse this tree in a specific order and, if needed, to update the current CSP with some newly gathered information (in the case of search for the best solution).

Two most known of these techniques are backtracking and —when searching for the best solution— branch and bound.

Informally, given a finite tree, the *backtracking search* starts at the root of the tree and proceeds by descending to its first descendant. This process continues as long as a node is not a leaf. If a leaf is encountered the search proceeds by moving back to the parent node of the leaf. Then the next descendant, if any, of this parent node is selected. This process continues until the control is back at the root node and all of its descendants have been visited. The picture in Figure 3.1 depicts the backtracking search on a simple tree.

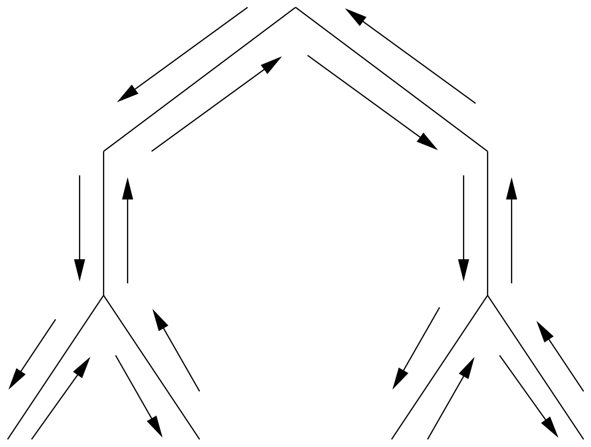

Figure 3.1: Backtracking search

The *branch and bound search* is a modification of the backtracking search that takes into account the value of some objective function defined in terms of some variables. This value is initialized to $\perp\infty$ if one searches for maximum and to ∞ if one searches for minimum. At each stage the currently best value of the objective function is maintained. This is used to prevent descending the nodes for which even in the best circumstances the value of the objective function cannot improve upon the currently best known value. Once all variables of the objective function have a known value the currently best known value is updated.

Backtracking and branch and bound, when combined with appropriate instances of the CONSTRAINT PROPAGATION procedure form more complex forms of search methods that are specific for constraint programming. Two best known techniques are *forward checking* and *look ahead*.

All these methods will be explained in the chapter on search methods.

3.1.5 CONSTRAINT PROPAGATION

At this stage let us return to the CONSTRAINT PROPAGATION procedure. In general, this procedure replaces a given CSP by a “simpler” one, yet equivalent. The idea is that such a replacement, if efficient, is profitable, since the subsequent search resulting from the repeated calls of the SPLIT and PROCEED BY CASES procedures is then performed on a smaller search space.

What “simpler” denotes depends on the applications. Typically, it means that the domains and/or constraints become smaller. The constraint propagation is performed by repeatedly reducing domains and/or reducing constraints while maintaining equivalence.

Let us consider some examples. We again use here proof rules but now they represent a replacement of a CSP by another one. The first two examples deal with domain reduction.

- Arbitrary CSP's.

  Consider a constraint C. Choose a variable x of it and perform the following operation on its domain D:

  remove from D all values that do not participate in a solution to C.

  The idea is that the removed values cannot be present in any solution to the considered CSP. We call this operation *projection of* C *on* D.

- Linear inequalities on integers.

  Assume that the domains are non-empty intervals of integers, written as $[a..b]$, and the constraints are linear inequalities of the form $x < y$. Then we can apply the following rule:

  $$\frac{\langle x < y\ ;\ x \in [l_x..h_x], y \in [l_y..h_y]\rangle}{\langle x < y\ ;\ x \in [l_x..min(h_x, h_y \perp 1)], y \in [max(l_y, h_x + 1), h_x]\rangle}$$

  The idea is that $x < y$ and $y \leq h_y$ imply $x \leq h_y \perp 1$. This in conjunction with $x \leq h_x$ implies that $x \leq min(h_x, h_y \perp 1)$, and analogously with the variable y.

  In particular, this rule allows us to conclude from the CSP

  $$\langle x < y\ ;\ x \in [50..200], y \in [0..100]\rangle$$

  the CSP

  $$\langle x < y\ ;\ x \in [50..99], y \in [51..100]\rangle$$

  the domains of which are clearly smaller.

Next, consider the reduction of constraints. We illustrate it by means of two examples. In each of them a new constraint is introduced. This can be viewed as a reduction of a constraint. Namely, an introduction of a new constraint, say on the variables $\bar{x}$, leads to an addition of a new conjunct to the conjunction of used constraints on $\bar{x}$ (if there is none, we can assume that a "universal" true constraint on $\bar{x}$ is present). The new conjunction is then semantically a subset of the old one.

- Transitivity.

  Consider the following rule that invokes the transitivity of the $<$ relation on reals:

  $$\frac{\langle x < y, y < z\ ;\ \mathcal{DE}\rangle}{\langle x < y, y < z, x < z\ ;\ \mathcal{DE}\rangle}$$

  This rule introduces a new constraint, $x < z$.

- Resolution rule.

  This rule deals with clauses, that is, disjunctions of literals. (Recall from Subsection 3.1.1 that a literal is a Boolean variable or its negation.) Let C_1 and C_2 be clauses, L a literal and $\bar{L}$ the literal opposite to L, that is $\overline{\neg x} = x$ and $\bar{x} = \neg x$.

  $$\frac{\langle C_1 \vee L, C_2 \vee \bar{L} \ ; \ \mathcal{DE}\rangle}{\langle C_1 \vee L, C_2 \vee \bar{L}, C_1 \vee C_2 \ ; \ \mathcal{DE}\rangle}$$

  This rule introduces a new constraint, the clause $C_1 \vee C_2$.

In general, various general techniques drawn from the fields of linear algebra, linear programming, integer programming, and automated theorem proving can be explained as a reduction of constraints.

The above examples dealt with atomic reduction steps in which either a domain or a constraint is reduced. The constraint propagation algorithms deal with the scheduling of such atomic reduction steps. In particular, these algorithms should avoid useless applications of the atomic reduction steps. The stopping criterion is a *local consistency notion*, that is, a constraint propagation algorithm terminates once the final CSP satisfies a specific local consistency notion.

The concept of local consistency is crucial for the theory of constraint programming. In the literature a plethora of such notions have been introduced. We illustrate them here by means of two examples that correspond with the two domain reduction examples presented above.

Consider an arbitrary CSP and suppose that we select for each constraint and each variable of it the projection operation discussed above. The local consistency notion that corresponds with these atomic domain reduction steps is the following one:

> For every constraint C and every variable x of it each value in the domain of x participates in a solution to C.

This property is called *arc consistency* and is perhaps the most popular local consistency notion.

Consider now a CSP the domains of which are intervals of integers and the constraints of which are linear inequalities of the form $x < y$. Suppose that we select for each such constraint the domain reduction explained above. Then the local consistency notion that corresponds with these atomic domain reduction steps is the following one:

> For every constraint C and every variable x of it each bound of the domain of x participates in a solution to C.

This property is called *bound consistency* and is more efficient to test for linear constraints on integer intervals than arc consistency.

It is important to realize that in general a locally consistent CSP does not need to be consistent. For example the CSP

$$\langle x \neq y, y \neq z, z \neq x \ ; \ x \in \{0,1\}, y \in \{0,1\}, z \in \{0,1\}\rangle$$

is easily seen to be arc consistent but it is not consistent.

To summarize, each constraint propagation algorithm reduces a given CSP to an equivalent one that satisfies some local consistency notion. Which local consistency notion is used depends on the type of CSP's considered.

3.2 Example: Boolean Constraints

Let us illustrate now the above discussion by means of an example. It will clarify what choices one needs to make when solving specific CSP's. These choices will be reflected in specific decisions concerning the subsidiary procedures of the generic SOLVE procedure.

Suppose that we wish to find all solutions to a given Boolean constraint satisfaction problem. Then we could consider the following selection of the procedures discussed above.

PREPROCESS

We would like to bring each Boolean constraint to one of the following forms:

- $x = y$,
- $\neg x = y$,
- $x \wedge y = z$,
- $x \vee y = z$.

Consequently, for PREPROCESS we choose transformations rules that transform each Boolean constraint into a set of constraints in the above form. An example of such a transformation rule is

$$\frac{x \wedge \phi = z}{x \wedge y = z, \phi = y}$$

where x, y, z are Boolean variables and where ϕ is a Boolean expression that does not contain y.

HAPPY

We choose the test: all solutions have been found.

SPLIT

We use the labelling rule discussed in Subsection 3.1.3. We need to be, however, more precise as this rule is parametrized by a variable the choice of which is of relevance, and by a value the choice of which is not relevant here.

We use here a well-known heuristic according to which the variable that occurs in the largest number of constraints (the "most constrained variable") is chosen first.

PROCEED BY CASES

As we want to find all solutions we choose here the backtracking search.

Constraint Propagation

Finally, we determine the actions of the constraint propagation algorithm. We do this by choosing specific domain reduction steps. They are based on the simple observation that for the Boolean constraints in one of the forms considered in the definition of the Preprocess procedure, if the values of some variables are determined then values of some other variables are determined, as well. For example, for the constraint $x \wedge y = z$, if we know that z is 1 (i.e., true), then we can conclude that both x and y are 1. This is expressed by means of the following rule:

$$\frac{\langle x \wedge y = z \ ; \ x \in D_x, y \in D_y, z \in \{1\}\rangle}{\langle \ ; \ x \in D_x \cap \{1\}, y \in D_y \cap \{1\}, z \in \{1\}\rangle}$$

where the absence of the constraint $x \wedge y = z$ in the conclusion indicates that this constraint is solved.

We can abbreviate it to a more suggestive form, namely

$$x \wedge y = z, z = 1 \rightarrow x = 1, y = 1.$$

In total there are six such rules for the constraint $x \wedge y = z$. We adopt similar rules for other constraints considered in the definition of the Preprocess procedure.

This completes the description of a sample procedure using which we can find all solutions to a Boolean constraint satisfaction problem.

3.3 Concluding Remarks

Many elements of constraint programming have been by necessity omitted in the above discussion. For example, we devoted to constraint propagation algorithms just one paragraph, hardly said anything about the branch and bound method, and did not even mention many important local consistency notions, like the ones that correspond to the examples of the constraint reduction discussed in Subsection 3.1.5.

Still we hope that this brief introduction to the specifics of constraint programming sheds some light on the subject and that it provides some insights into the choices that need to be made when trying to solve a problem by means of constraint programming techniques.

Chapter 4

Some Complete Constraint Solvers

By a *constraint solver* we mean any procedure that transforms a CSP into an equivalent one. If from the final CSP it is straightforward to generate all solutions to it or determine that no solution exists, we say that the constraint solver is *complete* and otherwise we say that the constraint solver is *incomplete*. This definition is admittedly imprecise, but we shall make it formal.

For a number of domains and constraints complete constraint solvers were developed. The aim of this chapter is to illustrate such solvers by means of two well-known examples.

4.1 Unification

The first complete solver we shall discuss deals with solving term equations. This problem is known as unification.

We begin by defining in the next subsection a language of terms. Then in Subsection 4.1.2 we introduce the substitutions and in Subsection 4.1.3 unifiers and most general unifiers. Next, in Subsection 4.1.4 we study a unification algorithm due to Martelli and Montanari. Finally, in Subsection 4.1.5 we discuss the efficiency issues.

4.1.1 Terms

An *alphabet* consists of the following disjoint classes of symbols:

- *variables*,
- *function symbols*,
- *parentheses*, which are: (and),
- *comma*, that is: , .

We assume that the set of variables is infinite and fixed. In contrast, the set of function symbols may vary and in particular may be empty.

Each function symbol has a fixed *arity*, that is the number of arguments associated with it. 0-ary function symbols are called

- *constants*, and are denoted by $a, b, c, d, \ldots$.

We denote function symbols of positive arity by $f, g, h, k, l, \ldots$.
Terms are defined inductively as follows:

- a variable is a term,
- if f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

In particular every constant is a term. Terms are denoted by $s, t, u, w, \ldots$.

The set of terms is thus determined by the set of function symbols of the considered alphabet.

In what follows we denote by $Var(t)$ the set of variables occurring in t.

4.1.2 Substitutions

Consider now a fixed alphabet and consequently a fixed set of terms. A *substitution* is a finite mapping from variables to terms which assigns to each variable x in its domain a term t different from x. We write it as

$$\{x_1/t_1, \ldots, x_n/t_n\}$$

where

- $x_1, \ldots, x_n$ are different variables,
- $t_1, \ldots, t_n$ are terms,
- for $i \in [1, n]$, $x_i \neq t_i$.

Informally, it is to be read: the variables $x_1, \ldots, x_n$ are *simultaneously replaced by* $t_1, \ldots, t_n$, respectively. A pair x_i/t_i is called a *binding*. When $n = 0$, the mapping becomes the empty mapping. The resulting substitution is then called *empty substitution* and is denoted by ϵ.

Further, given a substitution $\theta := \{x_1/t_1, \ldots, x_n/t_n\}$, we denote by $Dom(\theta)$ the set of variables $\{x_1, \ldots, x_n\}$ and by $Range(\theta)$ the set of terms $\{t_1, \ldots, t_n\}$. We now define the result of *applying a substitution θ to a term s*, written as $s\theta$, as the result of the simultaneous replacement of each occurrence in s of a variable from $Dom(\theta)$ by the corresponding term in $Range(\theta)$.

Example 4.1 Consider a language allowing us to build arithmetic expressions in prefix form. It contains two binary function symbols, "+" and "·" and infinitely many constants: 0, 1, ... Then $s := +(\cdot(x, 7), \cdot(4, y))$ is a term and for the substitution $\theta := \{x/0, y/+(z, 2)\}$ we have

$$s\theta = +(\cdot(0, 7), \cdot(4, +(z, 2))).$$

□

Next, we define the composition of two substitutions.

Definition 4.2 Let θ and η be substitutions. Their *composition*, written as $\theta\eta$, is defined as follows. We put for a variable x

$$(\theta\eta)(x) := (x\theta)\eta.$$

In other words, $\theta\eta$ assigns to a variable x the term obtained by applying the substitution η to the term $x\theta$. Clearly, for $x \notin Dom(\theta) \cup Dom(\eta)$ we have $(\theta\eta)(x) = x$, so $\theta\eta$ is a finite mapping from variables to terms, i.e. it uniquely identifies a substitution. □

For example, for $\theta = \{u/z, x/3, y/f(x,1)\}$ and $\eta = \{x/4, z/u\}$ we can check that

$$\theta\eta = \{x/3, y/f(4,1), z/u\}.$$

The following simple lemma will be needed in Subsection 4.1.4.

Lemma 4.3 (Binding) *For a variable x and a term t, $x\theta = t\theta$ iff $\theta = \{x/t\}\theta$.*

Proof. First, note that $x\{x/t\}\theta = t\theta$, so $x\theta = t\theta$ iff $x\theta = x\{x/t\}\theta$. Moreover, for $y \neq x$ we have $y\{x/t\} = y$, so $y\theta = y\{x/t\}\theta$ holds always. Thus $x\theta = x\{x/t\}\theta$ iff for all variables z we have $z\theta = z\{x/t\}\theta$. These two equivalences establish the claim. □

Next, we introduce the following notion.

Definition 4.4 Let θ and τ be substitutions. We say that θ is *more general than* τ if for some substitution η we have $\tau = \theta\eta$. □

For example, $\{x/y\}$ is more general than $\{x/a, y/a\}$ since $\{x/y\}\{y/a\} = \{x/a, y/a\}$.

Thus θ is more general than τ if τ can be obtained from θ by applying to it some substitution η. Since η can be chosen to be the empty substitution ϵ, we conclude that every substitution is more general than itself.

4.1.3 Unifiers and Mgus

A term equation $s = t$ can be viewed as a constraint on the sequence of variables $x_1, \ldots, x_n$ that occur in s and t. The domain of each of these variables consists of all terms of the considered alphabet. A sequence of terms $(t_1, \ldots, t_n)$ belongs to this constraint if the corresponding substitution $\theta := \{x_1/t_1, ..., x_n/t_n\}$ (or more precisely, the one obtained from $\{x_1/t_1, ..., x_n/t_n\}$ by removing the bindings x_i/t_i such that x_i is t_i) is such that $s\theta = t\theta$. So a solution to a term equation $s = t$ can be identified with a substitution which makes s and t equal.

We now deal with the problem of solving CSP's that consist of such constraints. Each such CSP is uniquely determined by its constraints. So these CSP's can be identified with finite sets of term equations, or simply equations, written as

$$\{s_1 = t_1, ..., s_n = t_n\}.$$

First, we define the result of applying a substitution θ to such a set of equations by

$$\{s_1 = t_1, ..., s_n = t_n\}\theta := \{s_1\theta = t_1\theta, ..., s_n\theta = t_n\theta\}.$$

One of the complications when solving the above CSP's is that the set of solutions is either empty or infinite. Indeed, if θ is a solution to it, then for every substitution η the substitution $\theta\eta$ is also a solution.

However, it turns out that such infinite sets admit a finite representation. This is the essence of the following definition and the theorem that will be discussed in the next subsection.

Definition 4.5 θ is called a *unifier* of $\{s_1 = t_1, ..., s_n = t_n\}$ if

$$s_1\theta = t_1\theta, \ldots, s_n\theta = t_n\theta.$$

If a unifier θ of a singleton set $\{s = t\}$ exists, then we say that s and t are *unifiable* and call θ a *unifier* of s and t.

A unifier θ of a set of equations E is called a *most general unifier* (in short *mgu*) of E if it is more general than all unifiers of E. $\Box$

Intuitively, an mgu is a substitution which makes in each equation two terms equal but which does it in a "most general way", without unnecessary bindings. So θ is an mgu iff every unifier is of the form $\theta\gamma$ for some substitution γ.

Unifiers can thus be identified with the solutions to the above considered CSP's. Additionally, any mgu is a finite representation of the set of all solutions.

Let us consider an example.

Example 4.6
(i) Consider the terms $f(g(x, a), z)$ and $f(y, b)$. Then $\{x/c, y/g(c, a), z/b\}$ is one of their unifiers and so is $\{y/g(x, a), z/b\}$ which is more general than the first one, since $\{x/c, y/g(c, a), z/b\} = \{y/g(x, a), z/b\}\{x/c\}$.

Actually, one can show that $\{y/g(x, a), z/b\}$ is an mgu of $f(g(x, a), z)$ and $f(y, b)$.

(ii) Consider the terms $f(g(x, a), z)$ and $f(g(x, b), b)$. They have no unifier as for no substitution θ we have $a\theta = b\theta$.

(iii) Finally consider the terms $g(x, a)$ and $g(f(x), a)$. They have no unifier either because for any substitution θ the term $x\theta$ is a proper substring of $f(x)\theta$. $\Box$

The problem of deciding whether a CSP that consists of a finite set of term equations has a solution is called the *unification problem.* This problem is solved by providing an algorithm that terminates with failure if the set has no unifier (i.e., no solution) and that otherwise produces a most general unifier of it.

In general, a unifier may not exist for two reasons. The first one is exemplified by Example 4.6(ii) which shows that two constants (or, more generally, two terms starting with a different function symbol) cannot unify. The second one is exemplified by (iii) above which shows that x and $f(x)$ (or more generally, x and a term different from x but in which x occurs) cannot unify. Each possibility can occur at some "inner level" of the considered two terms.

These two possibilities can be found in the algorithm below.

