Chapter 2

Constraints

2.1 Introduction

2.1.1 Motivations

Constraint programming originated from the logic programming community and has become a flourishing programming paradigm, which is implemented in a number of heterogeneous environments, like: ECL

PS

(e see [Agg95]), CONSTRAINT HANDLING RULES (see [Fru98]), Oz (see [Smo95]), CHIP (see [ADH+87]). As stated in [MS98], constraint programming, "based on a strong theoretical foundation, [...] is now becoming the method of choice for modelling many types of optimization problems, in particular, those involving heterogeneous constraints and combinatorial search"; thus "constraint programming has recently been identified by the ACM [Association for Computing Machinery] as one of the strategic directions in computing research".

The central notion of constraint programming is that of a so-called constraint satisfaction problem, which in a nutshell consists of a finite collection of constraint relations over a finite number of domains.

The only task left to the constraint programmer is to formulate a given problem as a constraint satisfaction problem. Then the problem is "solved" by the constraint programming system, by means of general or domain specific methods. Here "solving" can mean finding values, one from each domain, that "satisfy" the problem constraints, or the optimal values, with respect to some criteria, for the problem constraints. The latter task has grown into an independent area, and we shall introduce and study it separately in Chapter 5.

Problems that can be expressed in a natural way by means of constraints are, for instance, those that lack efficient solving algorithms; like the MAP COLOURABILITY PROBLEM or the 3-SAT problem and, in general, combinatorial problems that are computationally intractable. Yet constraint programming has grown further than this and it can be traced in areas like Temporal Reasoning, Scheduling,
Planning and Spatial Reasoning, see [DAA95].

For most of the aforementioned problems, the search for a solution usually involves a number of fruitless computations before any solution can be generated. Nowadays, constraint programming incorporates a variety of methods that are conceived for avoiding some fruitless explorations of the solution search space. In Chapter 4, we survey a number of these methods.

Moreover, the relational aspect of constraints often allows the programmer to re-use the same program for different purposes. In contrast, traditional programming languages do not usually provide support for specifying relations among the various components of programs. It is then the programmer who bears the burden to specify and maintain those relations in a dynamic situation.

The use of relations for programming is also popular in the database community, see the database relational model as for instance in [AHV95, Ull80, UW97]. Indeed, there are similarities between the two fields, as outlined in Chapter 6. However, in a relational database, relations are usually extensionally characterised, namely as tables; the task here is to efficiently query the database and retrieve all solutions/answers to the query. On the other hand, constraints can be assigned intensionally, like linear equations over real numbers are; the task is to satisfy all the problem constraints, and the solving algorithms are designed to cleverly produce a solution to the problem.

2.1.2 Outline and Structure

In this chapter we introduce the so-called constraint satisfaction problems, and the basic notions necessary for the comprehension of the remainder of Part I.

Section 2.2 presents the core notions of constraints, constraint satisfaction problems and their solutions; the terminology that we adopt usually follows the standard one; whenever we introduce some new conventions or terms, we signal these and explain their use.

A number of motivating examples are proposed in Section 2.3: these range from combinatorial problems (see Subsection 2.3.1) to problems that arise in areas like Image Understanding (see Subsection 2.3.3), Spatial and Temporal Reasoning, Planning (see Subsection 2.3.4). Some of those examples return in the remainder of this thesis: precisely when we explain the algorithms of Chapter 4, and introduce non-classical constraints in Chapter 5.

The algorithms that occur in the remainder of this thesis (Chapter 4, Sections 5.4 and 5.5) are explained via transformations of problems, as presented in Section 2.4, and via basic operations and orderings on problems, as illustrated in Section 2.5.
2.2 Constraint Problems and Their Solutions

In a constraint programming environment, problems are cast in terms of variables, domains and constraints: each variable is associated with a domain of interpretation, from which it takes its possible values; constraints on variables restrict the allowed domain values for variables. We define variables, domains, constraints and, finally, constraint satisfaction problems precisely in Subsection 2.2.1.