4.1.4 A Unification Algorithm

To solve the unification program we consider the following algorithm that deals with finite sets of term equations.

Martelli–Montanari Algorithm

Nondeterministically choose from the set of equations an equation of a form below and perform the associated action.

(1)	$f(s_1, ..., s_n) = f(t_1, ..., t_n)$	*replace by the equations* $s_1 = t_1, ..., s_n = t_n$,
(2)	$f(s_1, ..., s_n) = g(t_1, ..., t_m)$ where $f \neq g$	*halt with failure,*
(3)	$x = x$	*delete the equation,*
(4)	$t = x$ where t is not a variable	*replace by the equation* $x = t$,
(5)	$x = t$ where $x \notin Var(t)$ and x occurs elsewhere	*perform the substitution* $\{x/t\}$ *on all other equations*
(6)	$x = t$ where $x \in Var(t)$ and $x \neq t$	*halt with failure.*

The algorithm terminates when no action can be performed or when failure arises. Note that action (1) includes the case $c = c$ for every constant c which leads to deletion of such an equation. In addition, action (2) includes the case of two different constants.

To illustrate the operation of this algorithm consider an example.

Example 4.7
(i) Consider the set

$$\{k(z, f(x, b, z)) = k(h(x), f(g(a), y, z))\}.$$

Action (1) applies and yields

$$\{z = h(x), f(x, b, z) = f(g(a), y, z)\}.$$

Choosing the second equation again action (1) applies and yields

$$\{z = h(x), x = g(a), b = y, z = z\}.$$

Choosing the third equation action (4) applies and yields

$$\{z = h(x), x = g(a), y = b, z = z\}.$$

Now, choosing the last equation action (3) applies and yields

$$\{z = h(x), x = g(a), y = b\}.$$

Finally, choosing the second equation action (5) applies and yields

$$\{z = h(g(a)), x = g(a), y = b\}.$$

At this stage no action applies.

(ii) Consider the set

$$\{k(z, f(x, b, z)) = k(h(x), f(g(z), y, z))\}.$$

Let us try to repeat the choices made in (i). By action (1) we get the set

$$\{z = h(x), f(x, b, z) = f(g(z), y, z)\}.$$

Next, choosing the second equation action (1) applies again and yields

$$\{z = h(x), x = g(z), b = y, z = z\}.$$

Choosing the third equation action (4) applies and yields

$$\{z = h(x), x = g(z), y = b, z = z\}.$$

Now, choosing the fourth equation action (3) applies and yields

$$\{z = h(x), x = g(z), y = b\}.$$

Finally, choosing the second equation action (5) applies and yields

$$\{z = h(g(z)), x = g(z), y = b\}.$$

But now choosing the first equation action (6) applies and a failure arises. □

To prove termination of this algorithm we use the following relation $\prec_3$ defined on triples of natural numbers:

$$(a_1, a_2, a_3) \prec_3 (b_1, b_2, b_3)$$

iff

$$\begin{array}{ll} & a_1 < b_1 \\ \text{or} & a_1 = b_1 \text{ and } a_2 < b_2 \\ \text{or} & a_1 = b_1 \text{ and } a_2 = b_2 \text{ and } a_3 < b_3. \end{array}$$

For example, $(1, 15, 1) \prec_3 (1, 16, 1000)$.

This relation, called a *lexicographic ordering* (here on the triples of natural numbers), is *well-founded* which means that no infinite $\prec_3$ descending sequence of triples exists.

The correctness of the MARTELLI–MONTANARI algorithm also relies on the following notions.

Definition 4.8

(i) Two sets of equations are called *equivalent* if they have the same set of unifiers.

(ii) Given a set of equations E we say that an equation from E is in *solved form* if it is of the form $x = t$, where $x \notin Var(t)$ and x does not occur elsewhere in E. If each equation in E is in solved form, we say that E is *in solved form.* □

Note that in case both sets of equations have the same variables the above notion of equivalence boils down to the one introduced in Chapter 1. Also note that a finite set of equations is in solved form if it is of the form $\{x_1 = t_1, ..., x_n = t_n\}$ where the x_is are distinct variables and none of them occurs in a term t_j.

For example, the final set of equations $\{z = h(g(a)), x = g(a), y = b\}$ from Example 4.7(i) is in solved form, because the variables z, x and y occur in it only once.

The interest in sets of equations in solved form is revealed by the following lemma.

Lemma 4.9 (Solved Form) *If $E := \{x_1 = t_1, ..., x_n = t_n\}$ is in solved form, then the substitution $\theta := \{x_1/t_1, ..., x_n/t_n\}$ is an mgu of E.*

Proof. First note that θ is a unifier of E. Indeed, for $i \in [1, n]$ we have $x_i\theta = t_i$ and moreover $t_i\theta = t_i$, since by assumption no x_j occurs in t_i.

Next, suppose η is a unifier of E. Then for $i \in [1, n]$ we have $x_i\eta = t_i\eta = x_i\theta\eta$ because $t_i = x_i\theta$ and for $x \notin \{x_1, \ldots, x_n\}$ we have $x\eta = x\theta\eta$ because $x = x\theta$. Thus $\eta = \theta\eta$, that is θ is more general than η. $\Box$

We call θ the *unifier determined by E*. For example, the substitution

$$\{z/h(g(a)), x/g(a), y/b\}$$

is the unifier determined by

$$\{z = h(g(a)), x = g(a), y = b\},$$

the final set of equations in Example 4.7(i).

So from a set of term equations E in solved form it is straightforward to generate all unifiers of it: they are all of the form $\theta\eta$, where θ is determined by E. This is what we meant at the beginning of this chapter when we introduced complete constraint solvers.

To find a solution to a set of term equations it thus suffices to transform it into an equivalent set which is in solved form. The MARTELLI–MONTANARI algorithm does it if this is possible and otherwise it halts with failure. In the terminology introduced at the beginning of this chapter this algorithm is a complete constraint solver. More precisely, we have the following result.

Theorem 4.10 (Unification) *The* MARTELLI–MONTANARI *algorithm always terminates. If the original set of equations E has a unifier, then the algorithm successfully terminates and produces an equivalent solved set of equations determining an mgu of E and otherwise it terminates with failure.*

Proof. We establish four claims.

Claim 1 *The algorithm always terminates.*

Proof. The proof is rather subtle due to the fact that because of action (5) the individual equations may grow in size.

Given a set of equations E, we call a variable x *solved in E* if for some term t we have $x = t \in E$ and this is the only occurrence of x in E, that is, if the equation $x = t$ is in solved form. We call a variable *unsolved* if it is not solved.

With each set of equations E we now associate the following three functions:

$uns(E)$	–	the number of variables in E that are unsolved,
$lfun(E)$	–	the total number of occurrences of function symbols on the left-hand side of an equation in E,
$card(E)$	–	the number of equations in E.

We claim that each successful action of the algorithm reduces the triple of natural numbers

$$(uns(E), lfun(E), card(E))$$

in the lexicographic ordering $\prec_3$.

Indeed, no action turns a solved variable into an unsolved one, so $uns(E)$ never increases. Further, action (1) decreases $lfun(E)$ by 1, action (3) does not change $lfun(E)$ and decreases $card(E)$ by 1, action (4) decreases $lfun(E)$ by at least 1 and action (5) reduces $uns(E)$ by 1.

The termination is now the consequence of the well-foundedness of $\prec_3$. □

Claim 2 *Each action replaces the set of equations by an equivalent one.*

Proof. The claim holds for action (1) because for all θ we have $f(s_1, ..., s_n)\theta = f(t_1, ..., t_n)\theta$ iff for $i \in [1, n]$ it holds that $s_i\theta = t_i\theta$. For actions (3) and (4) the claim is obvious.

For action (5) consider two sets of equations $E \cup \{x = t\}$ and $E\{x/t\} \cup \{x = t\}$. (Recall that $E\{x/t\}$ denotes the set obtained from E by applying to each of its equations the substitution $\{x/t\}$.)

Then

$$\begin{array}{ll} & \theta \text{ is a unifier of } E \cup \{x = t\} \\ \text{iff} & \theta \text{ is a unifier of } E \text{ and } x\theta = t\theta \\ \text{iff} & \quad \{\text{Binding Lemma 4.3}\} \\ & \{x/t\}\theta \text{ is a unifier of } E \text{ and } x\theta = t\theta \\ \text{iff} & \theta \text{ is a unifier of } E\{x/t\} \text{ and } x\theta = t\theta \\ \text{iff} & \theta \text{ is a unifier of } E\{x/t\} \cup \{x = t\}. \end{array}$$

□

Claim 3 *If the algorithm successfully terminates, then the final set of equations is in solved form.*

Proof. If the algorithm successfully terminates, then the actions (1), (2) and (4) do not apply, so the left-hand side of every final equation is a variable. Moreover, actions (3), (5) and (6) do not apply, so these variables are distinct and none of them occurs on the right-hand side of an equation. □

Claim 4 *If the algorithm terminates with failure, then the set of equations at the moment of failure does not have a unifier.*

Proof. If the failure results by action (2), then the selected equation $f(s_1, ..., s_n) = g(t_1, ..., t_m)$ is an element of the current set of equations and for no θ we have $f(s_1, ..., s_n)\theta = g(t_1, ..., t_m)\theta$.

If the failure results by action (6), then the equation $x = t$ is an element of the current set of equations and for no θ we have $x\theta = t\theta$, because $x\theta$ is a proper substring of $t\theta$. □

These four claims and the Solved Form Lemma 4.9 imply directly the desired conclusion. □

4.1.5 Discussion

It is useful to point out that the MARTELLI-MONTANARI algorithm is inefficient, as for some inputs it can take an exponential time to compute an mgu.

A standard example is the following pair of two terms, where $n > 0$: $f(x_1, \ldots, x_n)$ and $f(g(x_0, x_0), \ldots, g(x_{n-1}, x_{n-1}))$. Define now inductively a sequence of terms $t_1, \ldots, t_n$ as follows:

$$t_1 := g(x_0, x_0),$$

$$t_{i+1} := g(t_i, t_i).$$

It is easy to check that $\{x_1/t_1, \ldots, x_n/t_n\}$ is then a mgu of the terms $f(x_1, \ldots, x_n)$ and $f(g(x_0, x_0), \ldots, g(x_{n-1}, x_{n-1}))$. However, a simple proof by induction shows that each t_i has more than 2^i symbols.

This shows that the total number of symbols in any mgu of the above two terms is exponential in their size. So as long as in MARTELLI–MONTANARI algorithm terms are represented as strings this algorithm runs in exponential time.

Finally, note that the mgu of the above two terms can be computed using n actions of the MARTELLI–MONTANARI algorithm. This shows that the number of actions used in an execution of the MARTELLI–MONTANARI algorithm is not the right measure of the time complexity of this algorithm.

More efficient unification algorithms avoid explicit presentations of the most general unifiers and rely on different internal representation of terms than strings.

4.2 Solving Linear Equations

In this section we discuss linear equations over reals. In the next subsection we clarify the syntax and adapt various notions from the previous section. Then in Subsection 4.2.2 we introduce the VARIABLE ELIMINATION algorithm for solving finite sets of linear equations. Finally, in Subsection 4.2.3 we discuss two improvements of this algorithm: the GAUSS-JORDAN ELIMINATION algorithm and the GAUSSIAN ELIMINATION algorithm.

4.2.1 Linear Expressions and Linear Equations

By a *linear expression* we mean a construct of the form

$$\Sigma_{i=1}^n s_i$$

where each s_i is either a real number or an expression of the form rx where r is a real number and x a variable.

For example, both $3x + 3.5y$ and $3x \perp 2 + 2.5y \perp 2x + 5$ are linear expressions. We can view linear expressions as terms built over an alphabet that for each real r contains a constant r and a unary function symbol $r\cdot$ (standing for the multiplication by r), and the binary function symbol "+" written in an infix form. Consequently can reuse from the previous section the notions of a substitution and an application of a substitution.

By a *linear equation* we mean an equality $s = t$ between the linear equations s and t. Each linear equation can be rewritten to a unique equivalent linear equation which is of the form

$$\Sigma_{i=1}^{n} a_i x_i = b$$

where each a_i is a non-zero real number, b is a real number, each x_i is a variable and where the variables $x_1, \ldots, x_n$ are ordered w.r.t. some predetermined ordering. We say then that the original equation *was normalized*.

For example, $3x + 3 + y \perp x = 2y + 6$ can be rewritten to $2x \perp y = 3$, where we assume that x precedes y.

Let us relate now linear equations to constraints. One natural choice would be to associate with each linear equation a relation on reals. We pursue here a more general possibility according to which each variable domain consists of the set of linear expressions. Given a linear equation e with the variables $x_1, \ldots, x_n$ we associate with it a set of n-tuples of linear expressions defined as follows. An n-tuple $(t_1, \ldots, t_n)$ belongs to this set if the equation $e\{x_1/t_1, \ldots, x_n/t_n\}$ can be normalized to the equation $0 = 0$.

For example, with the equation $x_1 + x_2 = x_3$ we associate a set, the elements of which are for instance $(1, 1, 2), (x, y, x + y)$ and $(z + 1, y, z + y + 2 \perp 1)$.

So a finite set of linear equations can be viewed as a CSP all domains of which equal the set of linear expressions. To find solutions to such CSP's we reuse several notions introduced in the previous section, namely unifiers, mgus, and the concept of sets of equations that are in solved form.

However, because we now consider equality in the sense of normalization to the equation $0 = 0$, we need to modify our terminology. Given a set of linear equations E and a substitution θ we say that θ is a *unifier* of E if each equation in $E\theta$ can be normalized to the equation $0 = 0$.

For example, $\{x_1/z + 1, x_2/y, x_3/z + y + 2 \perp 1\}$ is a unifier of $\{x_1 + x_2 = x_3\}$ because $z + 1 + y = z + y + 2 \perp 1$ can be normalized to $0 = 0$.

Further, we say that two substitutions θ and η are *equivalent* if for all variables x the equation $x\theta = x\eta$ can be normalized to $0 = 0$.

For example, the substitutions $\{x_1/z + 1, x_2/y, x_3/z + y + 2 \perp 1\}$ and $\{x_1/z + 1, x_2/y, x_3/z + y + 1\}$ are equivalent. Also $\{x/y + x \perp y\}$ and the empty substitution are equivalent.

The following observation summarizes the relevant properties of this notion.

Note 4.11 (Equivalence)

(i) Suppose that θ and η are equivalent. Then θ is a unifier of a set of linear equations iff η is.

(ii) *For a variable x and a linear expression t, $x\theta = t\theta$ can be normalized to $0 = 0$ iff θ and $\{x/t\}\theta$ are equivalent.* □

Note that property (ii) is a counterpart of the Binding Lemma 4.3.

We now say that the substitution θ is *more general than* τ if for some substitution η we have that τ is equivalent to $\theta\eta$. We can then reuse the notion of an mgu and also the concepts of equivalence between finite sets of linear equations, of sets of linear equations that are in solved form, and of substitution determined by a set of linear equations in solved form. It is easy to check that the Solved Form Lemma 4.9 remains valid for linear equations and the new notion of mgu.

4.2.2 Variable Elimination Algorithm

To solve finite sets of linear equations we introduce the following algorithm reminiscent of the Martelli–Montanari algorithm, where we assume that all initial linear equations are normalized.

Variable Elimination Algorithm

Nondeterministically choose from the set of equations an equation e and perform the associated action.

(1) e has no variables and is of the form $0 = 0$	*delete e,*
(2) e has no variables and is of the form $r = 0$, where r is a non-zero real	*halt with failure,*
(3) e has a variable, but is not in solved form	*rewrite if needed e as $x = t$, where $x \notin Var(t)$; perform the substitution $\{x/t\}$ on all other equations and normalize them*

The algorithm terminates when no action can be performed or when failure arises.

Note that not only the choice of equations is nondeterministic but also action (3) is nondeterministic in the choice of the variable to be put on the left hand side. The algorithm terminates when no action can be performed or when failure arises.

The following theorem summarizes the properties of this algorithm. It shows that the Variable Elimination algorithm is another example of a complete constraint solver.

Theorem 4.12 (Variable Elimination) *The* Variable Elimination *algorithm always terminates. If the original set of equations E has a solution, then the algorithm successfully terminates and produces an equivalent solved set of equations determining an mgu of E and otherwise it terminates with failure.*

Proof. We proceed as in the proof of the Unification Theorem 4.10 and establish four claims. The proof, however, is much more straightforward.

Claim 1 *The algorithm always terminates.*

Proof. It suffices to notice that actions (1) and (3) reduce the number of equations that are not in solved form. □

Claim 2 *Each action replaces the set of equations by an equivalent one.*

Proof. The claim is obvious for action (1).

For action (3) the proof uses the Equivalence Note 4.11(ii) and is analogous to the proof of Claim 2 in the proof of the Unification Theorem 4.10. The details are straightforward and left to the reader. □

Claim 3 *If the algorithm successfully terminates, then the final set of equations is in solved form.*

Proof. If the algorithm successfully terminates, then the actions (1) and (2) do not apply, so each final equation has a variable. Moreover, action (3) does not apply, so each equation is in solved form. □

Claim 4 *If the algorithm terminates with failure, then the set of equations at the moment of failure has no solution.*

Proof. The failure results by action (2). Then the selected equation has no solution. □

These four claims and the Solved Form Lemma 4.9 reused now for the sets of linear equations imply directly the desired conclusion. □

4.2.3 Gauss-Jordan Elimination and Gaussian Elimination

The Variable Elimination algorithm is inefficient because for each equation with a variable we need to test whether it is in solved form. Now, after performing action (3) an equation becomes in solved form and remains so during the further execution of the algorithm. So by separating the already considered equations from the remaining ones we can improve the efficiency. This observation leads to the following algorithm.

Gauss-Jordan Elimination Algorithm

```
$Sol := \emptyset$;
**while** $E \neq \emptyset$ **do**
    choose $e \in E$;
    $E := E \perp \{e\}$;
    Perform the associated action:
    (1) $e$ has no variables and is of the form $0 = 0$      *skip*,
    (2) $e$ has no variables and is of the form $r = 0$,     *halt with failure*,
        where $r$ is a non-zero real
    (3) $e$ has a variable                                   *rewrite if needed e as*
                                                             $x = t$, *where* $x \notin Var(t)$;
                                                             *perform the substitution* $\{x/t\}$
                                                             *on* $E \cup Sol$
```

```
                                                    and normalize these equations;
                                                    $Sol := Sol \cup \{x = t\}$
od
```

Note that in action (3) we do not now require that e is not in solved form. If it is, the action boils down to adding the equation $x = t$ to the set Sol.

This algorithm enjoys the same properties as the VARIABLE ELIMINATION algorithm, where Sol is the final set of equations. It obviously terminates after the number of loop iterations equal to the number of equations in the original set E. The rest of the correctness proof is the same as in the case of the VARIABLE ELIMINATION algorithm, where at each stage the current set of equations equals $E \cup Sol$.

As a final adjustment we modify the GAUSS-JORDAN ELIMINATION algorithm as follows. We split the variable elimination process in two phases. In the first phase the substitution is performed only on the equations in the set E (so-called *forward substitution).* In the second phase the substitutions are applied on the remaining equations proceeding "backwards" (so-called *backward substitution).*

To carry out the second phase we need to process the equations in a specific order. To this end we employ a stack with the operation **push**(e, S) which for an equation e and a stack S pushes e onto S, denote the empty stack by **empty**, and the top and the tail of a non-empty stack S respectively by **top**(S) and **tail**(S).