The notion of a solution to such problems is explained in Subsection 2.2.2. Intuitively, a solution to a problem assigns values, to each domain variable, according to the constraints imposed by the problem. For instance: we need to schedule a series of meetings on a certain date, so we open our agenda on that date and start filling in columns, corresponding to different day hours; these are the variables of our problem, and the events that can take place in those hours are the variable values. A constraint for this problem could be that there are no meetings that take place at the same hour.

In conclusion, the choice of variables and domains defines the search space for solutions; in the above case, the set of meetings that we want to schedule. Constraints characterise the structure of the solution search space; e.g., the fact that there cannot be overlapping meetings.

2.2.1 What Constraint Satisfaction Problems Are

Variables

To define a constraint satisfaction problem, we need a finite sequence of \( n \) distinct variables, say \( r := (x_1, \ldots, x_n) \). Consider a non-empty sequence \( s := (x_{i_1}, \ldots, x_{i_m}) \) of \( r \) variables such that, either \( i_j < i_{j+1} \) for each \( 1 \leq j < m \), or \( m = 1 \); then \( s \) is a scheme of \( r \) of length \( m \).

We shall usually denote the scheme of variables of a constraint satisfaction problem by \( X \); then, \( r, s, t \), or the same with subscripts or superscripts, will usually denote schemes of \( X \).

Besides, in this thesis, we shall write a scheme \( s := (x_1, \ldots, x_m) \) as \( s := x_{i_1}, \ldots, x_{i_m} \) to avoid an overload of notation. Notice that, if a scheme \( r \) contains \( n \) variables, then there are \( 2^n - 1 \) possible schemes of \( r \).

Example 2.2.1. The scheme \( s := x_1, x_2, x_3 \) gives rise to 7 schemes: i.e., \( s_1 := x_1, s_2 := x_2, s_3 := x_3, s_4 := x_1, x_2, s_5 := x_1, x_3, s_6 := x_2, x_3, s_7 := x_1, x_2, x_3 \) itself.

There are two useful operations on schemes: if \( s \) and \( t \) are two different schemes of \( r \), then \( s \cup t \) denotes the scheme of \( r \) on the variables in \( s \) and \( t \); we call this operation the join of \( s \) and \( t \). Vice versa, if \( t \) is a scheme of \( s \), different from \( s \), then \( s - t \) is the scheme on the variables of \( s \) minus those in \( t \), and we shall refer to it as the projection of \( t \) out of \( s \).
**Example 2.2.2.** Let us consider \( r := x_1, x_2, x_3 \), and its schemes \( s := x_1, x_2, t := x_2, x_3 \). Then the join of \( s \) and \( t \) is \( r \) itself; the projection of \( t \) out of \( s \) is \( x_3 \).

**Domains**

Each variable \( x_i \) in a scheme \( r := x_1, \ldots, x_n \) is interpreted over a domain, usually denoted with \( D_i \). The Cartesian product of all variable domains

\[
D := D_1 \times \cdots \times D_n
\]

is called the *domain* of \( r \), whereas the set of pairs \( \langle D_i, x_i \rangle \) is denoted by

\[
D := \{ \langle D_1, x_1 \rangle, \ldots, \langle D_n, x_n \rangle \}.
\]  

(DS)

The set \( D \) in (DS) is referred to as the *domain set* of \( r \). We shall usually adopt a more compact notation and write a domain set in the form

\[
D = D_1, \ldots, D_n
\]

every time meaning (DS) as above.

Given a scheme \( s := x_{i_1}, \ldots, x_{i_m} \) of \( r \), we denote the Cartesian product

\[
D_{i_1} \times \cdots \times D_{i_m}
\]

with \( D[s] \). Notice that, if \( s \) is a singleton as \( x_i \), then \( D_i = D[x_i] \). Similarly, if \( d_i \in D_i \) for every \( D_i \in D \), let \( d \) be the tuple

\[
(d_1, \ldots, d_n).
\]

Then \( d[s] \) denotes the tuple \((d_{i_1}, \ldots, d_{i_m})\).

In case \( D[s] \) and \( d[s] \) reduce to singletons, we shall blur the above distinctions, that is we shall feel free to write \( D_i \) instead of \( D[x_i] \), as well as \( d_i \) in place of \( d[x_i] \). Also, if \( s \) is the singleton of \( x_i \), then the tuple \( d[s] \) corresponds to \( d_i \).

In a number of cases that we investigate in the present thesis, domains are finite. However, there are situations in which domains are allowed to be infinite, as in the case of linear inequalities over real numbers; see the following example.

**Example 2.2.3.** Given a scheme \( r := x_1, x_2, x_3 \), let the variables of \( r \) range over real numbers. Precisely, let \( D_1 \) be the closed interval \([0, 1]\), whereas \( D_2 \) and \( D_3 \) are equal to \([1, 3]\). Then \( D \) is the Cartesian product \( D_1 \times D_2 \times D_3 \). While \( D \) is the following set:

\[
\{ ([0,1], x_1), ([1,3], x_2), ([1,3], x_3) \}.
\]

Notice that, being \( D_2 \) equal to \( D_3 \), if we cast \( D \) as a set of domains, we would lose the association between variables and their domains.
2.2. Constraint Problems and Their Solutions

Constraints

In this thesis, constraints are usually described as relations; so each constraint is associated to a scheme of variables, like in the database relational model.

Formally, given a scheme $r$ and a domain $D$ over $r$, let $s := \{x_{i_1}, \ldots, x_{i_m}\}$ be a scheme of $r$. Then a constraint over $s$, written as $C(s)$, is a subset of the Cartesian product $D[s]$. If $k$ is the length of $s$, then $C(s)$ is a $k$-ary constraint.

The above definition of constraint has the advantage of being easily generalised to non-standard constraints, as presented in Chapter 5. Moreover, it naturally captures the intuitive meaning of constraints: i.e., that of relating variable values and thereby restricting possible assignments, as we shall make precise in Subsection 2.2.2.

In the literature, constraints as relations can be represented extensionally, for instance as tables, or intensionally. The following is an example of the latter representation.

**Example 2.2.4.** Consider the following system of equations over real numbers:

\[
\begin{align*}
2x_1 + x_2 + 2 &= 3x_4 - 1 \\
x_2 &= 3x_1 \\
x_1 + x_2 &= 7x_3 \\
x_4 &= x_3 + x_2 + 1
\end{align*}
\]

Variables are the unknown $x_i$, for $i = 1, \ldots, 4$. Domains are equal, for instance, to $\mathbb{R}$. Each equality in the above system is regarded as a constraint on the related variables; for example, the first equality is interpreted as a constraint $C(x_1, x_2, x_4)$ on the scheme $x_1, x_2, x_4$.

In the case of Example 2.2.4, an extensional representation of constraints would be impossible: it would have to list all the allowed triples of real numbers.

We conclude this part with some definitions. Consider $k$ different constraints on $k$ schemes of $r$, say $C_i(s_i)$ for $i = 1, \ldots, k$. Then the set of pairs

\[
C := \{(C(s_1), s_1), \ldots, (C(s_k), s_k)\} \tag{CS}
\]

is referred to as a constraint set of $r$. If the number of involved constraints is of relevance, then we call the above set a $k$-constraint set. As in the case of domain sets, we shall usually adopt a more compact notation and denote constraint sets as (CS) in the following way:

\[
C := C(s_1), \ldots, C(s_k).
\]

We have now all the ingredients, namely variables, domain and constraint sets, to formalise the notion of constraint satisfaction problem as below.
Constraint Satisfaction Problems

A *constraint satisfaction problem*, briefly CSP, is a tuple \( P := \langle X, D, C \rangle \) defined as follows:

1. \( X \) is a scheme;
2. \( D \) is a domain set of \( X \);
3. \( C \) is a constraint set of \( X \).