The algorithm has the following form.

GAUSSIAN ELIMINATION ALGORITHM

```
$Sol :=$ empty;
while $E \neq \emptyset$ do
  choose $e \in E$;
  $E := E \perp \{e\}$;
  Perform the associated action:
  (1) $e$ has no variables and is of the form $0 = 0$      skip,
  (2) $e$ has no variables and is of the form $r = 0$,     halt with failure,
      where $r$ is a non-zero real
  (3) $e$ has a variable                                  rewrite if needed $e$ as
                                                          $x = t$, where $x \notin Var(t)$;
                                                          perform the substitution $\{x/t\}$
                                                          on $E$
                                                          and normalize these equations;
                                                          push$(x = t, Sol)$
od;
while $Sol \neq$ empty do
    $e :=$ top$(Sol)$; suppose $e$ is $x = t$;
    $Sol :=$ tail$(Sol)$;
    $E := E \cup \{e\}$;
    perform the substitution $\{x/t\}$ on $Sol$ and normalize these equations
od
```

The algorithm terminates with E as the final set of equations. We leave the proof of its correctness as an exercise. To illustrate use of this algorithm consider the following example.

Example 4.13 Take the following system of linear equations:

$$\begin{array}{rcrcrcrcr}
5\,x_1 & + & 4\,x_2 & + & 3\,x_3 & + & 9\,x_4 & = & 49 \\
3\,x_1 & + & 2\,x_2 & + & x_3 & + & 2\,x_4 & = & 19 \\
-14\,x_1 & - & 8\,x_2 & - & 7\,x_3 & + & 5\,x_4 & = & -68 \\
12\,x_1 & + & 6\,x_2 & - & 25\,x_3 & + & 10\,x_4 & = & -38
\end{array}$$

Selecting the first equation we can rewrite it into

$$x_1 = \perp 0.8x_2 \perp 0.6x_3 \perp 1.8x_4 + 9.8. \tag{4.1}$$

By virtue of action (3) we now perform the appropriate substitution in the other three equations. After normalization this yields

$$\begin{array}{rcrcrcr}
-0.4\,x_2 & - & 0.8\,x_3 & - & 3.4\,x_4 & = & -\,10.4 \\
3.2\,x_2 & + & 1.4\,x_3 & + & 30.2\,x_4 & = & 69.2 \\
3.6\,x_2 & - & 32.2\,x_3 & - & 11.6\,x_4 & = & -\,155.6
\end{array}$$

Selecting in the above system the first equation we can rewrite it into

$$x_2 = \perp 2x_3 \perp 8.5x_4 + 26. \tag{4.2}$$

As before we perform the appropriate substitution in the other two equations. After normalization this yields

$$\begin{array}{rcrcr}
-5\,x_3 & + & 3\,x_4 & = & -14 \\
-25\,x_3 & + & 19\,x_4 & = & -\,62
\end{array}$$

We now rewrite the first equation above into

$$x_3 = 0.6x_4 + 2.8 \tag{4.3}$$

and perform the appropriate substitution in the other equation. This yields after normalization

$$x_4 = 2. \tag{4.4}$$

In the execution of the GAUSSIAN ELIMINATION algorithm the first **while** loop now terminated with $E = \emptyset$ and the second **while** loop is entered with the stack *Sol* formed by four equations: (4.1), (4.2), (4.3) and (4.4), pushed onto it in this order, so with the last one as the top.

Taking (4.4) and performing the corresponding substitution in (4.1), (4.2) and (4.3) we get after the normalization

$$\begin{array}{rcrcrcr} x_1 & = & -\,0.8\,x_2 & - & 0.6\,x_3 & + & 6.2 \\ x_2 & = & & - & 2\,x_3 & + & 9 \\ x_3 & = & & & & & 4 \end{array}$$

Next, we take the last equation and perform the corresponding substitution in the other two equations. After normalization this yields

$$\begin{array}{rcrcr} x_1 & = & -\,0.8\,x_2 & + & 3.8 \\ x_2 & = & & & 1 \end{array}$$

As a final step we substitute in the first equation above x_2 by 1. In this way we obtain after normalization

$$x_1 = 3.$$

The algorithm now terminates and we end up with the following set of equations E:

$$\{x_1 = 3,\ x_2 = 1,\ x_3 = 4,\ x_4 = 2\}.$$

□

4.3 Bibliographic Remarks

The unification problem was introduced and solved by Robinson (1965) who recognized its importance for automated theorem proving. The unification problem also appeared implicitly in the PhD thesis of Herbrand in 1930 (see Herbrand (1971, page 148)) in the context of solving term equations, but in an informal way and without proofs. The MARTELLI–MONTANARI algorithm presented in Subsection 4.1.4 is from Martelli & Montanari (1982). It is similar to Herbrand's original algorithm. The presentation in Section 4.1 is adapted from Apt (1997).

Efficient unification algorithms are presented in Paterson & Wegman (1978) and Martelli & Montanari (1982). A thorough analysis of the time complexity of various unification algorithms is carried out in Albert, Casas & Fages (1993). For a recent survey on unification see Baader & Siekmann (1994). Robinson (1992) provides an interesting account of the history of the unification algorithms.

The analysis of complexity of the GAUSSIAN ELIMINATION algorithm and its elegant reformulation in terms of matrix operations can be found in a number of standard textbooks, e.g., Chvátal (1983), from which we took Example 4.13.

References

Albert, L., Casas, R. & Fages, F. (1993), 'Average case analysis of unification algorithms', *Theoretical Computer Science* **113**(1, 24), 3–34.

Apt, K. R. (1997), *From Logic Programming to Prolog*, Prentice-Hall, London, U.K.

Baader, F. & Siekmann, J. (1994), Unification Theory, *in* D. Gabbay, C. Hogger & J. Robinson, eds, 'Handbook of Logic in Artificial Intelligence and Logic Programming Vol. 2, Deduction Methodologies', Oxford University Press, pp. 41–125.

Chvátal, V. (1983), *Linear Programming*, W.H. Freeman and Company, New York.

Herbrand, J. (1971), *Logical Writings*, Reidel. W.D. Goldfarb, ed.

Martelli, A. & Montanari, U. (1982), 'An efficient unification algorithm', *ACM Transactions on Programming Languages and Systems* **4**, 258–282.

Paterson, M. & Wegman, M. (1978), 'Linear unification', *J. Comput. System Sci.* **16**(2), 158–167.

Robinson, J. (1965), 'A machine-oriented logic based on the resolution principle', *J. ACM* **12**(1), 23–41.

Robinson, J. (1992), 'Logic and logic programming', *Communications of ACM* **35**(3), 40–65.

Chapter 5

Local Consistency Notions

Ideally, we would like to solve CSP's directly, by means of some efficient algorithm. But the definition of a CSP is extremely general, so, as already mentioned in Chapter 1, no general efficient methods for solving CSP's exist.

Various general techniques were developed to solve CSP's and in the absence of efficient algorithms a combination of these techniques is a natural way to proceed.

In Chapter 3 we explained that the main idea is to reduce a given CSP to another one that is equivalent but easier to solve. This process is called constraint propagation and the algorithms that achieve such a reduction usually aim at reaching some "local consistency". Informally, local consistency means that certains subparts of the considered CSP are consistent, that is have a solution.

In this chapter we review various most common forms of local consistency. In a later chapter we discuss the algorithms that achieve them. To achieve a smooth transition between these two chapters, each time we introduce a notion of local consistency we also provide its characterization. These characterizations will be used later to generate the appropriate algorithms.

5.1 A Proof Theoretical Framework

5.1.1 Proof Rules and Derivations

We start by introducing a very simple framework based on the proof rules. We shall use it to characterize the discussed notions of local consistency. Also, this framework will allow us to define specific constraint solvers in a particularly simple way.

In this framework we introduce proof rules and derivations. The proof rules are used to express transformations of CSP's. So they are of the form

$$\frac{\phi}{\psi}$$

where ϕ and ψ are CSP's. We assume here that ϕ is not failed and its set of constraints is non-empty.

In general, achieving local consistency either reduces the domains of the considered variables or reduces the considered constraints. This translates into two types of rules.

Assume that

$$\phi := \langle \mathcal{C} \ ; \ \mathcal{DE} \rangle$$

and

$$\psi := \langle \mathcal{C}' \ ; \ \mathcal{DE}' \rangle.$$

We distinguish:

- *Domain reduction rules*, or in short *reduction rules.* These are rules in which the new domains are respective subsets of the old domains and the new constraints are respective restrictions of the old constraints to the new domains.

  So here

  - $\mathcal{DE} := x_1 \in D_1, \ldots, x_n \in D_n$,
  - $\mathcal{DE}' := x_1 \in D'_1, \ldots, x_n \in D'_n$,
  - for $i \in [1..n]$ we have $D'_i \subseteq D_i$,
  - $\mathcal{C}'$ is the result of restricting each constraint in $\mathcal{C}$ to the corresponding subsequence of the domains $D'_1, \ldots, D'_n$.

  Here a failure is reached only when a domain of one or more variables gets reduced to the empty set.

  When all constraints in $\mathcal{C}'$ are solved, we call such a rule a *solving rule.*

- *Transformation rules.* These rules are not domain reduction rules and are such that $\mathcal{C}' \neq \emptyset$ and $\mathcal{DE}'$ extends $\mathcal{DE}$.

  The fact that $\mathcal{DE}'$ extends $\mathcal{DE}$ means that the domains of common variables are identical and that possibly new domain expressions have been added to $\mathcal{DE}$. Such new domain expressions deal with new variables on which some constraints have been introduced.

  Here a failure is reached only when the false constraint $\perp$ is generated.

  A typical case of transformation rules are

  Introduction rules. These rules are of the form

  $$\frac{\langle \mathcal{C} \ ; \ \mathcal{DE} \rangle}{\langle \mathcal{C}, C \ ; \ \mathcal{DE} \rangle}$$

  in which a new constraint, C, was introduced in the conclusion.

  If such a rule does not depend on $\mathcal{DE}$, then we abbreviate it to

  $$\frac{\mathcal{C}}{\mathcal{C}, C}$$

  and similarly with other transformation rules.

Our intention is to use the proof rules to reduce one CSP to another CSP in such a way that the equivalence (usually w.r.t. to the initial set of variables) is maintained. This motivates the following definition.

Definition 5.1 A proof rule

$$\frac{\phi}{\psi}$$

is called *equivalence preserving* (respectively *equivalence preserving* w.r.t. a set of variables X) if ϕ and ψ are equivalent (respectively equivalence preserving w.r.t. X). $\Box$

From the way we introduce the proof rules in the sequel it will be clear that all of them are equivalence preserving. If the sets of variables of the premise CSP and conclusion CSP differ, then the rules will be equivalence preserving w.r.t. the set of common variables.

Now that we have defined the proof rules, we define the result of applying a proof rule to a CSP. Intuitively, we just replace in a given CSP the part that coincides with the premise by the conclusion and restrict the "old" constraints to the new domains.

Because of variable clashes we need to be more precise. So assume a CSP of the form $\langle \mathcal{C} \cup \mathcal{C}_1 \ ; \ \mathcal{DE} \cup \mathcal{DE}_1 \rangle$ and consider a rule of the form

$$\frac{\langle \mathcal{C}_1 \ ; \ \mathcal{DE}_1 \rangle}{\langle \mathcal{C}_2 \ ; \ \mathcal{DE}_2 \rangle} \tag{5.1}$$

Call a variable that appears in the conclusion but not in the premise an *introduced* variable of the rule. By appropriate renaming we can assume that no introduced variable of this rule appears in $\langle \mathcal{C} \ ; \ \mathcal{DE} \rangle$.

Let now $\mathcal{C}'$ be the result of restricting each constraint in $\mathcal{C}$ to the domains in $\mathcal{DE} \cup \mathcal{DE}_2$. We say that rule (5.1) *can be applied* to $\langle \mathcal{C} \cup \mathcal{C}_1 \ ; \ \mathcal{DE} \cup \mathcal{DE}_1 \rangle$ and call

$$\langle \mathcal{C}' \cup \mathcal{C}_2 \ ; \ \mathcal{DE} \cup \mathcal{DE}_2 \rangle$$

the *result of applying rule* (5.1) *to* $\langle \mathcal{C} \cup \mathcal{C}_1 \ ; \ \mathcal{DE} \cup \mathcal{DE}_1 \rangle$.

The following lemma explains why the equivalence preserving rules are important.

Lemma 5.2 (Equivalence) *Suppose that the CSP ψ is the result of applying an equivalence preserving rule to the CSP ϕ. Then ϕ and ψ are equivalent.* $\Box$

To discuss the effect of an application of a proof rule to a CSP we introduce the following notions.

Definition 5.3 Consider two CSP's ϕ and ψ and a rule R.

- We call ϕ a *reformulation* of ψ if the removal of solved constraints from ϕ and ψ yields the same CSP.
- Suppose that ψ is the result of applying the rule R to the CSP ϕ. If ψ is not a reformulation of ϕ, then we call this a *relevant application* of R to ϕ.

- Suppose that the rule R cannot be applied to ϕ or no application of it to ϕ is relevant. Then we say that ϕ is *closed under the applications of R*. □

The last notion is crucial for our considerations. To understand it better consider the following examples.

Example 5.4

(i) Consider the following rule already mentioned in Subsection 3.1.5:

$$\frac{\langle x < y \ ; \ x \in [l_x..h_x], y \in [l_y..h_y]\rangle}{\langle x < y \ ; \ x \in [l_x..min(h_x, h_y \perp 1)], y \in [max(l_y, h_x + 1), h_x]\rangle}$$

It is easy to check that the considered there CSP

$$\langle x < y \ ; \ x \in [50..99], y \in [51..100]\rangle$$

is closed under the applications of this rule.

(ii) Assume for a moment the expected interpretation of propositional formulas. As a more subtle example consider now the CSP $\phi := \langle x \wedge y = z \ ; \ x = 1, y = 0, z = 0\rangle$. Here $x = 1$ is an abbreviation for the domain expression $x \in \{1\}$ and similarly for the other variables.

This CSP is closed under the applications of the transformation rule

$$\frac{\langle x \wedge y = z \ ; \ x = 1, y \in D_y, z \in D_z\rangle}{\langle y = z \ ; \ x = 1, y \in D_y, z \in D_z\rangle}$$

Indeed, this rule can be applied to ϕ; the outcome is $\psi := \langle y = z \ ; \ x = 1, y = 0, z = 0\rangle$. After the removal of solved constraints from ϕ and ψ we get in both cases the solved CSP $\langle \emptyset \ ; \ x = 1, y = 0, z = 0\rangle$.

(iii) In contrast, the CSP $\phi := \langle x \wedge y = z \ ; \ x = 1, y \in \{0, 1\}, z \in \{0, 1\}\rangle$ is not closed under the applications of the above rule because $\langle y = z \ ; \ x = 1, y \in \{0, 1\}, z \in \{0, 1\}\rangle$ is not a reformulation of ϕ. □

By iterating rule applications we obtain derivations. However, we would like to consider only those applications of the proof rules that cause some change. This leads us to the following definition.

Definition 5.5 Assume a set of proof rules. By a *derivation* we mean a maximal sequence of CSP's such that each of them is obtained from the previous one by a relevant application of a proof rule.

A derivation is called *successful* if it is finite and its last element is a solved CSP. A derivation is called *failed* if it is finite and its last element is a failed CSP. □

Because of the requirement that each rule application is relevant the derivations in general will be finite. Note that some finite derivations are neither successful nor failed. In fact, many constraint solvers yield CSP's that are neither solved nor failed — their aim is to bring the initial CSP to some specific, simpler form that usually satisfies some specific local consistency notion.

In this chapter we shall not use the derivations explicitly but it is useful to have them in mind when considering various proof rules. Namely, suppose that we are given a set of rules $\mathcal{R}$ such that all derivations obtained using them are finite. Then to obtain a CSP that is closed under the applications of the rules from $\mathcal{R}$ it suffices to keep applying them in such a way that each application is relevant. The final CSP in any derivation produced in this way is then closed under the applications of the rules from $\mathcal{R}$.

To avoid redundant rule applications, in any implementation the rules introduced here should be selected and scheduled in an appropriate way. This matter will be discussed in a later chapter.

5.1.2 Examples of Proof Rules

Before we proceed it is helpful to get familiarized with this proof theoretic framework.

When presenting specific proof rules we delete from the conclusion all solved constraints. Also, we abbreviate the domain expression $x \in \{a\}$ to $x = a$.

As an example of a domain reduction rule consider the following rule:

EQUALITY 1

$$\frac{\langle x = y \; ; \; x \in D_1, y \in D_2 \rangle}{\langle x = y \; ; \; x \in D_1 \cap D_2, y \in D_1 \cap D_2 \rangle}$$

Note that this rule yields a failure when $D_1 \cap D_2 = \emptyset$. In case $D_1 \cap D_2$ is a singleton this rule becomes a solving rule, that is the constraint $x = y$ becomes solved (and hence deleted). Note also the following solving rule:

EQUALITY 2

$$\frac{\langle x = x \; ; \; x \in D \rangle}{\langle \; ; \; x \in D \rangle}$$

Following the just introduced convention we dropped the constraint from the conclusion of the *EQUALITY 2* rule. This explains its format.

As further examples of solving rules consider the following three concerning disequality:

DISEQUALITY 1

$$\frac{\langle x \neq x \; ; \; x \in D \rangle}{\langle \; ; \; x \in \emptyset \rangle}$$

DISEQUALITY 2

$$\frac{\langle x \neq y \; ; \; x \in D_1, y \in D_2 \rangle}{\langle \; ; \; x \in D_1, y \in D_2 \rangle}$$

where $D_1 \cap D_2 = \emptyset$,

DISEQUALITY 3

$$\frac{\langle x \neq y \ ; \ x \in D, y = a \rangle}{\langle \ ; \ x \in D \perp \{a\}, y = a \rangle}$$

where $a \in D$, and similarly with $x \neq y$ replaced by $y \neq x$.

So the *DISEQUALITY 1* rule yields a failure while the *DISEQUALITY 3* rule can yield a failure.

Next, as an example of a transformation rule consider the following rule that substitutes a variable by a value:

SUBSTITUTION

$$\frac{\langle \mathcal{C} \ ; \ \mathcal{DE}, x = a \rangle}{\langle \mathcal{C}\{x/\overline{a}\} \ ; \ \mathcal{DE}, x = a \rangle}$$

where x occurs in $\mathcal{C}$.

We assume here that the constraints in $\mathcal{C}$ are written in some further unspecified language. Here $\overline{a}$ stands for the constant that denotes in this language the value a and $\mathcal{C}\{x/\overline{a}\}$ denotes the set of constraints obtained from $\mathcal{C}$ by substituting in each of them every occurrence of x by $\overline{a}$. So x does not occur in $\mathcal{C}\{x/\overline{a}\}$.

Another example of a transformation rule is the following one

DELETION

$$\frac{\langle \mathcal{C}, \top \ ; \ \mathcal{DE} \rangle}{\langle \mathcal{C} \ ; \ \mathcal{DE} \rangle}$$

in which we delete the true constraint $\top$.

As a final example recall the following introduction rule already mentioned in Subsection 3.1.5 of Chapter 3.

TRANSITIVITY of $<$

$$\frac{\langle x < y, y < z \ ; \ \mathcal{DE} \rangle}{\langle x < y, y < z, x < z \ ; \ \mathcal{DE} \rangle}$$

where we assume that all variables range over reals.