Whenever we need to be more specific and highlight the scheme \( X \) of a CSP, we shall talk of a *CSP over \( X \)*. So, by writing the *CSPs over \( X \)*, we refer to all the CSPs that have the same scheme \( X \).

### 2.2.2 Global Satisfaction

**Assignments**

Given a scheme \( X \) of \( n \) variables and \( D := D_1 \times \cdots \times D_n \), a domain over \( X \), a tuple \( d \in D \) is a *total assignment* or *total instantiation*. The name is motivated by the fact that each tuple \( d \in D \) gives rise to a function that assigns a single value in \( D_i \), namely \( d[i] \), to each variable \( x_i \in X \), thereby instantiating it; and vice versa as well. We shall blur this distinction and consider assignments as functions whenever convenient.

Given a scheme \( s \) of \( X \), a total assignment for \( s \), namely a tuple of \( D \[s\] \), is an *\( s \) assignment* over \( D \). If the scheme \( s \) of \( X \) is not relevant, we generically talk of *assignment*.

Indeed, if \( r \) is a scheme of \( s \), every \( s \) assignment \( d \) gives rise to an \( r \) assignment, namely \( d \[r\] \). In the literature, this is usually referred to as *assignment restriction*, since assignments are usually defined as functions. Vice versa, every \( r \) assignment \( e \) can be extended to an \( s \) assignment, possibly more than one, in an arbitrary way. So, if \( e \in D \[r\] \), then \( d \in D \[s\] \) is an *\( r \) extension* of \( e \) to \( s \) iff \( d \[r\] = e \).

**Example 2.2.5.** The system of equations in Example 2.2.4 gives rise to a CSP. An example of a total assignment is the tuple \( d := (0, 1, 2, 3) \). If \( s \) is the scheme \( x_1, x_2, x_3 \), then \( d \[s\] \) is the \( s \) assignment \((0, 1, 2)\).

**Consistency and solutions**

Let us consider a CSP \( P := \langle X, D, C \rangle \), and let \( C(s) \) be a constraint on a scheme \( s \) of \( P \). Suppose that the scheme \( t \) extends \( s \). If a \( t \) assignment \( d \) over \( D \) is such that \( d \[s\] \) belongs to \( C(s) \), then we say that \( d \) satisfies or is *consistent with* the constraint \( C(s) \).

Informally, a solution to the CSP \( P := \langle X, D, C \rangle \) is a total assignment of which each projection, over a scheme \( s \) of \( X \), satisfies every constraint of \( P \) over
2.3. Examples

s. Formally, consider a total assignment $d$ over $D$; if, for every $C(s)$ over $s$ of $P$, the tuple $d[s]$ satisfies $C(s)$, then we say that $d$ satisfies or is consistent with $P$. The assignment $d$ is commonly referred to as a solution to $P$.

Thus a CSP $P$ is consistent or satisfiable iff there exists a total assignment that satisfies it. The subset $Sol(P)$ of the domain set $D$ denotes the set of solutions to $P$.

**Example 2.2.6.** Let us consider the systems of equations in Example 2.2.4, that gives rise to a CSP. The total assignment $d := (0,0,0,0)$ is not a solution because of the fourth equation. If $s$ is the scheme $x_1, x_2, x_3$, then $d[s]$ is an $s$ assignment; this can be extended to an assignment that is a solution to the system, namely $e := (0,0,0,1)$. In fact, it is not difficult to check that the assignment $e$ satisfies all equations/constraints of the problem.

2.3 Examples

We started this chapter by claiming that CSPs are ubiquitous. In the remainder of this section, we present some examples to support our claim.

2.3.1 Map Colourability

The first example we discuss is combinatorial. A planar map, like the one sketched below, can be represented by means of a graph $G := (V, E)$ and a finite set $D_i$ of colours, one for each vertex $x_i$ in $V$. Hence the MAP COLOURABILITY PROBLEM (see [GJ79]) consists in colouring the graph vertices so that no two adjacent vertices are painted with the same colour.