This completes the presentation of our proof theoretic framework.

5.2 Arc Consistency

We begin now our review of the most common notions of local consistency. The first notion we introduce is arc consistency. Informally, a binary constraint is arc consistent if every value in each domain participates in a solution. And a CSP is arc consistent if all its binary constraints are. Here is the precise definition.

Definition 5.6

- Consider a binary constraint C on the variables x, y with the domains D_x and D_y, that is $C \subseteq D_x \times D_y$. We call C *arc consistent* if
  - $\forall a \in D_x \exists b \in D_y \ (a, b) \in C$,
  - $\forall b \in D_y \exists a \in D_x \ (a, b) \in C$.
- We call a CSP *arc consistent* if all its binary constraints are arc consistent. □

So a binary constraint is arc consistent if every value in each domain has a "witness" in the other domain, where we call b a witness for a if the pair (a, b) (or, depending on the ordering of the variables, (b, a)) belongs to the constraint.

Among the notions of local consistency arc consistency is the most important one. The reason is that CSP's with binary constraints are very common and that, as we shall see, achieving arc consistency can be done efficiently.

The following three examples illustrate this notion.

Example 5.7
(i) Consider the CSP which consists of only one constraint, $x < y$ interpreted over the domains $D_x = [5..10]$ of x and $D_y = [3..7]$ of y. This CSP is not arc consistent. Indeed, take for instance the value 8 in the domain $[5..10]$ of x. Then there is no b in $[3..7]$ such that $8 < b$.

(ii) Next, consider the CSP of Example 2.2 that formalizes the n *Queens* problem where $n \geq 4$. It is easy to see that this CSP is arc consistent. Informally it means that no matter where we place one queen on the chess board $n \times n$ we can place a queen in any other row so that they do not attack each other.

More formally, we need to analyze each constraint separately. Consider for instance the constraint $x_i \perp x_j \neq i \perp j$ with $1 \leq i < j \leq n$ and take $a \in [1..n]$. Then there exists $b \in [1..n]$ such that $a \perp b \neq i \perp j$: just take $b \in [1..n]$ that is different from $a \perp i + j$.

(iii) Finally, consider the CSP of Example 2.8 that deals with the crossword puzzle. This CSP is not arc consistent. Indeed, take for instance the already discussed constraint

$$\begin{aligned} C_{1,2} = \{&(\text{HOSES, SAILS}), (\text{HOSES, SHEET}), (\text{HOSES, STEER}), \\ &(\text{LASER, SAILS}), (\text{LASER, SHEET}), (\text{LASER, STEER})\} \ . \end{aligned}$$

on the variables associated with the positions 1 and 2. Both positions need to be filled by five letter words, so both domains equal {HOSES, LASER, SAILS, SHEET, STEER}.

Now, no word in this set begins with the letter I, so for the value SAILS for the first variable no value for the second variable exists such that the resulting pair satisfies the considered constraint. □

Next, we characterize the notion of arc consistency in terms of proof rules. To this end we introduce the following two rules, where C is a constraint on the variables x and y:

ARC CONSISTENCY 1

$$\frac{\langle C\ ;\ x \in D_x, y \in D_y\rangle}{\langle C\ ;\ x \in D'_x, y \in D_y\rangle}$$

where $D'_x := \{a \in D_x \mid \exists\, b \in D_y\ (a,b) \in C\}$

ARC CONSISTENCY 2

$$\frac{\langle C\ ;\ x \in D_x, y \in D_y\rangle}{\langle C\ ;\ x \in D_x, y \in D'_y\rangle}$$

where $D'_y := \{b \in D_y \mid \exists\, a \in D_x\ (a,b) \in C\}$.

In other words, the *ARC CONSISTENCY 1* and *2* rules deal with projections, respectively on the first or the second domain. Intuitively, they remove from the domain of x, respectively the domain of y, the values that do not participate in a solution to the constraint C. These rules can be visualized by means of Figure 5.1.

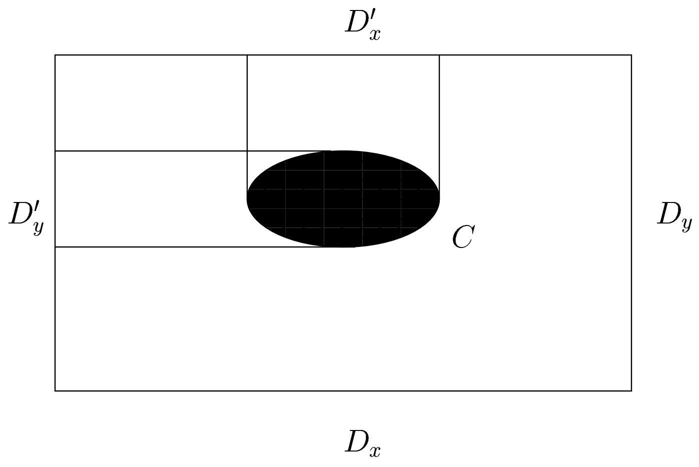

Figure 5.1: Projection functions

The following simple result provides a characterization of arc consistency in terms of the above two rules.

Note 5.8 (Arc Consistency) *A CSP is arc consistent iff it is closed under the applications of the ARC CONSISTENCY rules 1 and 2.*

Proof. It suffices to note that a constraint C on the variables x and y with the respective domains D_x and D_y is arc consistent iff $D'_x = D_x$ and $D'_y = D_y$, where D'_x and D'_y are defined as in the *ARC CONSISTENCY* rules *1* and *2*. □

5.3 Hyper-arc Consistency

The notion of arc consistency generalizes in a natural way to arbitrary constraints.

Definition 5.9

- Consider a constraint C on the variables $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$, that is $C \subseteq D_1 \times \cdots \times D_n$. We call C *hyper-arc consistent* if for every $i \in [1..n]$ and $a \in D_i$ there exists $d \in C$ such that $a = d[i]$.
- We call a CSP *hyper-arc consistent* if all its constraints are hyper-arc consistent. □

Intuitively, a constraint C is hyper-arc consistent if for every involved domain each element of it participates in a solution to C.

The following two examples of Boolean constraints illustrate this notion.

Example 5.10
(i) Consider the following CSP already mentioned in Subsection 5.1.1:

$$\langle x \land y = z \ ; \ x = 1, y \in \{0,1\}, z \in \{0,1\}\rangle.$$

It is easily seen to be hyper-arc consistent. Indeed, each element of every domain participates in a solution. For example, the value 0 of the domain of z participates in the solution $(1, 0, 0)$.

(ii) In contrast, the CSP

$$\langle x \land y = z \ ; \ x \in \{0,1\}, y \in \{0,1\}, z = 1\rangle$$

is not hyper-arc consistent. Indeed, the value 0 of the domain of x does not participate in any solution. □

We can easily modify the characterization of arc consistency to the case of hyper-arc consistency. To this end we introduce the following proof rule:

HYPER-ARC CONSISTENCY

$$\frac{\langle C \ ; \ x_1 \in D_1, \ldots, x_n \in D_n\rangle}{\langle C \ ; \ x_1 \in D_1, \ldots, x_{i-1} \in D_{i-1}, x_i \in D'_i, x_{i+1} \in D_{i+1}, \ldots, x_n \in D_n\rangle}$$

where C is a constraint on the variables $x_1, \ldots, x_n$, $i \in [1..n]$, and

$$D'_i := \{a \in D_i \mid \exists d \in C \ a = d[i]\}.$$

The following analogue of the Arc Consistency Note 5.8 holds.

Note 5.11 (Hyper-arc Consistency) *A CSP is hyper-arc consistent iff it is closed under the applications of the HYPER-ARC CONSISTENCY rule.*

Proof. Again, it suffices to note that a constraint C on the variables $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$ is hyper-arc consistent iff for every $i \in [1..n]$ we have $D'_i = D_i$, where D'_i is defined as in the *HYPER-ARC CONSISTENCY* rule. □

5.4 Directional Arc Consistency

The notion of arc consistency can be modified to take into account some linear ordering $\prec$ on the considered variables. The idea is that we require the existence of witnesses only "in one direction". This yields directional arc consistency defined as follows.

Definition 5.12 Assume a linear ordering $\prec$ on the considered variables.

- Consider a binary constraint C on the variables x, y with the domains D_x and D_y. We call C *arc consistent w.r.t.* $\prec$ if
  - $\forall a \in D_x \exists b \in D_y \ (a, b) \in C$ provided $x \prec y$,
  - $\forall b \in D_y \exists a \in D_x \ (a, b) \in C$ provided $y \prec x$.
- We call a CSP *directionally arc consistent w.r.t.* $\prec$ if all its binary constraints are directionally arc consistent w.r.t. $\prec$. □

To see the difference between the notions of arc consistency and directional arc consistency consider the following examples.

Example 5.13
(i) The CSP

$$\langle x < y \ ; \ x \in [5..10], y \in [3..7] \rangle$$

considered in Example 5.7(i) is not directionally arc consistent w.r.t. any linear ordering $\prec$. Indeed, if $x \prec y$ then, as already noted, for $x = 8$ no $y \in [3..7]$ exists such that $8 < y$. And if $y \prec x$ then for $y = 4$ no $x \in [5..10]$ exists such that $x < 4$.

(ii) In contrast, the CSP

$$\langle x < y \ ; \ x \in [2..10], y \in [3..7] \rangle$$

is not arc consistent but it is directionally arc consistent w.r.t. the ordering $\prec$ such that $y \prec x$. Namely, for each $b \in [3..7]$ a value $a \in [2..10]$ exists such that $a < b$, namely $a = 2$.

(iii) Finally, reconsider the CSP of Example 2.8 that deals with the crossword puzzle. In Example 5.7(iii) we noted that it is not arc consistent. In fact, this CSP is not directionally arc consistent w.r.t. any linear ordering $\prec$ on its variables.

In particular, the constraint $C_{1,2}$ is not directionally arc consistent no matter what ordering we choose on its variables. Indeed, as already noted, no word in the set {HOSES, LASER, SAILS, SHEET, STEER} begins with the letter I, so the value SAILS for the first variable has no "witness" in the second domain. Moreover, no word in this set has L as the third letter, so the value LASER for the second variable has no "witness" in the first domain. □

As in the case of arc and hyper-arc consistency a simple characterization of directional arc consistency can be given in terms of rules. To this end we can reuse the *ARC CONSISTENCY* rules *1* and *2*. We say that the *ARC CONSISTENCY* rule *1* (respectively, rule *2*) *respects the variable ordering* $\prec$ if $x \prec y$ (respectively $y \prec x$). So for any two variables x and y exactly one of the *ARC CONSISTENCY* rules *1* and *2* respects the variable ordering $\prec$.

We have then the following result the proof of which we leave as Exercise 2.

Note 5.14 (Directional Arc Consistency) *A CSP is directionally arc consistent w.r.t.* $\prec$ *iff it is closed under the applications of those ARC CONSISTENCY rules 1 and 2 that respect the variable ordering* $\prec$. □

5.5 Path Consistency

The notions of local consistency introduced so far dealt with each constraint separately. This does not need to be the case in general. In this section we discuss a notion of local consistency which involves two constraints considered together. Here and elsewhere it will be useful to limit oneself to a specific type of CSP's.

Definition 5.15 We call a CSP $\mathcal{P}$ *normalized* if for each subsequence $\bar{x}$ of its variables there exists at most one constraint on $\bar{x}$ in $\mathcal{P}$.

Given a normalized CSP and a subsequence $\bar{x}$ of its variables we denote by $C_{\bar{x}}$ the unique constraint on the variables $\bar{x}$ if it exists and otherwise the "universal" relation on $\bar{x}$ that equals the Cartesian product of the domains of the variables in $\bar{x}$. □

Every CSP is trivially equivalent to a normalized CSP. Indeed, for each subsequence $\bar{x}$ of the variables of $\mathcal{P}$ such that a constraint on $\bar{x}$ exists, we just need to replace the set of all constraints on $\bar{x}$ by its intersection. Note that the universal relations $C_{\bar{x}}$ are not constraints of the normalized CSP.

Given a subsequence x, y of the variables of a normalized CSP we introduce a "supplementary" relation $C_{y,x}$ defined by

$$C_{y,x} := \{(b, a) \mid (a, b) \in C_{x,y}\}.$$

The supplementary relations are not parts of the considered CSP as none of them is defined on a subsequence of its variables, but they allow us a more compact presentation. We now introduce the following notion.

Definition 5.16 We call a normalized CSP *path consistent* if for each subset $\{x, y, z\}$ of its variables we have

$$C_{x,y} = \{(a, b) \mid \exists c\, ((a, c) \in C_{x,z}, (b, c) \in C_{y,z})\}.$$

□

In other words, a normalized CSP is path consistent if for each subset $\{x, y, z\}$ of its variables and the pair $(a, b) \in C_{x,y}$ there exists c such that $(a, c) \in C_{x,z}$ and $(b, c) \in C_{y,z}$.

In the above definition we use the constraints of the form $C_{u,v}$ for any *subset* $\{u, v\}$ of the considered sequence of variables. If u, v is not a *subsequence* of the original sequence of variables, then $C_{u,v}$ is a supplementary relation that is not a constraint of the original

CSP. At the expense of some redundancy we can rewrite the above definition so that only the constraints of the considered CSP and the universal relations are involved. This is the contents of the following simple observation that hopefully clarifies the above definition.

Note 5.17 (Alternative Path Consistency) *A normalized CSP is path consistent iff for each subsequence x, y, z of its variables we have*

$$C_{x,y} = \{(a,b) \mid \exists c\, ((a,c) \in C_{x,z}, (b,c) \in C_{y,z})\},$$

$$C_{x,z} = \{(a,c) \mid \exists b\, ((a,b) \in C_{x,y}, (b,c) \in C_{y,z})\},$$

$$C_{y,z} = \{(b,c) \mid \exists a\, ((a,b) \in C_{x,y}, (a,c) \in C_{x,z})\}.$$

□

Recall, that the relations $C_{x,y}, C_{x,z}$ and $C_{y,z}$ denote either constraints of the considered normalized CSP or the universal binary relations on the domains of the corresponding variables.

To further clarify this notion consider the following two examples.

Example 5.18
(i) Consider the following normalized CSP

$$\langle x < y, y < z, x < z \ ; \ x \in [0..4], y \in [1..5], z \in [6..10]\rangle.$$

This CSP is path consistent. Indeed, we have

$$C_{x,y} = \{(a,b) \mid a < b, a \in [0..4], b \in [1..5]\},$$

$$C_{x,z} = \{(a,c) \mid a < c, a \in [0..4], c \in [6..10]\},$$

$$C_{y,z} = \{(b,c) \mid b < c, b \in [1..5], c \in [6..10]\}.$$

It is straightforward to check that these three constraints satisfy the conditions of the Alternative Path Consistency Note 5.17. For example, for every pair $(a,c) \in C_{x,z}$ there exists $b \in [1..5]$ such that $a < b$ and $b < c$. Namely, we can always take $b = 5$.

(ii) Take now the following normalized CSP

$$\langle x < y, y < z, x < z \ ; \ x \in [0..4], y \in [1..5], z \in [5..10]\rangle$$

that differs from the previous one only in the domain for z. Then this CSP is not path consistent. Indeed, in this CSP we have

$$C_{x,z} = \{(a,c) \mid a < c, a \in [0..4], c \in [5..10]\}$$

and for the pair of values $4 \in [0..4]$ and $5 \in [5..10]$ no value $b \in [1..5]$ exists such that $4 < b$ and $b < 5$. □

To characterize the notion of path consistency we introduce three transformation rules that correspond to the Alternative Path Consistency Note 5.17. In these rules, following the convention introduced in Subsection 5.1.1, we omit the domain expressions. We assume here that x, y, z is a subsequence of the variables of the considered CSP.

PATH CONSISTENCY 1

$$\frac{C_{x,y}, C_{y,z}, C_{x,z}}{C'_{x,y}, C_{y,z}, C_{x,z}}$$

where the constraint $C'_{x,y}$ on the variables x, y is defined by

$$C'_{x,y} := C_{x,y} \cap \{(a,b) \mid \exists c\, ((a,c) \in C_{x,z}, (b,c) \in C_{y,z})\},$$

PATH CONSISTENCY 2

$$\frac{C_{x,y}, C_{y,z}, C_{x,z}}{C_{x,y}, C_{y,z}, C'_{x,z}}$$

where the constraint $C'_{x,z}$ on the variables x, z is defined by

$$C'_{x,z} := C_{x,z} \cap \{(a,c) \mid \exists b\, ((a,b) \in C_{x,y}, (b,c) \in C_{y,z})\},$$

PATH CONSISTENCY 3

$$\frac{C_{x,y}, C_{y,z}, C_{x,z}}{C_{x,y}, C'_{y,z}, C_{x,z}}$$

where the constraint $C'_{y,z}$ on the variables y, z is defined by

$$C'_{y,z} := C_{y,z} \cap \{(b,c) \mid \exists a\, ((a,b) \in C_{x,y}, (a,c) \in C_{x,z})\}.$$

The following observation provides the corresponding characterization result.

Note 5.19 (Path Consistency) *A normalized CSP is path consistent iff it is closed under the applications of the PATH CONSISTENCY rules 1, 2 and 3.*

Proof. Using the Alternative Path Consistency Note 5.17 it is straightforward to see that a normalized CSP is closed under the applications of the *PATH CONSISTENCY* rules *1, 2* and *3* iff for each subsequence x, y, z of its variables we have

- $C'_{x,y} = C_{x,y}$, where $C'_{x,y}$ is defined as in the *PATH CONSISTENCY 1* rule,
- $C'_{x,z} = C_{x,z}$, where $C'_{x,z}$ is defined as in the *PATH CONSISTENCY 2* rule,
- $C'_{y,z} = C_{y,z}$, where $C'_{y,z}$ is defined as in the *PATH CONSISTENCY 3* rule. □

The notion of path consistency involves two relations in the sense that each relation $C_{x,y}$ is defined in terms of two other relations, $C_{x,z}$ and $C_{y,z}$. It can be easily generalized to involve m relations.

Definition 5.20 We call a normalized CSP *m-path consistent*, where $m \geq 2$, if for each subset $\{x_1, \ldots, x_{m+1}\}$ of its variables we have

$$C_{x_1,x_{m+1}} = \{(a_1, a_{m+1}) \mid \exists a_2, \ldots, a_m \, \forall i \in [1..m](a_i, a_{i+1}) \in C_{x_i,x_{i+1}}\}.$$

□

In other words, a normalized CSP is m-path consistent if for each subset $\{x_1, \ldots, x_{m+1}\}$ of its variables and a pair $(a_1, a_{m+1}) \in C_{x_1,x_{m+1}}$ there exists a sequence of values $a_2, \ldots, a_m$ such that for all $i \in [1..m]$ we have $(a_i, a_{i+1}) \in C_{x_i,x_{i+1}}$. Let us call this sequence $a_2, \ldots, a_m$ a *path connecting a_1 and a_{m+1}*.

Clearly 2-path consistency is identical to path consistency. The following theorem shows that for other values of m the equivalence still holds.