The instance of the MAP COLOURABILITY PROBLEM which corresponds to the above map is the following:

1. the set of vertices is $V := \{x_1, x_2, x_3\}$;
2. the set of colours are, respectively: $D_1 := \{\text{aqua, blue, cyan}\}$ for $x_1$; then $D_1 := \{\text{cyan}\}$ for $x_2$; finally $D_3 := \{\text{blue, cyan}\}$ for $x_3$;
3. the only arcs in the graph are $(x_1, x_2), (x_2, x_3), (x_1, x_3)$. 

\[
\begin{array}{|c|c|}
\hline
x_1 & x_2 \\
\hline
\{\text{aqua, cyan, blue}\} & \{\text{cyan}\} \\
\hline
x_3 \mapsto \{\text{cyan, blue}\} \\
\hline
\end{array}
\]
Chapter 2. Constraints

The encoding of this as a CSP is straightforward:

1. variables correspond to the vertices in \( V \);
2. each colour set corresponds to a variable domain;
3. constraints are posted between those variables that are connected by an arc in the graph: i.e., \( C(x_i, x_j) \) states that \( x_i \) and \( x_j \) have different colours, for \( 1 \leq i < j \leq 3 \).

A solution to the MAP COLOURABILITY PROBLEM is an assignment of colours to all the variables that satisfy all the given constraints. In the case of the depicted map above, a solution is as follows: \( x_1 \) is \textit{aqua}, \( x_2 \) is \textit{cyan} and \( x_3 \) is \textit{blue}.

2.3.2 Satisfiability Problems

In Chapter 9, we shall also deal with \textit{propositional satisfiability}. A conjunction \( \phi \) of propositional disjunctions of the form

\[ p \lor \neg q \lor r \]  

(2.1)

can be encoded as a CSP as follows (other encodings have been devised in the literature, see for instance [Wal00]):

- variables are proposition letters, such as \( p, q \) and \( r \);
- domains only contain the Boolean values 0 (false) and 1 (true);
- constraints are posted between those variables/letters that occur in the same disjunct. For instance, (2.1) corresponds to a constraint on the variables \( p, q \) and \( r \), that only rules out the tuple \((0, 1, 0)\) from the set \( \{0, 1\}^3 \).

A solution is thus an assignment, to all the variables/letters in the formula \( \phi \), that satisfies \( \phi \).

2.3.3 Image Understanding

Computer vision is an important AI area, that arose as part of robotics. Nowadays, its applications have moved beyond robotics; for instance, we encounter computer vision methods in the interpretation of satellite data.

Computer vision involves image analysis and understanding. A prototypical problem in this sense (see [DAA95]) is the \textit{scene labelling problem}. The task is to reconstruct objects in a three dimensional scene by means of their bidimensional representations. This problem was first encoded as a CSP by Waltz; see [Wal75] as quoted in [DAA95]. The original problem is transformed from one of labelling lines to one of labelling junctions between lines. Waltz's procedure relies on two physical constraints to make the problem tractable:
2.3. Examples

- a number of combinations of line labellings at a junction are not physically realisable;

- each line connects two junctions; thereby the labellings at the two junctions must both assign the same label to the line.

Thus the goal is to find a physically consistent set of labellings for junctions. The procedure by Waltz dramatically reduces the search space size by means of the algorithms that will be presented in Section 4.2. We invite the reader to consult [DAA95] for a more detailed account of this.

Here we focus on another typical problem of image understanding, successfully tackled as a CSP in [Aie02]: document understanding. In Aiello's thesis (ib.), given a set of labellings that identify the basic components of a document, a CSP solver is used to reconstruct a reading order for the document. For instance, the labelled layout of Figure 2.1 is encoded as the following CSP:

1. each rectangular box in the drawing is associated with one variable;

2. variable domains collect pairs of real numbers, interpreting the upper-left corner and the lower-right corner of a rectangular box;

3. constraints are expressed through the bidimensional Allen relations. For instance, in Figure 2.1, $x_1$ and $x_2$ are related as follows: $x_1$ equals $x_2$ states that the projections of the two documents on the $Y$ axis coincide; whereas $x_1$ precedes $x_2$ states that the horizontal component of $x_1$ precedes that of $x_2$.