Theorem 5.21 (Path Consistency) *Consider a normalized CSP that is path consistent. Then it is m-path consistent for each $m \geq 2$.*

Proof. We proceed by induction. As already noticed, the claim holds for $m = 2$. Assume the claim holds for some $m \geq 2$. Consider now a subset $\{x_1, \ldots, x_{m+2}\}$ of the variables of this CSP and take a pair $(a_1, a_{m+2}) \in C_{x_1,x_{m+2}}$. By virtue of path consistency applied to the set $\{x_1, x_{m+1}, x_{m+2}\}$ there exists a value a_{m+1} such that $(a_1, a_{m+1}) \in C_{x_1,x_{m+1}}$ and $(a_{m+1}, a_{m+2}) \in C_{x_{m+1},x_{m+2}}$. By virtue of m-path consistency applied to the set $\{x_1, \ldots, x_{m+1}\}$ there exists a path connecting a_1 and a_{m+1}. Consequently, there exists a path connecting a_1 and a_{m+2}.

This concludes the proof of the inductive step and hence the proof of the claim. □

So this generalization of path consistency does not bring anything new.

5.6 Directional Path Consistency

Just as the the notion of arc consistency can be modified to the notion of directional arc consistency, the notion of path consistency can also be modified to a notion that takes into account a linear ordering $\prec$ on the considered variables. This yields directional path consistency defined as follows.

Definition 5.22 Assume a linear ordering $\prec$ on the considered variables. We call a normalized CSP *directionally path consistent w.r.t. $\prec$* if for each subset $\{x, y, z\}$ of its variables we have

$$C_{x,y} = \{(a, b) \mid \exists c \, ((a, c) \in C_{x,z}, (b, c) \in C_{y,z})\} \text{ provided } x \prec z \text{ and } y \prec z.$$

□

So in this definition only specific triples of variables are considered. This definition relies on the supplementary relations because the ordering $\prec$ may differ from the original ordering of the variables. For example, in the original ordering z can precede x. In this case $C_{z,x}$ and not $C_{x,z}$ is a constraint of the CSP under consideration.

But just as in the case of path consistency we can rewrite this definition using the original constraints only. In fact, we have the following analogue of the Alternative Path Consistency Note 5.17.

Note 5.23 (Alternative Directional Path Consistency) *A normalized CSP is directionally path consistent w.r.t. $\prec$ iff for each subsequence x, y, z of its variables we have*

$$C_{x,y} = \{(a,b) \mid \exists c\,((a,c) \in C_{x,z}, (b,c) \in C_{y,z})\} \text{ provided } x \prec z \text{ and } y \prec z,$$

$$C_{x,z} = \{(a,c) \mid \exists b\,((a,b) \in C_{x,y}, (b,c) \in C_{y,z})\} \text{ provided } x \prec y \text{ and } z \prec y,$$

$$C_{y,z} = \{(b,c) \mid \exists a\,((a,b) \in C_{x,y}, (a,c) \in C_{x,z})\} \text{ provided } y \prec x \text{ and } z \prec x.$$

□

Thus out of the above three equalities precisely one is applicable.

To see the difference between the notions of path consistency and directional path consistency consider the following example.

Example 5.24 Reconsider the CSP

$$\langle x < y, y < z, x < z \; ; \; x \in [0..4], y \in [1..5], z \in [5..10]\rangle$$

of Example 5.18(ii). We already noted that it not path consistent. The argument given there also shows that it is not directionally path consistent w.r.t. the ordering $\prec$ in which $x \prec y$ and $z \prec y$.

In contrast, it is easy to see that this CSP is directionally path consistent w.r.t. the ordering $\prec$ in which $x \prec z$ and $y \prec z$. Indeed, for every pair $(a,b) \in C_{x,y}$ there exists $c \in [5..10]$ such that $a < c$ and $b < c$. Namely, we can always take $c = 6$.

Similarly, this CSP is also directionally path consistent w.r.t. the ordering $\prec$ in which $y \prec x$ and $z \prec x$, as for every pair $(b,c) \in C_{y,z}$ there exists $a \in [0..4]$ such that $a < c$ and $b < c$, namely $a = 0$. □

To characterize the notion of directional path consistency we can reuse the *PATH CONSISTENCY* rules *1*,*2* and *3*. In analogy to the case of directional arc consistency we say that the *PATH CONSISTENCY* rule *1* (respectively, rule *2*, or rule *3*) *respects the variable ordering* $\prec$ if $x \prec z$ and $y \prec z$ (respectively $x \prec y$ and $z \prec y$, or $y \prec x$ and $z \prec x$). So for any three variables x, y and z exactly one of the *PATH CONSISTENCY* rules *1, 2* and *3* respects the variable ordering $\prec$.

We then have the following characterization result.

Note 5.25 (Directional Path Consistency) *A CSP is directionally path consistent w.r.t. $\prec$ iff it is closed under the applications of those PATH CONSISTENCY rules 1, 2 and 3 that respect the variable ordering $\prec$.* □

5.7 k-Consistency

The notion of path consistency can be also generalized in a different way. For further discussion we need to be more precise about the way values are assigned to variables. To this end we introduce the following notions.

Definition 5.26 Consider a CSP $\mathcal{P}$.

- By an *instantiation* we mean a mapping defined on a subset of the variables of $\mathcal{P}$ which assigns to each variable a value from its domain. We represent instantiations as sets of the form

  $$\{(x_1, d_1), \ldots, (x_k, d_k)\}.$$

  This notation assumes that $x_1, \ldots, x_k$ are different variables and that for $i \in [1..k]$ d_i is an element from the domain of x_i.

- We say that an instantiation $\{(x_1, d_1), \ldots, (x_k, d_k)\}$ *satisfies a constraint* C *on* $x_1, \ldots, x_k$ if $(d_1, \ldots, d_k) \in C$.

- Consider an instantiation I with a domain X and a sequence Y of different elements of X. We denote by $I \mid Y$ the instantiation obtained by restricting I to the elements of Y and call it a *projection of* I *onto* Y.

- We call an instantiation I with a domain X *consistent* if for every constraint C of $\mathcal{P}$ on a subsequence Y of X the projection $I \mid Y$ satisfies C.

- We call a consistent instantiation *k-consistent* if its domain consists of k variables.

- We call an instantiation a *solution* to $\mathcal{P}$ if it is consistent and defined on all variables of $\mathcal{P}$. □

So an instantiation is a solution to a CSP if it is k-consistent, where k is the number of its variables. Next, we introduce some properties of CSP the study of which is the topic of this section.

Definition 5.27

- We call a CSP 1-*consistent* if every unary constraint on a variable x is satisfied by all the values in the domain of x.

- We call a CSP *k-consistent*, where $k > 1$, if for every $(k \perp 1)$-consistent instantiation and for every variable x not in its domain there exists a value in the domain of x such that the resulting instantiation is k-consistent. □

So, a CSP is 1-consistent if every unary constraint on a variable x coincides with the domain of x. And a CSP is k-consistent where $k > 1$, if every $(k \perp 1)$-consistent instantiation can be extended to a k-consistent instantiation no matter what new variable is selected. 1-consistency is usually called *node consistency*.

The following observation shows that the notion of k-consistency generalizes those of arc consistency and path consistency. By a *binary constraint satisfaction problem* we mean here a CSP all constraints of which are unary or binary.

Note 5.28 (Characterization)

(i) A node consistent CSP is arc consistent iff it is 2*-consistent.*

(ii) A node consistent normalized binary CSP is path consistent iff it is 3*-consistent.* □

It is useful to notice that in general there is no relation between k-consistency and l-consistency for $k \neq l$.

Example 5.29 First notice that for every $k > 1$ there exists a CSP that is $(k \perp 1)$-consistent but not k-consistent.

Indeed, consider a domain $D := [1..k \perp 1]$ and take a sequence of k variables $x_1, \ldots, x_k$, each with the domain D and assume the constraints $x_i \neq x_j$ for $i \in [1..k]$ and $j \in [i+1..k]$. These constraints just state that all variables are different.

It is easy to see that the resulting CSP is $(k \perp 1)$-consistent because every $(k \perp 2)$-consistent instantiation "uses" $k \perp 2$ values out of $[1..k \perp 1]$ so it can be extended to a $(k \perp 1)$-consistent instantiation by assigning the remaing value to the newly selected variable. However, this CSP is not k-consistent as there is no solution to it. □

More interesting is the next example.

Example 5.30 We now show that for every $k > 2$ there exists a CSP that is not $(k \perp 1)$-consistent but is k-consistent.

Indeed, take a sequence of k variables $x_1, \ldots, x_k$, with the domain $\{a, b\}$ for the variable x_1 and with the singleton domain $\{a\}$ for each of the variables $x_2, \ldots, x_k$. Now assume the constraints $x_1 \neq x_2$ and $x_1 \neq x_3$ and the unary constraints $x_l = a$ for $l \in [2..k]$.

Consider now the instantiation the domain of which consists of the variables x_l for $l \neq 2, 3$ and which assigns to each of these variables the value a. Clearly, this instantiation is $(k \perp 2)$-consistent. It can be extended to an instantiation of $k \perp 1$ variables either by adding the pair (x_2, a) or the pair (x_3, a) and in each case the resulting instantiation is not $(k \perp 1)$-consistent. So this CSP is not $(k \perp 1)$-consistent.

However, this CSP is k-consistent. Indeed, every $(k \perp 1)$-consistent instantiation is a projection of the consistent instantiation

$$\{(x_1, b), (x_2, a), (x_3, a), \ldots, (x_k, a)\}.$$

□

As in the case of the previously considered notions of local consistency, it is possible to characterize the notion of k-consistency by means of rules, as well. However, these rules are rather complex, so we omit them.

5.8 Strong k-Consistency

So a natural question arises what is the use of k-consistency. After all we have just noticed that k-consistency, even for a large, relative to the number of variables, k does not guarantee yet that the CSP is consistent. This seems to indicate that in general there is no relation between k-consistency and consistency. This is, however, not completely true. What we need is an accumulated effect of k-consistencies.

This brings us to the following definition.

Definition 5.31 We call a CSP *strongly k-consistent*, where $k \geq 1$, if it is i-consistent for every $i \in [1..k]$. □

Now, the following simple result relates strong consistency and consistency.

Theorem 5.32 (Consistency 1) *Consider a CSP with k variables, where $k \geq 1$, such that*

- *at least one domain is non-empty,*
- *it is strongly k-consistent.*

Then this CSP is consistent.

Proof. We construct a solution to the considered CSP by induction. More precisely, we prove that

(i) there exists a 1-consistent instantiation,

(ii) for every $i \in [2..k]$ each $(i \perp 1)$-consistent instantiation can be extended to an i-consistent instantiation.

Rearrange the variables in question in a sequence $x_1, \ldots, x_k$ such that the domain D_1 of x_1 is non-empty. Since the considered CSP is 1-consistent and the domain D_1 is non-empty, there exists $d_1 \in D_1$ such that $\{(x_1, d_1)\}$ is 1-consistent. This proves (i).

Now (ii) is simply a consequence of the fact that the CSP is i-consistent for every $i \in [2..k]$. □

Note that the assumption that at least one domain is non-empty is essential. Indeed, every CSP with all domains empty is trivially i-consistent for every $i \in [1..k]$, where k is the number of variables. Yet it has no solution.

In general, a CSP can involve a large number of variables, so to use the above theorem one either needs to check strong k-consistency or to reduce the given CSP to a strong k-consistent one, in both cases for a large k. We shall see now that in practice one can use a much smaller k.

5.9 Graphs and CSP's

In this subsection we show how a graph-theoretic analysis leads to an improvement of the Consistency 1 Theorem 5.32.

Recall that a *graph* is a pair (N, A) where N is a non-empty set and A is a relation on N, that is a subset of the Cartesian product $N \times N$. We call the elements of N *nodes* and the elements of A *arcs*.

Definition 5.33 Take a CSP $\mathcal{P}$. Consider a graph defined as follows. Its nodes are all the variables of $\mathcal{P}$ and two variables are connected by an arc if there is a constraint that involves them. We say then that this graph is *associated with* $\mathcal{P}$. □

Let us consider some examples.

Example 5.34
(i) Consider the *SEND + MORE = MONEY* puzzle discussed in Example 2.1. The graph associated with the first CSP presented there has eight nodes, namely the variables S, E, N, D, M, O, R, Y and is complete, that is every two nodes are connected by an arc. This is already so even if we disregard all disequality constraints —the only equality constraint already involves all eight variables.

(ii) The graph associated with a CSP with two constraints, $x + y = z$ and $x + u = v$ is depicted in Figure 5.2.

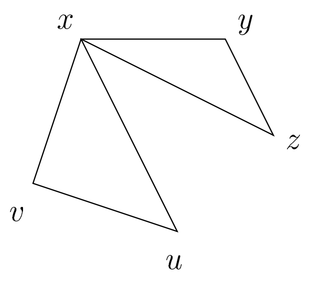

Figure 5.2: A graph associated with a CSP

(iii) Finally, the graph associated with a CSP with four constraints, $x < z, x < y, y < u, y < v$ is depicted in Figure 5.3. □

Next, we associate with each graph a numeric value.

Definition 5.35 Consider a finite graph G. Assume a linear ordering $\prec$ on its nodes.

- The $\prec$-*width* of a node of G is the number of arcs in G that connect it to $\prec$-smaller nodes.
- The $\prec$-*width* of G is the maximum of the $\prec$-widths of its nodes.

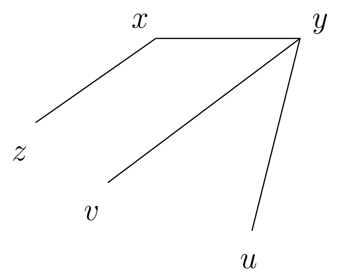

Figure 5.3: Another graph associated with a CSP

- The *width* of G is the minimum of its $\prec$-widths for all linear orderings $\prec$ on its nodes. □

So for example the width of a graph that is a tree is 1 and the width of a complete graph with n nodes is $n \perp 1$.

To clarify this notion let us now consider the graphs discussed in Example 5.34.

Example 5.36
(i) Consider the graph discussed in Example 5.34(i). It is a complete graph with 8 nodes, so its width is 7.

(ii) Next, the width of the graph discussed in Example 5.34(ii) and depicted in Figure 5.2 is 2. Indeed, consider any ordering $\prec$ of the variables in which x is the first variable. Then the $\prec$-width of this graph is 2. Further, the width of this graph is at least 2 since it contains a complete subgraph with 3 nodes.

(iii) Finally, the width of the graph discussed in Example 5.34(iii) and depicted in Figure 5.3 is 1, since it is a tree. □

We now prove the following result that greatly improves upon the Consistency 1 Theorem 5.32.

Theorem 5.37 (Consistency 2) *Consider a CSP such that*

- *all its domains are non-empty,*
- *it is strongly k-consistent,*
- *the graph associated with it has width $k \perp 1$.*

Then this CSP is consistent.

Proof. Let $\prec$ be a linear ordering on the variables of this CSP such that the $\prec$-width of the graph associated with this CSP is $k \perp 1$. Let $x_1, \ldots, x_n$ be the sequence of the variables of this CSP linearly ordered by $\prec$ and $D_1, \ldots, D_n$ the corresponding domains.

We now prove that

(i) there exists a consistent instantiation with the domain $\{x_1\}$,

(ii) for every $j \in [1..n \perp 1]$ each consistent instantiation with the domain $\{x_1, \ldots, x_j\}$ can be extended to a consistent instantiation with the domain $\{x_1, \ldots, x_{j+1}\}$.

Since the considered CSP is 1-consistent and the domain D_1 is non-empty, there exists some $d_1 \in D_1$ such that $\{(x_1, d_1)\}$ is consistent. This proves (i).

To prove (ii) let Y be the set of variables in $\{x_1, \ldots, x_j\}$ that are connected by an arc with x_{j+1}. By the definition of the width of a graph the set Y has at most $k \perp 1$ elements.

Let now I be a consistent instantiation with the domain $\{x_1, \ldots, x_j\}$ and consider its projection $I \mid Y$ on Y. By the definition of k-strong consistency for some $d_{j+1} \in D_{j+1}$ $I \mid Y \cup \{(x_{j+1}, d_{j+1})\}$ is a consistent instantiation.

Now, each constraint on a sequence of variables from $\{x_1, \ldots, x_{j+1}\}$ is either on a sequence of variables from $\{x_1, \ldots, x_j\}$ or from $Y \cup \{x_{j+1}\}$. In both cases the corresponding projection of $I \cup \{(x_{j+1}, d_{j+1})\}$ is consistent. So the instantiation $I \cup \{(x_{j+1}, d_{j+1})\}$ itself is consistent. □

The above theorem is indeed an improvement over the Consistency 1 Theorem 5.32 because it rarely happens that for a CSP the width of its graph equals the number of its variables minus one. One such case is when the graph associated with a CSP is complete.

To understand better the use of this theorem consider under its assumptions the situation when $k = 2$. In this case the graph associated with this CSP is a tree. So by virtue of the Characterization Note 5.28(i) we can conclude that for a CSP such that all its domains are non-empty and the graph associated with it is a tree, node and arc consistency imply consistency. In fact, we can slightly improve this observation as follows.

Theorem 5.38 (Directional Arc Consistency) *Consider a CSP such that*

- *all its domains are non-empty,*
- *it is node consistent and directionally arc consistent w.r.t. some ordering $\prec$,*
- *the $\prec$-width of the graph associated with it is 1, (i.e., this graph is a tree).*

Then this CSP is consistent.

Proof. It is a straightforward specialization of the proof of the Consistency 2 Theorem 5.37 for $k = 2$. In this case the set Y considered in its proof is either empty or a singleton. □

An analogous improvement can be made for the case $k = 3$.

Theorem 5.39 (Directional Path Consistency) *Consider a normalized binary CSP such that*

- *all its domains are non-empty,*
- *it is node consistent,*

- *it is directionally arc consistent and directionally path consistent w.r.t. some ordering $\prec$,*
- *the $\prec$-width of the graph associated with it is 2.*

Then this CSP is consistent.

Proof. The result follows by a simple analysis of the proof of the Consistency 2 Theorem 5.37 for $k = 3$. In this case the set Y considered in this proof has $0, 1$ or 2 elements. We then rely on, respectively, node, directional arc, and directional path consistency. $\square$

5.10 Exercises

Exercise 1 Prove the Equivalence Lemma 5.2. $\square$

Exercise 2 Prove the Directional Arc Consistency Note 5.14. $\square$

Exercise 3 Prove the Alternative Path Consistency Note 5.17. $\square$

Exercise 4 Prove the Alternative Directional Path Consistency 5.23. $\square$

Exercise 5 Prove the Directional Path Consistency Note 5.25. $\square$

Exercise 6 Prove the Characterization Note 5.28. $\square$

5.11 Bibliographic Remarks

The proof theoretic framework of Section 5.1 is from Apt (1998).

The notion of arc consistency was introduced in Mackworth (1977). The notions of directional arc consistency and directional path consistency are due to Dechter & Pearl (1988), where also the Directional Arc Consistency Theorem 5.38 and Directional Path Consistency Theorem 5.38 were established. In turn, the notion of hyper-arc consistency is from Mohr & Masini (1988) where it is called arc consistency. This notion also appears implicitly in Davis (1987) and the adopted terminology follows Marriott & Stuckey (1998).

Further, the notions of path consistency and m-path consistency are due to Montanari (1974) where the Path Consistency Theorem 5.21 was proved. In turn, the notion of k-consistency is from Freuder (1978) while the Characterization Note 5.28, the notion of strong k-consistency, and the Consistency 2 Theorem 5.37 are from Freuder (1982).

References

Apt, K. R. (1998), ‘A proof theoretic view of constraint programming’, *Fundamenta Informaticae* **33**(3), 263–293. Available via `http://www.cwi.nl/~apt`.