A solution to the above CSP is a reading order that satisfies all the Allen constraints of the problem.

2.3.4 Temporal Reasoning

Another example we consider pertains to Qualitative Temporal Reasoning and Scheduling, see [Gen98]. Suppose to have a series of tasks, each one taking a continuous interval of time, and to be all accomplished the same day, as in the following simple plot.

Arie fidgets in his pocket, searching for his small agenda. Much to his surprise, he realises to have lost it. All he can vaguely remember is to have an important meeting that day in Amsterdam, in the meeting room B2.24. He knows that he should meet Bert a long time before Barbara starts her meeting with Kees in the B2.24 room. Arie remembers Cees talking about his own meeting with Dick in that room, and that this should be over by the time Dora meets Alfons there. Besides, the meeting of Barbara and Kees should be before the meeting of Dora and Alfons in the B2.24 room. Arie perfectly knows that
all these events take place sequentially in the B2.24 room, with an interval of at least ten minutes in between any two meetings. Arie also remembers that everybody has precisely one meeting that day. Are Arie’s memories consistent?

This story can be encoded as a CSP with four variables, one for each meeting in the story. So, let us state the following: $x_1$ means “Arie and Bert’s meeting”; $x_2$ is “Barbara and Kees’ meeting”; $x_3$ denotes “Cees and Dick’s meeting”; $x_4$ represents “Dora and Alfons’ meeting”. Each variable is interpreted over the real line. Reading the story, we encounter four constraints between those variables; the constraints are formalised through the Allen relation *precedes* and its inverse, *follows*:

1. $C(x_1, x_2)$ is $x_1 \text{ precedes } x_2$; this states that Arie and Bert’s meeting takes place before Barbara and Kees’ meeting;

2. $C(x_3, x_4)$ is $x_3 \text{ precedes } x_4$; this encodes the fact that Cees and Dick’s meeting takes place before Dora and Alfons’ meeting;

3. $C(x_2, x_4)$ is $x_2 \text{ precedes } x_4$; this translates the fact that Barbara and Kees’ meeting takes place before Dora and Alfons’ meeting;

4. $C(x_1, x_4)$ is $x_1 \text{ precedes } \lor \text{ follows } x_4$; this translates the fact that Arie and Bert’s meeting has to take place before or after Dora and Alfons’ meeting.

Then a solution is found when all the above constraints are satisfied.

A more challenging application for constraint programming, involving Temporal Reasoning, is provided by planning. We refer the reader to [KvB97] for a clear introduction to planning as CSP, and a comparison of the constraint programming planner CPLAN with other state-of-the-art planners.
2.4 Equivalent Problems

A number of CSP algorithms, like those that we present in Chapter 2, are better understood if we assume that the input problem has at most one constraint per scheme, or even precisely one per scheme. These two properties are obtained, respectively, by means of two procedures, named 'normalisation' and 'completion'. We describe those procedures in this subsection, and show that neither of them add or remove solutions with respect to the original problem; thus they preserve equivalence, which is precisely defined as follows — we remind that Sol(P) denotes the set of all solutions to P, see p. 13.

**Definition 2.4.1.** Consider two CSPs \( P := (X, D, C) \) and \( P' := (X, D', C') \) on the same scheme \( X \). Then \( P \) and \( P' \) are equivalent CSPs if they have the same solution set: i.e., \( \text{Sol}(P) = \text{Sol}(P') \).

**Example 2.4.2.** Consider the CSP in Example 2.2.4 and the CSP with the same scheme and domains, that has as constraints the following equations over real numbers:

\[
\begin{align*}
5x_1 + 2 &= 3x_4 - 1 \\
x_2 &= 3x_1 \\
4x_1 &= 7x_3 \\
x_4 &= x_3 + 3x_1 + 1
\end{align*}
\]

The former CSP and the latter are equivalent, since they have the same solutions over real numbers. In fact, the latter CSP is obtained from the former by replacing the occurrences of \( x_2 \) by \( 3x_1 \) in the first, third and fourth equations; this is an equivalence-preserving transformation, which is taught in high schools.

The above definition can be easily extended to compare CSPs on different schemes; yet, we shall not be in need of such extension in the present thesis.

2.4.1 Normalisation

If we transform a CSP so that it has at most one constraint on each scheme of variables, we obtain a normal form for it, as explained in the following definition.

**Definition 2.4.3.** Consider a CSP \( P := (X, D, C) \). The normalisation of \( P \) is a CSP \( P' := (X, D, C') \) that shares the variable scheme \( X \) and domain set \( D \) with \( P \). Then, for each scheme \( s \) of \( X \) such that there exist \( k \geq 1 \) constraints \( C_1(s), \ldots, C_k(s) \) on \( s \) in \( P \),

- there exists precisely one constraint \( C'(s) \) in \( C' \),
- and \( C'(s) \) is the constraint on \( s \) that is equal to \( \bigcap_{i=1}^{k} C_i(s) \).
The problem $P$ is normalised if it satisfies the above requirements. The above definition is consistent due to the following fact.

**Fact 2.4.4.** Every CSP has precisely one normalisation.

**Example 2.4.5.** The Temporal Reasoning problem in Subsection 2.3.4 is normalised, since there is at most one constraint on each scheme.

Notice that, in the above example, the original CSP has only binary constraints and so does its normalisation. Indeed, intersection does not modify the arity of constraints, hence the following fact.

**Fact 2.4.6.** A CSP has a $k$-ary constraint iff its normalisation does.

The above fact is trivial but not to overlook; in fact, some CSP algorithms only deal with constraints of a fixed arity, like binary ones. Thus the above fact ensures that normalisation does not modify the nature of a CSP, so to speak; passing to the normalisation is just to simplify the description of the algorithms.

Solving a CSP also means finding a total instantiation that is consistent with every constraint of the problem. Therefore, the following question is of primary concern: do we add or remove any solution by normalising a CSP? The answer is clearly negative, and its simple proof is outlined in the following lemma; it relies on the fact that an instantiation is consistent with a CSP if it satisfies all its constraints.

**Lemma 2.4.7.** A CSP $P$ and its normalisation are equivalent CSPs.

**Proof.** Let $d$ be a tuple in $D$ that is consistent with all the constraints $C(s)$ of $P$. Then consider a scheme $s$ of $X$ and analyse the following three cases.

1. If there is only one constraint $C(s)$ in $P$, then the same constraint on $s$ and no other is in the normalisation of $P$; hence $d$ is consistent with the constraint $C(s)$ of the normalisation of $P$.

2. If there is more than one constraint on $s$ in $P$, then $d[s]$ has to be consistent with all of them, thereby with their intersection as well.

3. Finally, if there are no constraints in $P$ on $s$, then there are no constraints in the normalisation of $P$ either.

The other implication follows by inspecting the same three cases and assuming that $d$ is consistent with all the constraints of the normalisation of $P$. 

\[\square\]
2.4. Equivalent Problems

2.4.2 Completions

Several constraint propagation algorithms can be better described by assuming a stronger working hypothesis than normalisation: i.e., that the input problem has precisely one constraint on each scheme of variables, so that the problem is complete in the following sense.

**Definition 2.4.8.** Consider a CSP P and its normalisation \( P^N := \langle X, D, C \rangle \). The *completion* of P is the problem \( \tilde{P} := \langle X, D, \tilde{C} \rangle \) whose constraint set \( \tilde{C} \) enjoys the following properties:

- for each scheme s of X, if \( C(s) \) belongs to C of \( P^N \), then it is also the only constraint on s in \( \tilde{C} \);
- if \( P^N \) has no constraints on s, then \( \tilde{C} \) has precisely one constraint \( \tilde{C}(s) \) on s, that is \( D[s] \).