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

Dechter, R. & Pearl, J. (1988), ‘Network-based heuristics for constraint-satisfaction problems’, *Artificial Intelligence* **34**(1), 1–38.

Freuder, E. (1978), ‘Synthesizing constraint expressions’, *Communications of the ACM* **21**, 958–966.

Freuder, E. (1982), ‘A sufficient condition for backtrack-free search’, *Journal of the ACM* **29**(1), 24–32.

Mackworth, A. (1977), ‘Consistency in networks of relations’, *Artificial Intelligence* **8**(1), 99–118.

Marriott, K. & Stuckey, P. (1998), *Programming with Constraints*, The MIT Press, Cambridge, Massachusetts.

Mohr, R. & Masini, G. (1988), Good old discrete relaxation, *in* Y. Kodratoff, ed., ‘Proceedings of the 8th European Conference on Artificial Intelligence (ECAI)’, Pitman Publishers, pp. 651–656.

Montanari, U. (1974), ‘Networks of constraints: Fundamental properties and applications to picture processing’, *Information Science* **7**(2), 95–132. Also Technical Report, Carnegie Mellon University, 1971.

Chapter 6

Some Incomplete Constraint Solvers

For some specific domains and constraints for which no efficient solving methods exist or are known to exist specialized techniques have been developed.

The aim of this chapter is to illustrate such techniques for two most popular domains. These methods amount to incomplete constraint solvers in the terminology of Chapter 4. We define these solvers using the theoretic framework introduced in Section 5.1 so by means of the proof rules that work on CSP's.

In order to use this proof theoretic framework for specific domains the rules have to be "customized" to a specific language in which constraints are defined and to specific domains. In what follows we present such a customization for Boolean constraints and for linear constraints over integer intervals and finite integer domains.

In each case we illustrate the use of the introduced techniques by means of examples. Then we relate the introduced proof rules to a notion of local consistency. This allows us to view these specialized techniques as methods of achieving some form of local consistency.

In any implementation the rules introduced in this chapter should be selected and scheduled in an appropriate way. This can be done by using any of the algorithms discussed in the next chapter.

6.1 Boolean Constraints

Boolean constraint satisfaction problems deal with Boolean variables and constraints on them defined by means of Boolean connectives and equality. We already referred to these CSP's at various places but let us define them now formally.

By a *Boolean variable* we mean a variable which ranges over the domain that consists of two values: 0 denoting **false** and 1 denoting **true**. By a *Boolean domain expression* we mean an expression of the form $x \in D$ where $D \subseteq \{0, 1\}$. In what follows we write the Boolean domain expression $x \in \{1\}$ as $x = 1$ and $x \in \{0\}$ as $x = 0$. By a *Boolean expression* we mean an expression built out of Boolean variables using three connectives: $\neg$ (*negation*), $\wedge$ (*conjunction*) and $\vee$ (*disjunction*).

The only relation symbol in the language is the equality symbol $=$. Next, by a *Boolean constraint* we mean a formula of the form $s = t$, where s, t are Boolean expressions. In presence of Boolean domain expressions each Boolean constraint (so a formula) uniquely determines a constraint (so a set) on the sequence of its variables, so from now on we identify each Boolean constraint with the constraint determined by it.

Finally, by a *Boolean constraint satisfaction problem*, in short *Boolean CSP*, we mean a CSP with Boolean domain expressions and all constraints of which are Boolean constraints.

Note that in our framework the Boolean constants, **true** and **false**, are absent. They can be easily modeled by using two predefined variables, say x_T and x_F, with fixed Boolean domain expressions $x_T = 1$ and $x_F = 0$.

Usually, one studies Boolean expressions, more often known under the name of *propositional formulas*. By studying Boolean CSP's instead of Boolean expressions we do not lose any expressiveness since a Boolean expression s can be modelled by a CSP

$$\psi_s := \langle s = x_T \ ; \ x_1 \in \{0,1\}, \ldots, x_n \in \{0,1\}, x_y = 1\rangle$$

where $x_1, \ldots, x_n$ are the variables of s. Then s is satisfiable (i.e., becomes true for some assignment of truth values to its variables) iff ψ_s has a solution.

There are many ways to deal with Boolean constraints. The one we discuss here separates them into two classes: those that are in a "simple" form and those that are in a "compound" form. Those in a simple form will be dealt with directly; the latter ones will be reduced to the former ones.

In the sequel x, y, z denote different Boolean variables. We call a Boolean constraint *simple* if it is in one of the following form:

- $x = y$; we call it the *equality* constraint,
- $\neg x = y$; we call it the *NOT* constraint,
- $x \wedge y = z$; we call it the *AND* constraint,
- $x \vee y = z$; we call it the *OR* constraint.

6.1.1 Transformation Rules

By introducing auxiliary variables it is straightforward to transform each Boolean CSP into an equivalent one all constraints of which are simple. This process, as already discussed in Chapter 3, can be viewed as a preprocessing stage, or the procedure Preprocess of the Solve procedure of Section 3.1.

More precisely, we adopt the following two transformation rules for negation:

$$\frac{\langle \neg s = t \ ; \ \mathcal{DE}\rangle}{\langle \neg x = t, \ s = x \ ; \ \mathcal{DE}, x \in \{0,1\}\rangle}$$

where s is not a variable and where x does not appear in $\mathcal{DE}$ (in short, x is a fresh variable),

$$\frac{\langle \neg s = t \ ; \ \mathcal{DE}\rangle}{\langle \neg s = y, \ t = y \ ; \ \mathcal{DE}, y \in \{0,1\}\rangle}$$

where t is not a variable and where y is a fresh variable.

The following transformation rules for the binary connectives are equally straightforward. Here *op* stands for $\wedge$ or for $\vee$.

$$\frac{\langle s\ op\ t = u\ ;\ \mathcal{DE}\rangle}{\langle s\ op\ t = z,\ u = z\ ;\ \mathcal{DE}, z \in \{0,1\}\rangle}$$

where u is not a variable or is a variable identical to s or t, and where z is a fresh variable,

$$\frac{\langle s\ op\ t = u\ ;\ \mathcal{DE}\rangle}{\langle x\ op\ t = u,\ s = x\ ;\ \mathcal{DE}, x \in \{0,1\}\rangle}$$

where s is not a variable or is a variable identical to t or u, and where x is a fresh variable,

$$\frac{\langle s\ op\ t = u\ ;\ \mathcal{DE}\rangle}{\langle s\ op\ y = u,\ t = y\ ;\ \mathcal{DE}, y \in \{0,1\}\rangle}$$

where t is not a variable or is a variable identical to s or u, and where y is a fresh variable.

It is straightforward to prove that each these rules is equivalence preserving w.r.t. to the set of the variables of the premise (see Exercise 7).

6.1.2 Domain Reduction Rules

We now introduce proof domain reduction rules that deal with the simple Boolean constraints. We write these rules in a simplified form already suggested in Subsection 3.1.5 that we illustrate by means of three representative examples.

We write the solving rule

$$\frac{\langle \neg x = y\ ;\ x \in D_x, y = 0\rangle}{\langle\ ;\ x \in D_x \cap \{1\}, y = 0\rangle}$$

as

$$\neg x = y, y = 0 \rightarrow x = 1,$$

the already mentioned in Section 3.1.5 transformation rule

$$\frac{\langle x \wedge y = z\ ;\ x = 1, y \in D_y, z \in D_z\rangle}{\langle y = z\ ;\ x = 1, y \in D_y, z \in D_z\rangle}$$

as

$$x \wedge y = z, x = 1 \rightarrow z = y,$$

and the solving rule

$$\frac{\langle x \vee y = z\ ;\ x = 0, y \in D_y, z = 1\rangle}{\langle\ ;\ x = 0, y \in D_y \cap \{1\}, z = 1\rangle}$$

as

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

EQU 1	$x = y, x = 1 \to y = 1$
EQU 2	$x = y, y = 1 \to x = 1$
EQU 3	$x = y, x = 0 \to y = 0$
EQU 4	$x = y, y = 0 \to x = 0$
NOT 1	$\neg x = y, x = 1 \to y = 0$
NOT 2	$\neg x = y, x = 0 \to y = 1$
NOT 3	$\neg x = y, y = 1 \to x = 0$
NOT 4	$\neg x = y, y = 0 \to x = 1$
AND 1	$x \wedge y = z, x = 1, y = 1 \to z = 1$
AND 2	$x \wedge y = z, x = 1, z = 0 \to y = 0$
AND 3	$x \wedge y = z, y = 1, z = 0 \to x = 0$
AND 4	$x \wedge y = z, x = 0 \to z = 0$
AND 5	$x \wedge y = z, y = 0 \to z = 0$
AND 6	$x \wedge y = z, z = 1 \to x = 1, y = 1$
OR 1	$x \vee y = z, x = 1 \to z = 1$
OR 2	$x \vee y = z, x = 0, y = 0 \to z = 0$
OR 3	$x \vee y = z, x = 0, z = 1 \to y = 1$
OR 4	$x \vee y = z, y = 0, z = 1 \to x = 1$
OR 5	$x \vee y = z, y = 1 \to z = 1$
OR 6	$x \vee y = z, z = 0 \to x = 0, y = 0$

Table 6.1: Proof system *BOOL*

Using this convention we now introduce 20 solving rules presented in Table 6.1. We call the resulting proof system *BOOL*.

To read properly such formulas it helps to remember that 0 and 1 are domain elements, so atomic formulas of the form $x = 0$ and $x = 1$ are domain expressions while all other atomic formulas are constraints. Intuitively, the *AND 1* rule should be interpreted as: $x \wedge y = z, x = 1$ and $y = 1$ imply that z *becomes* 1. Additionally, as a side effect of applying such a rule, the constraint — here $x \wedge y = z$ — is deleted. And analogously with the other rules.

Note also that each of these rules can yield a failed CSP —take for instance the *NOT 2* rule. When applied to the CSP $\langle \neg x = y \ ; \ x = 0, y = 0\rangle$ it yields $\langle \ ; \ x = 0, y \in \emptyset\rangle$.

Further, observe that no rule is introduced for $x \wedge y = z$ when $z = 0$. In this case either $x = 0$ or $y = 0$, but $x = 0 \vee y = 0$ is not a legal Boolean domain expression. Alternatively, either $x = z$ or $y = z$ holds, but rewriting $x \wedge y = z$ into $x = z \vee y = z$ is not a legal step, since the latter is not a simple Boolean constraint. The same considerations apply to $x \vee y = z$ when $z = 1$.

Finally, note that some Boolean CSP's that are closed under these rules are neither failed nor successful even if some of the domains is a singleton. Take for instance the CSP $\langle x \wedge y = z \ ; \ x \in \{0, 1\}, y \in \{0, 1\}, z = 0\rangle$ to which no rule applies.

In Subsection 6.1.4 we shall provide a characterization of the Boolean CSP's that are

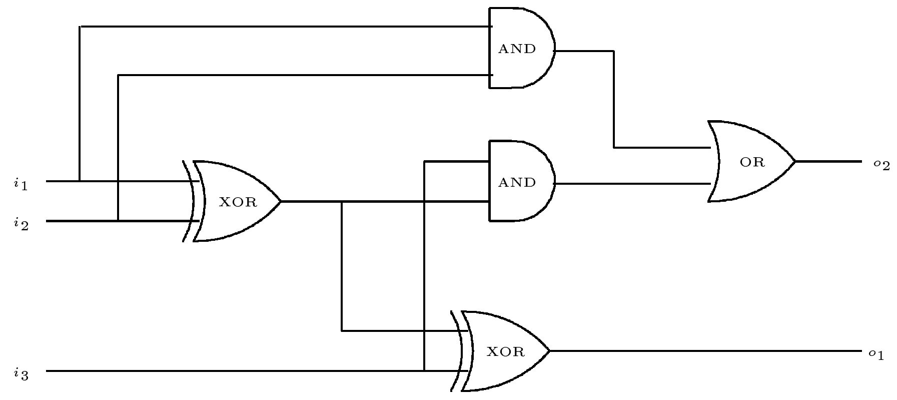

Figure 6.1: Full adder circuit

closed under the rules of the system *BOOL*.

6.1.3 Example: Full Adder Circuit

We now show how we can reason using the proof rules for Boolean CSP's about the full adder circuit originally introduced in Example 2.6 of Chapter 2 and depicted in Figure 6.1.

More precisely, we show how in the case of the Boolean constraints

$$(i_1 \oplus i_2) \oplus i_3 = o_1,$$

$$(i_1 \wedge i_2) \vee (i_3 \wedge (i_1 \oplus i_2)) = o_2$$

representing this circuit we can conclude that $i_1 = 1, i_2 = 1$ and $o_1 = 0$ follows from the assumption that $i_3 = 0$ and $o_2 = 1$.

As a first step, we transform the above two constraints using the transformation rules into the following five constraints:

$$i_1 \oplus i_2 = x_1,$$

$$i_1 \wedge i_2 = y_1,$$

$$x_1 \oplus i_3 = o_1,$$

$$i_3 \wedge x_1 = y_2,$$

$$y_1 \vee y_2 = o_2,$$

where we used an informal optimization and used only one variable, x_1, to represent $i_1 \oplus i_2$.

To reason about the Boolean circuits we need some rules that deal with the exclusive disjunction. For the purpose of the above example we need just two rules that are presented in Table 6.2 that we add to the system *BOOL*.

XOR1	$x \oplus y = z, x = 1, y = 1 \rightarrow z = 0.$
XOR2	$x \oplus y = z, x = 0, y = 0 \rightarrow z = 0.$

Table 6.2: Proof rules for *XOR*

We can now derive the desired conclusion by means of the following successful derivation of five steps. The selected constraints are underlined. The original CSP, so

$$\langle i_1 \oplus i_2 = x_1, i_1 \wedge i_2 = y_1, x_1 \oplus i_3 = o_1, \underline{i_3 \wedge x_1 = y_2}, y_1 \vee y_2 = o_2 \ ; \ i_3 = 0, o_2 = 1\rangle$$

gets transformed by the *AND 4* rule into

$$\langle i_1 \oplus i_2 = x_1, i_1 \wedge i_2 = y_1, x_1 \oplus i_3 = o_1, \underline{y_1 \vee y_2 = o_2} \ ; \ i_3 = 0, o_2 = 1, y_2 = 0\rangle.$$

By the *OR 4* rule we now get

$$\langle i_1 \oplus i_2 = x_1, \underline{i_1 \wedge i_2 = y_1}, x_1 \oplus i_3 = o_1 \ ; \ i_3 = 0, o_2 = 1, y_2 = 0, y_1 = 1\rangle.$$

Next, applying the *AND 6* rule we get

$$\langle \underline{i_1 \oplus i_2 = x_1}, x_1 \oplus i_3 = o_1 \ ; \ i_3 = 0, o_2 = 1, y_2 = 0, y_1 = 1, i_1 = 1, i_2 = 1\rangle.$$

Using the *XOR 1* rule we now obtain

$$\langle \underline{x_1 \oplus i_3 = o_1} \ ; \ i_3 = 0, o_2 = 1, y_2 = 0, y_1 = 1, i_1 = 1, i_2 = 1, x_1 = 0\rangle.$$

Finally, by the *XOR 2* rule we conclude

$$\langle \ ; \ i_3 = 0, o_2 = 1, y_2 = 0, y_1 = 1, i_1 = 1, i_2 = 1, x_1 = 0, o_1 = 0\rangle.$$

Because at each step the equivalence is maintained, we conclude that for every solution to the original CSP we have $i_1 = 1, i_2 = 1$ and $o_1 = 0$.

6.1.4 A Characterization of the System *BOOL*

It is natural to ask in what sense the proof rules of the system *BOOL* are complete. After all, perhaps some obvious rule is missing and its absence does not allow us to draw some specific conclusion. To provide an answer to this question we establish the following result that uses the notion of hyper-arc consistency introduced in Section 5.3.

Theorem 6.1 (*BOOL) A non-failed Boolean CSP is hyper-arc consistent iff it is closed under the applications of the rules of the proof system BOOL.*

That is, a non-failed Boolean CSP is hyper-arc consistent iff it is a final CSP in a derivation for the proof system *BOOL*.
Proof. Let ϕ be the CSP under consideration. Below $C := x \wedge y = z$ is some *AND* constraint belonging to ϕ. We view it as a constraint on the variables x, y, z in this order. Let D_x, D_y and D_z be respectively the domains of x, y and z.

($\Rightarrow$) We need to consider each rule in turn. We analyse here only the *AND* rules. For other rules the reasoning is similar.

AND 1 rule.

Suppose that $D_x = \{1\}$ and $D_y = \{1\}$. If $0 \in D_z$, then by the hyper-arc consistency for some $d_1 \in D_x$ and $d_2 \in D_y$ we have $(d_1, d_2, 0) \in C$, so $(1, 1, 0) \in C$ which is a contradiction.

This shows that $D_z = \{1\}$ which means that ϕ is closed under the applications of this rule.

AND 2 rule.

Suppose that $D_x = \{1\}$ and $D_z = \{0\}$. If $1 \in D_y$, then by the hyper-arc consistency for some $d_1 \in D_x$ and $d_2 \in D_z$ we have $(d_1, 1, d_2) \in C$, so $(1, 1, 0) \in C$ which is a contradiction.

This shows that $D_y = \{0\}$ which means that ϕ is closed under the applications of this rule.

AND 3 rule.

This case is symmetric to that of the *AND 2* rule.

AND 4 rule.

Suppose that $D_x = \{0\}$. If $1 \in D_z$, then by the hyper-arc consistency for some $d_1 \in D_x$ and $d_2 \in D_y$ we have $(d_1, d_2, 1) \in C$, so $(1, 1, 1) \in C$ which is a contradiction.

This shows that $D_z = \{0\}$ which means that ϕ is closed under the applications of this rule.

AND 5 rule.

This case is symmetric to that of the *AND 4* rule.

AND 6 rule.

Suppose that $D_z = \{1\}$. If $0 \in D_x$, then by the hyper-arc consistency for some $d_1 \in D_y$ and $d_2 \in D_z$ we have $(0, d_1, d_2) \in C$, so $0 \in D_z$ which is a contradiction.

This shows that $D_x = \{1\}$. By a symmetric argument also $D_y = \{1\}$ holds. This means that ϕ is closed under the applications of this rule.

($\Leftarrow$) Consider the *AND* constraint C. We have to analyze six cases.

Case 1. Suppose $1 \in D_x$.

Assume that neither $(1, 1) \in D_y \times D_z$ nor $(0, 0) \in D_y \times D_z$. Then either $D_y = \{1\}$ and $D_z = \{0\}$ or $D_y = \{0\}$ and $D_z = \{1\}$.

If the former holds, then by the *AND 3* rule we get $D_x = \{0\}$ which is a contradiction. If the latter holds, then by the *AND 5* rule we get $D_z = \{0\}$ which is a contradiction.

We conclude that for some d we have $(1, d, d) \in C$.

Case 2. Suppose $0 \in D_x$.

Assume that $0 \notin D_z$. Then $D_z = \{1\}$, so by the *AND 6* rule we get $D_x = \{1\}$ which is a contradiction. Hence $0 \in D_z$. Let now d be some element of D_y. We then have $(0, d, 0) \in C$.

Case 3. Suppose $1 \in D_y$.

This case is symmetric to Case 1.

Case 4. Suppose $0 \in D_y$.

This case is symmetric to Case 2.

Case 5. Suppose $1 \in D_z$.

Assume that $(1,1) \notin D_x \times D_y$. Then either $D_x = \{0\}$ or $D_y = \{0\}$. If the former holds, then by the *AND 4* rule we conclude that $D_z = \{0\}$. If the latter holds, then by the *AND 5* rule we conclude that $D_z = \{0\}$. For both possibilities we reached a contradiction. So both $1 \in D_x$ and $1 \in D_y$ and consequently $(1,1,1) \in C$.

Case 6. Suppose $0 \in D_z$.

Assume that both $D_x = \{1\}$ and $D_y = \{1\}$. By the rule *AND 1* rule we conclude that $D_z = \{1\}$ which is a contradiction. So either $0 \in D_x$ or $0 \in D_y$ and consequently for some d either $(0,d,0) \in C$ or $(d,0,0) \in C$.

An analogous reasoning can be spelled out for the equality, *OR* and *NOT* constraints and is omitted. □

Note that the restriction to non-failed CSP's is necessary. For example, the failed CSP

$$\langle x \wedge y = z \ ; \ x \in \emptyset, y \in \{0,1\}, z \in \{0,1\}\rangle$$

is not hyper-arc consistent but it is closed under the applications of the proof rules of *BOOL*.

It is also easy to check that all the proof rules of the *BOOL* system are needed, that is, the above result does not hold when any of these 20 rules is omitted. For example, if the rule *AND 4* is left out, then the CSP $\langle x \wedge y = z \ ; \ x = 0, y \in \{0,1\}, z \in \{0,1\}\rangle$ is closed under the applications of all remaining rules but is not hyper-arc consistent.

The above theorem shows that in order to reduce a Boolean CSP to an equivalent one that is either failed or hyper-arc consistent it suffices to close it under the applications of the rules of the *BOOL* system.

6.2 Linear Constraints on Integer Intervals

As an example of another incomplete constraint solver we now consider linear constraints over domains that consist of integer intervals. As an example we analyse the *SEND + MORE = MONEY* puzzle discussed in Example 2.1 of Chapter 2.

First, let us introduce the relevant notions. Using an infinite first-order language that contained a constant for each real we defined in Section 4.2 linear expressions over reals. Here we are interested in linear expressions over integers. We define them in a slightly different way which shows that the underlying first-order language can be chosen to be finite.

By a *linear expression* we mean a term in the language that contains two constants 0 and 1, the unary minus function $\perp$ and two binary functions "+" and "$\perp$", both written in the infix notation. We abbreviate terms of the form

$$\underbrace{1 + \ldots + 1}_{n \text{ times}}$$

to n, terms of the form

$$\underbrace{x + \ldots + x}_{n \text{ times}}$$

to nx and analogously with $\perp 1$ and $\perp x$ used instead of 1 and x. So each linear expression can be equivalently rewritten to an expression in the form

$$\Sigma_{i=1}^{n} a_i x_i + b$$

where $n \geq 0$, $a_1, \ldots, a_n$ are non-zero integers, $x_1, \ldots, x_n$ are different variables and b is an integer.

By a *linear constraint* we mean a formula of the form

$$s \; op \; t$$

where s and t are linear expressions and $op \in \{<, \leq, =, \neq, \geq, >\}$. For example

$$4x + 3y \perp x \leq 5z \perp 7 \perp y \qquad (6.1)$$

is a linear constraint.

In what follows we drop the qualification "linear" when discussing linear expressions and linear constraints.

Further, we call

- $s < t$ and $s > t$ *strict inequality constraints*,
- $s \leq t$ and $s \geq t$ *inequality constraints*,
- $s = t$ an *equality constraint*, or an *equation*,
- $s \neq t$ a *disequality constraint*,
- $x \neq y$, for variables x, y, a *simple disequality constraint.*

By an *integer interval*, or an *interval* in short, we mean an expression of the form

$$[a..b]$$

where a and b are integers; $[a..b]$ denotes the set of all integers between a and b, including a and b. If $a > b$, we call $[a..b]$ the *empty interval.*

Finally, by a *range* we mean an expression of the form

$$x \in I$$

where x is a variable and I is an interval. We abbreviate $x \in [a..b]$ to $x = a$ if $a = b$ and write $x \in \emptyset$ if $a > b$.

In what follows we discuss various rules that allow us to manipulate linear constraints over interval domains. We assume that all considered linear constraints have at least one variable.

6.2.1 Reduction Rules for Inequality Constraints

We begin with the inequality constraints. As the heavy notation can blur the simple idea behind consider first an illustrative special case.

Take the constraint

$$3x + 4y \perp 5z \leq 7$$

with the domain expressions $x \in [l_x..h_x], y \in [l_y..h_y], z \in [l_z..h_z]$. We can rewrite it as

$$x \leq \frac{7 \perp 4y + 5z}{3}$$

Any value of x that satisfies this constraint also satisfies the inequality

$$x \leq \frac{7 \perp 4l_y + 5h_z}{3}$$

where we maximized the value of the right-hand side by minimizing y and maximizing z w.r.t. their domains.

Because we seek integer solutions, we can in fact write

$$x \leq \lfloor \frac{7 \perp 4l_y + 5h_z}{3} \rfloor$$

So we can reduce the interval $[l_x..h_x]$ to $[l_x..min(\lfloor \frac{7-4l_y+5h_z}{3} \rfloor, h_x)]$ without losing any solution. A similar procedure can be applied to the variables y and z, though in the case of z we should remember that a division by a negative number leads to the change of $\leq$ to $\geq$.

After this example consider now a general case. Using appropriate transformation rules that are pretty straightforward and omitted, each inequality constraint can be equivalently written in the form

$$\sum_{i \in POS} a_i x_i \perp \sum_{i \in NEG} a_i x_i \leq b \tag{6.2}$$

where

- a_i is a positive integer for $i \in POS \cup NEG$,
- x_i and x_j are different variables for $i \neq j$ and $i, j \in POS \cup NEG$,
- b is an integer.

For example, (6.1) can be equivalently written as $3x + 4y \perp 5z \leq 7$. Assume the ranges

$$x_i \in [l_i..h_i]$$

for $i \in POS \cup NEG$.

Choose now some $j \in POS$ and let us rewrite (6.2) as

$$x_j \leq \frac{b \perp \sum_{i \in POS - \{j\}} a_i x_i + \sum_{i \in NEG} a_i x_i}{a_j}$$

Computing the maximum of the expression on the right-hand side w.r.t. the ranges of the involved variables we get

$$x_j \leq \alpha_j$$

where

$$\alpha_j := \frac{b \perp \sum_{i \in POS-\{j\}} a_i l_i + \sum_{i \in NEG} a_i h_i}{a_j}$$

so, since the variables assume integer values,

$$x_j \leq \lfloor \alpha_j \rfloor.$$

We conclude that

$$x_j \in [l_j..min(h_j, \lfloor \alpha_j \rfloor)].$$

By analogous calculations we conclude for $j \in NEG$

$$x_j \geq \lceil \beta_j \rceil$$

where

$$\beta_j := \frac{\perp b + \sum_{i \in POS} a_i l_i \perp \sum_{i \in NEG-\{j\}} a_i h_i}{a_j}$$

In this case we conclude that

$$x_j \in [max(l_j, \lceil \beta_j \rceil)..h_j].$$

This brings us to the following reduction rule for inequality constraints:

LINEAR INEQUALITY 1

$$\frac{\langle \sum_{i \in POS} a_i x_i \perp \sum_{i \in NEG} a_i x_i \leq b \ ; \ x_1 \in [l_1..h_1], \ldots, x_n \in [l_n..h_n] \rangle}{\langle \sum_{i \in POS} a_i x_i \perp \sum_{i \in NEG} a_i x_i \leq b \ ; \ x_1 \in [l'_1..h'_1], \ldots, x_n \in [l'_n..h'_n] \rangle}$$

where for $j \in POS$

$$l'_j := l_j, \ h'_j := min(h_j, \lfloor \alpha_j \rfloor)$$

and for $j \in NEG$

$$l'_j := max(l_j, \lceil \beta_j \rceil), \ h'_j := h_j.$$

The domain reduction rule dealing with the linear inequalities of the form $x < y$ and discussed in Subsection 3.1.5 is an instance of this rule.

6.2.2 Reduction Rules for Equality Constraints

Each equality constraint can be equivalently written as two inequality constraints. By combining the corresponding reduction rules for these two inequality constraints we obtain a reduction rule for an equality constraint. More specifically, each equality constraint can be equivalently written in the form

$$\sum_{i \in POS} a_i x_i \perp \sum_{i \in NEG} a_i x_i = b \tag{6.3}$$

where we adopt the conditions that follow (6.2) in the previous subsection.

Assume now the ranges

$$x_1 \in [l_1..h_1], \ldots, x_n \in [l_n..h_n].$$

We infer then both the ranges in the conclusion of the *LINEAR INEQUALITY 1* reduction rule and the ranges for the linear inequality

$$\sum_{i \in NEG} a_i x_i \perp \sum_{i \in POS} a_i x_i \leq \perp b$$

which are

$$x_1 \in [l''_1..h''_1], \ldots, x_n \in [l''_n..h''_n]$$

where for $j \in POS$

$$l''_j := max(l_j, \lceil \gamma_j \rceil), \; h''_j := h_j$$

with

$$\gamma_j := \frac{b \perp \sum_{i \in POS-\{j\}} a_i h_i + \sum_{i \in NEG} a_i l_i}{a_j}$$

and for $j \in NEG$

$$l''_j := l_j, \; h''_j := min(h_j, \lfloor \delta_j \rfloor)$$

with

$$\delta_j := \frac{\perp b + \sum_{i \in POS} a_i h_i \perp \sum_{i \in NEG-\{j\}} a_i l_i}{a_j}$$

This yields the following reduction rule:

LINEAR EQUALITY

$$\frac{\langle \sum_{i \in POS} a_i x_i \perp \sum_{i \in NEG} a_i x_i = b \; ; \; x_1 \in [l_1..h_1], \ldots, x_n \in [l_n..h_n] \rangle}{\langle \sum_{i \in POS} a_i x_i \perp \sum_{i \in NEG} a_i x_i = b \; ; \; x_1 \in [l'_1..h'_1], \ldots, x_n \in [l'_n..h'_n] \rangle}$$

where for $j \in POS$

$$l'_j := max(l_j, \lceil \gamma_j \rceil),\ h'_j := min(h_j, \lfloor \alpha_j \rfloor)$$

and for $j \in NEG$

$$l'_j := max(l_j, \lceil \beta_j \rceil),\ h'_j := min(h_j, \lfloor \delta_j \rfloor).$$

As an example of the use of the above domain reduction rule consider the CSP

$$\langle 3x \perp 5y = 4\ ;\ x \in [0..9], y \in [1..8]\rangle.$$

A straightforward calculation shows that $x \in [3..9]$, $y \in [1..4]$ are the ranges in the conclusion of *LINEAR EQUALITY* rule. Another application of the rule yields the ranges $x \in [3..8]$ and $y \in [1..4]$ upon which the process stabilizes. That is, the CSP $\langle 3x \perp 5y = 4\ ;\ x \in [3..8], y \in [1..4]\rangle$ is closed under the applications of the *LINEAR EQUALITY* rule.

Note that if in (6.3) there is only one variable, the *LINEAR EQUALITY* rule reduces to the following solving rule:

$$\frac{\langle ax = b\ ;\ x \in [l..h]\rangle}{\langle\ ;\ x \in \{\frac{b}{a}\} \cap [l..h]\rangle}$$

So, if a divides b and $l \leq \frac{b}{a} \leq h$, the domain expression $x = \frac{b}{a}$ is inferred, and otherwise a failure is reached. For instance, we have

$$\frac{\langle 4x = 19\ ;\ x \in [0..100]\rangle}{\langle\ ;\ x \in \emptyset\rangle}$$

6.2.3 Rules for Disequality Constraints

The reduction rules for simple disequalities are very natural. First, note that the following rule

SIMPLE DISEQUALITY 1

$$\frac{\langle x \neq y\ ;\ x \in [a..b], y \in [c..d]\rangle}{\langle\ ;\ x \in [a..b], y \in [c..d]\rangle}$$

where $b < c$ or $d < a$, is an instance of the *DISEQUALITY 2* solving rule introduced in Subsection 5.1.2 and where following the convention there mentioned we dropped the constraint from the conclusion of the proof rule.

Next, we adopt the following two solving rules that are instances of the *DISEQUALITY 3* solving rule:

SIMPLE DISEQUALITY 2

$$\frac{\langle x \neq y\ ;\ x \in [a..b], y = a\rangle}{\langle\ ;\ x \in [a+1..b], y = a\rangle}$$

SIMPLE DISEQUALITY 3

$$\frac{\langle x \neq y \ ; \ x \in [a..b], y = b\rangle}{\langle \ ; \ x \in [a..b \perp 1], y = b\rangle}$$

and similarly with $x \neq y$ replaced by $y \neq x$. Recall that the domain expression $y = a$ is a shorthand for $y \in [a..a]$.

To deal with disequality constraints that are not simple ones we use the following notation. Given a linear expression s and a sequence of ranges involving all the variables of s we denote by s^- the minimum s can take w.r.t. these ranges and by s^+ the maximum s can take w.r.t. these ranges. The considerations of Subsection 6.2.1 show how s^- and s^+ can be computed.

We now introduce the following transformation rule for non-simple disequality constraints:

DISEQUALITY 3

$$\frac{\langle s \neq t \ ; \ \mathcal{DE}\rangle}{\langle x \neq t, \ x = s \ ; \ x \in [s^-..s^+], \mathcal{DE}\rangle}$$

where

- s is not a variable,
- x is a fresh variable,
- $\mathcal{DE}$ is a sequence of the ranges involving the variables present in s and t,
- s^- and s^+ are computed w.r.t. the ranges in $\mathcal{DE}$.

An analogous rule is introduced for the inequality $s \neq t$, where t is not a variable.

6.2.4 Rules for Strict Inequality Constraints

Finally, to deal with the strict inequality constraints it suffices to use transformation rules that rewrite $s < t$ into $s \leq t$, $s \neq t$, and similarly with $s > t$.

6.2.5 Shifting from Intervals to Finite Domains

In our presentation we took care that all the rules preserved the property that the domains are intervals. In some constraint programming systems, such as ECL^{i}PS^e, this property is relaxed and instead of finite intervals finite sets of integers are chosen. To model the use of such finite domains it suffices to modify some of the rules introduced above.

In the case of inequality constraints we can use the following minor modification of the *LINEAR INEQUALITY 1* reduction rule:

LINEAR INEQUALITY 2

$$\frac{\langle \sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \leq b \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle}{\langle \sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \leq b \ ; \ x_1 \in [l'_1..h'_1] \cap D_1, \ldots, x_n \in [l'_n..h'_n] \cap D_n \rangle}$$

where l'_j and h'_j are defined as in the *LINEAR INEQUALITY 1* reduction rule with $l_j := min(D_j)$ and $h_j := max(D_j)$.

Note that in this rule the domains are now arbitrary finite sets of integers. An analogous modification can be introduced for the case of the reduction rule for equality constraints.

In the case of a simple disequality constraint we use the *DISEQUALITY 3* solving rule of Subsection 5.1.2. So now, in contrast to the case of interval domains, an arbitrary element can be removed from a domain, not only the "boundary" one.

This concludes our presentation of the proof rules that can be used to build a constraint solver for linear constraints over interval and finite domains. Such proof rules are present in one form or another within each constraint programming system that supports linear constraints over interval and finite domains.

6.2.6 Example: the *SEND + MORE = MONEY* Puzzle

To illustrate use of the above rules for linear constraints over interval and finite domains we analyze in detail the first formulation as a CSP of the *SEND + MORE = MONEY* puzzle introduced in Example 2.1 of Chapter 2. Recall that in this formulation we have the equality constraint

$$\begin{array}{rl} & 1000 \cdot S + 100 \cdot E + 10 \cdot N + D \\ + & 1000 \cdot M + 100 \cdot O + 10 \cdot R + E \\ = & 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y \end{array}$$

together with 28 simple disequality constraints $x \neq y$ for $x, y \in \{S, E, N, D, M, O, R, Y\}$ where x preceeds y in the alphabetic order, and with the range $[1..9]$ for S and M and the range $[0..9]$ for the other variables.

In the ECL^{i}PS^e system the above CSP is internally reduced to the one with the following domain expressions:

$$S = 9, E \in [4..7], N \in [5..8], D \in [2..8], M = 1, O = 0, R \in [2..8], Y \in [2..8]. \quad (6.4)$$

We now show how this outcome can be formally derived using the rules we introduced.

First, using the transformation rules for linear constraints we can transform the above equality to

$$9000 \cdot M + 900 \cdot O + 90 \cdot N + Y - (91 \cdot E + D + 1000 \cdot S + 10 \cdot R) = 0.$$

Applying the *LINEAR EQUALITY* reduction rule with the initial ranges we obtain the following sequence of new ranges:

$$S = 9, E \in [0..9], N \in [0..9], D \in [0..9], M = 1, O \in [0..1], R \in [0..9], Y \in [0..9].$$

At this stage a subsequent use of the same rule yields no new outcome. However, by virtue of the fact that $M = 1$ we can now apply the *SIMPLE DISEQUALITY 3* solving rule to $M \neq O$ to conclude that $O = 0$. Using now the facts that $M = 1, O = 0, S = 9$, the solving rules *SIMPLE DISEQUALITY 2* and *3* can be repeatedly applied to shrink the ranges of the other variables. This yields the following new sequence of ranges:

$$S = 9, E \in [2..8], N \in [2..8], D \in [2..8], M = 1, O = 0, R \in [2..8], Y \in [2..8].$$

Now five successive iterations of the *LINEAR EQUALITY* rule yield the following sequences of shrinking ranges of E and N with other ranges unchanged:

$$E \in [2..7], N \in [3..8],$$

$$E \in [3..7], N \in [3..8],$$

$$E \in [3..7], N \in [4..8],$$

$$E \in [4..7], N \in [4..8],$$

$$E \in [4..7], N \in [5..8],$$

upon which the reduction process stabilizes. At this stage the solving rules for disequalities are not applicable either. In other words, the CSP with the original constraints and the ranges (6.4) is closed under the applications of the *LINEAR EQUALITY* rule and the *SIMPLE DISEQUALITY* rules *1*, *2* and *3*.

So using these reduction rules we reduced the original ranges to (6.4). The derivation, without counting the initial applications of the transformation rules, consists of 24 steps.

Using the *SUBSTITUTION* rule of Subsection 5.1.2 and obvious transformation rules that deal with bringing the equality constraints to the form (6.3), the original equality constraint gets reduced to

$$90 \cdot N + Y \perp (91 \cdot E + D + 10 \cdot R) = 0.$$

Moreover, ten simple disequality constraints between the variables E, N, D, R and Y are still present.

6.2.7 A Characterization of the *LINEAR EQUALITY* Rule

It is useful to realize that the rules introduced in this section are quite limited in their strength. For example, using them we cannot even solve the CSP

$$\langle x + y = 10, x \perp y = 0 \ ; \ x \in [0..10], y \in [0..10]\rangle$$

has no application of the *LINEAR EQUALITY* rule of Subsection 6.2.2 to this CSP is relevant.

The point is that the domain reduction rules like the *LINEAR EQUALITY* rule are in some sense "orthogonal" to the rules that deal with algebraic manipulations (that can be expressed as transformation rules): their aim is to reduce the domains and not to transform constraints. So it makes sense to clarify what these rules actually achieve. This is the aim of this section. By means of example, we concentrate here on the *LINEAR EQUALITY* rule.

To analyze it it will be useful to consider first its counterpart that deals with the intervals of reals. This rule is obtained from the *LINEAR EQUALITY* rule by deleting from the definitions of l'_j and h'_j the occurrences of the functions $\lceil\ \rceil$ and $\lfloor\ \rfloor$. So this rule deals with an equality constraint in the form

$$\sum_{i \in POS} a_i x_i \perp \sum_{i \in NEG} a_i x_i = b \tag{6.5}$$

where

- a_i is a positive integer for $i \in POS \cup NEG$,
- x_i and x_j are different variables for $i \neq j$ and $i, j \in POS \cup NEG$,
- b is an integer,

and intervals of reals.

In what follows for two reals r_1 and r_2 we denote by $[r_1, r_2]$ the closed interval of real line bounded by r_1 and r_2. So $\pi \in [3, 4]$ while $[3..4] = \{3, 4\}$.

The rule in question has the following form:

$\mathcal{R}$-LINEAR EQUALITY

$$\frac{\langle \sum_{i \in POS} a_i x_i \perp \sum_{i \in NEG} a_i x_i = b \ ;\ x_1 \in [l_1, h_1], \ldots, x_n \in [l_n, h_n] \rangle}{\langle \sum_{i \in POS} a_i x_i \perp \sum_{i \in NEG} a_i x_i = b \ ;\ x_1 \in [l^r_1, h^r_1], \ldots, x_n \in [l^r_n, h^r_n] \rangle}$$

where for $j \in POS$

$$l^r_j := max(l_j, \gamma_j),\ h^r_j := min(h_j, \alpha_j)$$

and for $j \in NEG$

$$l^r_j := max(l_j, \beta_j),\ h^r_j := min(h_j, \delta_j).$$

Recall that $\alpha_j, \beta_j, \gamma_j$ and δ_j are defined in Subsection 6.2.2. For the purpose of the subsequent calculations recall that

$$\alpha_j = \frac{b \perp \sum_{i \in POS-\{j\}} a_i l_i + \sum_{i \in NEG} a_i h_i}{a_j}$$

and

$$\gamma_j = \frac{b \perp \sum_{i \in POS-\{j\}} a_i h_i + \sum_{i \in NEG} a_i l_i}{a_j}$$

It is straightforward to see that the *$\mathcal{R}$-LINEAR EQUALITY* rule, when interpreted over the intervals of reals, is also equivalence preserving. We need now the following lemma.

Note 6.2 (Convexity) *Consider an equality constraint of the form*

$$\sum_{i=1}^{n} a_i x_i = b$$

where $a_1, \ldots, a_n, b$ are integers. If $d = (d_1, \ldots, d_n)$ and $e = (e_1, \ldots, e_n)$ are its solutions over reals, then for any $\alpha \in [0, 1]$

- *$f := d + \alpha(e \perp d)$ is also a solution,*
- *for $i \in [1..n]$ we have*

$$f_i \in [d_i, e_i] \text{ if } d_i \leq e_i,$$

$$f_i \in [e_i, d_i] \text{ if } e_i \leq d_i,$$

where $f = (f_1, \ldots, f_n)$.

Proof. Straightforward and left as Exercise 14. □

Intuitively, the above note states that given two points in the n dimensional space $\mathcal{R}^n$ that represent solutions to an equality constraint, any point lying on the interval connecting these two points also represents a solution. Moreover, each such point lies inside the parallelogram determined by these two points.

To characterize the *$\mathcal{R}$-LINEAR EQUALITY* rule we use the notion of hyper-arc consistency. More precisely, we now prove the following result. By a *$\mathcal{R}$-LINEQ* CSP we mean here a CSP the domains of which are intervals of reals and the constraints of which are linear equality constraints.

Theorem 6.3 *($\mathcal{R}$-LINEAR EQUALITY) A non-failed $\mathcal{R}$-LINEQ CSP is hyper-arc consistent iff it is closed under the applications of the $\mathcal{R}$-LINEAR EQUALITY rule.*

That is, a non-failed $\mathcal{R}$-LINEQ CSP is hyper-arc consistent iff it is a final CSP in a derivation in which only the *$\mathcal{R}$-LINEAR EQUALITY* rule is applied.
Proof. ($\Rightarrow$) This implication is a direct consequence of the following general observation.

Claim Consider two CSP's $\mathcal{P}_1$ and $\mathcal{P}_2$ such that

- $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent,
- $\mathcal{P}_1$ has only one constraint,
- $\mathcal{P}_1$ is hyper-arc consistent.

Then the domains of $\mathcal{P}_1$ are respective subsets of the domains of $\mathcal{P}_2$.

Proof. Take an element a that belongs to a domain of $\mathcal{P}_1$. Since $\mathcal{P}_1$ has one constraint only and is hyper-arc consistent, a participates in a solution d to $\mathcal{P}_1$. Due to the equivalence of $\mathcal{P}_1$ and $\mathcal{P}_2$ d is also a solution to $\mathcal{P}_2$. Thus a that belongs to the corresponding domain in $\mathcal{P}_2$. $\square$

It suffices now to use the fact that the *$\mathcal{R}$-LINEAR EQUALITY* rule is equivalence preserving and that the domains of the CSP in the conclusion of this rule are respective subsets of the domains in the premise.

($\Leftarrow$) Choose a constraint of the considered CSP. It is of the form (6.5) with the corresponding domain expressions

$$x_1 \in [l_1, h_1], \ldots, x_n \in [l_n, h_n].$$

Then for $j \in [1..n]$ we have

$$l_j^r = l_j \text{ and } h_j^r = h_j. \tag{6.6}$$

Fix now some $j \in [1..n]$. Assume that $j \in POS$. The case when $j \in NEG$ is analogous and is omitted. Choose some $d \in [l_j, h_j]$. We now show that d participates in a solution to (6.5). By (6.6) we have $l_j^r \leq d \leq h_j^r$, so by the definition of l_j^r and h_j^r we have $\gamma_j \leq d \leq \alpha_j$. Thus for some $\alpha \in [0, 1]$, namely for

$$\alpha := \frac{d \perp \gamma_j}{\alpha_j \perp \gamma_j}$$

we have $d = \gamma_j + \alpha(\alpha_j \perp \gamma_j)$.

Next, note that we have both

$$\sum_{i \in POS - \{j\}} a_i l_i + a_j \alpha_j \perp \sum_{i \in NEG} a_i h_i = b$$

and

$$\sum_{i \in POS - \{j\}} a_i h_i + a_j \gamma_j \perp \sum_{i \in NEG} a_i l_i = b.$$

By assumption the considered CSP is not failed, so each of its domains is non-empty, that is $l_i \leq h_i$ for $i \in [1..n]$. Thus by the Convexity Note 6.2 a solution $(d_1, \ldots, d_n)$ to (6.5) exists such that $d = d_j$ and $d_i \in [l_i, h_i]$ for $i \in [1..n]$. $\square$

In contrast, the *LINEAR EQUALITY* rule behaves differently. Indeed, a CSP closed under the applications of this rule does not need to be hyper-arc consistent. Take for example the CSP

$$\langle 3x \perp 5y = 4 \ ; \ x \in [3..8], \ y \in [1..4] \rangle$$

discussed at the end of Subsection 6.2.2. It has precisely two solutions: $x = 3, y = 1$ and $x = 8, y = 4$, so it is not hyper-arc consistent.

So we need another notion to characterize CSP's closed under the applications of this rule. To this end we introduce the following definition. By a *LINEQ* CSP we mean here a CSP the domains of which are intervals of integers and the constraints of which are linear equality constraints.

Definition 6.4

- Consider a constraint C on a non-empty sequence of variables, the domains of which are intervals of reals or integers. We call C *interval consistent* if for every variable of it each of its two domain bounds participates in a solution to C.

- A CSP the domains of which are intervals of reals or integers is called *interval consistent* if every constraint of it is.

- Given a LINEQ CSP ϕ denote by ϕ^r the CSP obtained from ϕ by replacing each integer domain $[l..h]$ by the corresponding interval $[l, h]$ of reals and by interpreting all constraints over the reals. We say that ϕ is *bounds consistent* if ϕ^r is interval consistent. □

Observe that for a LINEQ CSP interval consistency implies bounds consistency. In turn, both forms of local consistency are weaker than hyper-arc consistency.

Also note that a LINEQ ϕ is bounds consistent if for every constraint C of it the following holds: for every variable of C each of its two domain bounds participates in a solution to ϕ^r.

The following examples clarify these notions.

Example 6.5
(i) The already mentioned CSP $\langle 3x \perp 5y = 4 \ ; \ x \in [3..8], y \in [1..4]\rangle$ is interval consistent as both $x = 3, y = 1$ and $x = 8, y = 4$ are solutions of $3x \perp 5y = 4$.

(ii) In contrast, the CSP

$$\phi := \langle 2x + 2y \perp 2z = 1 \ ; \ x \in [0..1], y \in [0..1], z \in [0..1]\rangle$$

is not interval consistent. Indeed, the equation $2x + 2y \perp 2z = 1$ has precisely three solutions in the unit cube formed by the real unit intervals for x, y and z: (0,0.5,0), (1,0,0.5) and (0.5,1,1). None of them is an integer solution. But each domain bound participates in a solution to $\phi^r := \langle 2x + 2y \perp 2z = 1 \ ; \ x \in [0, 1], y \in [0, 1], z \in [0, 1]\rangle$. So ϕ is bounds consistent. □

The following result now characterizes the *LINEAR EQUALITY* rule.

Theorem 6.6 (*LINEAR EQUALITY*) *A non-failed LINEQ CSP is bounds consistent iff it is closed under the applications of the LINEAR EQUALITY rule.*

That is, a non-failed LINEQ CSP is bounds consistent iff it is a final CSP in a derivation in which only the *LINEAR EQUALITY* rule is applied.
Proof. Let ϕ be a non-failed LINEQ CSP and ϕ^r the corresponding $\mathcal{R}$-LINEQ CSP. Clearly ϕ^r is non-failed either.

To prove the claim in view of the *$\mathcal{R}$-LINEAR EQUALITY* Theorem 6.3 it suffices to establish the following two equivalences:

1. ϕ is bounds consistent iff ϕ^r is hyper-arc consistent,

2. ϕ is closed under the applications of the *LINEAR EQUALITY* rule iff ϕ^r is closed under the applications of the $\mathcal{R}$-*LINEAR EQUALITY* rule.

Proof of 1.
($\Rightarrow$) By a simple application of the Convexity Note 6.2 to ϕ^r.
($\Leftarrow$) Immediate by the definition of ϕ^r.

Proof of 2. We first clarify the relationship between the applications of the *LINEAR EQUALITY* rule to ϕ and the applications of the $\mathcal{R}$-*LINEAR EQUALITY* rule to ϕ^r.

Let ψ be the conclusion of an application of the *LINEAR EQUALITY* rule to ϕ and χ the conclusion of an application of the $\mathcal{R}$-*LINEAR EQUALITY* rule to ϕ^r. Then the following holds.

(i) The intervals of ψ^r are respectively smaller than the intervals of χ.

This is due to the roundings resulting from the use of the $\lceil\rceil$ and $\lfloor\rfloor$ functions in the definition of the *LINEAR EQUALITY* rule;

(ii) The intervals of χ are respectively smaller than the intervals of ϕ^r.

This is by the definition of the $\mathcal{R}$-*LINEAR EQUALITY* rule.

(iii) If the intervals of χ have integer bounds, then ψ^r and χ coincide.

This is due to the fact that the roundings resulting from the use of the $\lceil\rceil$ and $\lfloor\rfloor$ functions in the definition of the *LINEAR EQUALITY* rule have no effect when applied to integer values.

Now $\psi = \phi$ implies $\psi^r = \phi^r$, which in turn implies by (i) and (ii) $\chi = \phi^r$. So ($\Rightarrow$) is an immediate consequence of (i) and (ii) while ($\Leftarrow$) is an immediate consequence of (iii).
□

Example 6.5(ii) shows that bounds consistency cannot be replaced here by interval consistency. In other words, to characterize the *LINEAR EQUALITY* rule it is needed to resort to a study of solutions over reals.

6.3 Exercises

Exercise 7 Prove that each of the transformation rules of Subsection 6.1.1 is equivalence preserving w.r.t. to the set of the variables of the premise. □

Exercise 8 Prove that every derivation in the proof system that consists of the transformation rules of Subsection 6.1.1 is finite. □

Exercise 9 Prove that every derivation in the proof system *BOOL* is finite. □

Exercise 10 Consider the proof system *BOOL'* given in Table 6.3.

Note that the rules *AND 1–3* of *BOOL* are replaced by the rules *AND 1'* and *AND 2'* of *BOOL'* and the rules *OR 2–4* of *BOOL* are replaced by the rules *OR 2'* and *OR 3'* of *BOOL'*. Also, the rule *AND 6* of *BOOL* is split in *BOOL'* into two rules, *AND 3'* and *AND 6'* and analogously for the rules *OR 6* of *BOOL* and *OR 3'* and *OR 6'* of *BOOL'*.

EQU 1 $\perp$ *4*	as in the system *BOOL*
NOT 1 $\perp$ *4*	as in the system *BOOL*
AND 1'	$x \wedge y = z, x = 1 \to z = y$
AND 2'	$x \wedge y = z, y = 1 \to z = x$
AND 3'	$x \wedge y = z, z = 1 \to x = 1$
AND 4	as in the system *BOOL*
AND 5	as in the system *BOOL*
AND 6'	$x \wedge y = z, z = 1 \to y = 1$
OR 1	as in the system *BOOL*
OR 2'	$x \vee y = z, x = 0 \to z = y$
OR 3'	$x \vee y = z, y = 0 \to z = x$
OR 4'	$x \vee y = z, z = 0 \to x = 0$
OR 5	as in the system *BOOL*
OR 6'	$x \vee y = z, z = 0 \to y = 0$

Table 6.3: Proof system *BOOL'*

(i) Prove that if a non-failed Boolean CSP is closed under the applications of the rules of the proof system *BOOL'*, then it is hyper-arc consistent.

(ii) Show that the converse result does not hold.
Hint. Take the CSP $\langle x \wedge y = z \ ; \ x = 1, y \in \{0,1\}, z \in \{0,1\}\rangle$.

(iii) Call a Boolean CSP *limited* if none of the following four CSP's forms a subpart of it:

- $\langle x \wedge y = z \ ; \ x = 1, y \in \{0,1\}, z \in \{0,1\}\rangle$,
- $\langle x \wedge y = z \ ; \ x \in \{0,1\}, y = 1, z \in \{0,1\}\rangle$,
- $\langle x \vee y = z \ ; \ x = 0, y \in \{0,1\}, z \in \{0,1\}\rangle$,
- $\langle x \vee y = z \ ; \ x \in \{0,1\}, y = 0, z \in \{0,1\}\rangle$. □

(iv) Prove that if a non-failed Boolean CSP is limited and hyper-arc consistent, then it is closed under the applications of the rules of the proof system *BOOL'*. □

Exercise 11 Prove the counterpart of the *BOOL* Theorem 6.1 for the six proof rules given in Table 6.4 that deal with the exclusive disjunction. □

Exercise 12 Prove that every derivation in which only the *LINEAR EQUALITY* rule is applied is finite. □

Exercise 13

XOR1	$x \oplus y = z, x = 1 \rightarrow \neg y = z.$
XOR2	$x \oplus y = z, x = 0 \rightarrow y = z.$
XOR3	$x \oplus y = z, y = 1 \rightarrow \neg x = z.$
XOR4	$x \oplus y = z, y = 0 \rightarrow x = z.$
XOR5	$x \oplus y = z, z = 1 \rightarrow \neg x = y.$
XOR6	$x \oplus y = z, z = 0 \rightarrow x = y.$

Table 6.4: Different proof rules for *XOR*

(i) Prove that in the case of a single linear equality constraint the $\mathcal{R}$-*LINEAR EQUALITY* rule is idempotent, that is a CSP is closed under the application of this rule after one iteration.

(ii) Prove that in the case of more than one linear equality constraint the $\mathcal{R}$-*LINEAR EQUALITY* rule can yield an infinite derivation.
Hint. Take $\langle x = y, x = 2y \; ; \; x \in [0, 100], y \in [0, 100]\rangle$ and consider a derivation in which the constraints are selected in an alternating fashion.

Exercise 14 Prove the Convexity Note 6.2. □

6.4 Bibliographic Remarks

The proof system *BOOL'* from Exercise 10 is taken from Codognet & Diaz (1996). The proof system *BOOL* is a modification of it obtained in order to get the *BOOL* Theorem 6.1. The reduction rules in the style of the system *BOOL* are sometimes referred to under the name of *unit propagation*. The inference problem discussed in Subsection 6.1.3 is from Frühwirth (1995).

The reduction rules *LINEAR INEQUALITY 1* and *LINEAR EQUALITY* are simple modifications of the reduction rule introduced in Davis (1987, page 306) that dealt with closed intervals of reals. These rules are well-known and were implemented in various constraint programming systems, for example CHIP. They are instances of the so-called cutting-plane proof technique used in linear programming (see e.g. Cook, Cunningham, Pulleyblank & Schrijver (1998, Section 6.7)). The use of the *SEND + MORE = MONEY* puzzle to discuss the effect of these rules is from van Hentenryck (1989, page 143).

The presentation in Section 6.2 is based on Apt (1998). The $\mathcal{R}$-*LINEAR EQUALITY* Theorem 6.3 generalizes a corresponding result stated in Davis (1987, page 326) for the more limited case of so-called unit coefficient constraints. Exercise 13 is taken from Davis (1987, page 304).

The notion of bounds consistency was introduced in a number of papers, for example Hentenryck, Saraswat & Deville (1998). The terminology here adopted follows Marriott & Stuckey (1998). A similar notion was introduced in Lhomme (1993) for the case of constraints on reals.

References

Apt, K. R. (1998), 'A proof theoretic view of constraint programming', *Fundamenta Informaticae* **33**(3), 263–293. Available via `http://www.cwi.nl/~apt`.

Codognet, P. & Diaz, D. (1996), 'A simple and efficient Boolean constraint solver for constraint logic programming', *Journal of Automated Reasoning* **17**(1), 97–128.

Cook, W., Cunningham, W., Pulleyblank, W. & Schrijver, A. (1998), *Combinatorial Optimization*, John Wiley & Sons, Inc., New York.

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

Frühwirth, T. (1995), Constraint Handling Rules, *in* A. Podelski, ed., 'Constraint Programming: Basics and Trends', LNCS 910, Springer-Verlag, pp. 90–107. (Châtillon-sur-Seine Spring School, France, May 1994).

Hentenryck, P. V., Saraswat, V. & Deville, Y. (1998), 'Design, implementation, and evaluation of the constraint language cc(fd)', *Journal of Logic Programming* **37**(1–3), 139–164. Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriot, Eds.).

Lhomme, O. (1993), Consistency techniques for numeric CSPs, *in* 'Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-93)', pp. 232–238.

Marriott, K. & Stuckey, P. (1998), *Programming with Constraints*, The MIT Press, Cambridge, Massachusetts.

van Hentenryck, P. (1989), *Constraint Satisfaction in Logic Programming*, Logic Programming Series, The MIT Press, Cambridge, MA.