We say that a CSP P is *complete* iff \( P = \tilde{P} \).

The completion of a CSP is obtained by normalising the problem, and adding the necessary constraints as in the above definition.

While normalisation does not alter the nature of a CSP (i.e. if it is binary its normalisation is binary too), its completion instead modifies it. For instance, if \( P \) is a binary CSP, the choice of \( \tilde{P} \) is by no means optimal: it may have too many constraints with respect to those in \( P \). Some CSP algorithms (see Subsection 4.4.2), require the input problem to be complete but only up to constraints of arity at most \( k \), so to speak. Hence, we refine the above definition as follows.

**Definition 2.4.9.** Consider a CSP \( P := \langle X, D, C \rangle \) on \( n > 0 \) variables, and let \( k \) be a natural number, not greater than \( n \) and different from 0.

- The CSP \( \tilde{P}_k \) is the *k completion* of P if the constraints of \( \tilde{P}_k \) are all the k-ary constraints of \( \tilde{P} \). The problem P is *k complete* iff \( P = \tilde{P}_k \).
- While \( \tilde{P}^s_k \) is the *strong k completion* of P iff the constraints of \( \tilde{P}^s_k \) are all the i-ary constraints of \( \tilde{P} \) for every \( 0 < i \leq k \). The problem P is *strongly k complete* iff \( P = \tilde{P}^s_k \).

**Example 2.4.10.** The Temporal Reasoning problem in Subsection 2.3.4 is 2 complete, since it has precisely one binary constraint on each scheme of \( x_1, x_2, x_3, x_4 \) of length 2.

The above definitions of completions are consistent due to the following fact.
FACT 2.4.11.
(i). Every CSP $P := \langle X, D, C \rangle$ on $n$ variables has precisely one $k$ completion and one strong $k$ completion, for every $k \leq n$.
(ii). The completion of $P := \langle X, D, C \rangle$ is the strong $n$ completion of $P$, for $n$ equal to the cardinality of $X$.

As in the case of normalisation, the completion of a CSP $P$ is equivalent to $P$. The proof of the following lemma is analogous to that of Lemma 2.4.7.

LEMMA 2.4.12.
(i). A CSP $P$ and its $k$ completion are equivalent problems, for every $k \geq 0$ that is not greater than the number of variables in $P$.
(ii). A CSP $P$ and its strong $k$ completion are equivalent problems, for every $k \geq 0$ that is not greater than the number of variables in $P$.
(iii). A CSP $P$ and its completion $\bar{P}$ are equivalent problems.

2.5 Combining and Comparing Problems

2.5.1 Basic Operations

Most algorithms for solving or simplifying CSPs (see Chapter 4) can be described by means of functions and their iterations. These functions are obtained by means of some basic functions on relations, common to most of those algorithms, and in addition some specific ones. We introduce the basic functions as below, since this will allow us to obtain a more general and compact notation to describe all the algorithms presented in Chapter 4.

Consider a domain $D$ over $X := x_1, \ldots, x_n$ and a scheme $s$ of $X$. Given $B \subseteq D \{s\}$ and a scheme $t := x_{j_1}, \ldots, x_{j_k}$ of $s$, the projection of $B$ over $t$ is defined as follows:

$$\Pi^s_t(B) := B_{j_1} \times \cdots \times B_{j_k}.$$ 

When the reference to $s$ is clear, we shall write $\Pi_t$ instead of $\Pi^s_t$. So, for instance, $D \{t\}$ is equal to $\Pi_t(D)$ for every scheme $t$ of $X$; in particular $D = \Pi_X(D)$. We extend the operation on tuples $d \in D \{s\}$ in the obvious way, and call $\Pi_t(d)$ the projection of the tuple $d$ on $t$, for $t$ a scheme of $s$.

We shall abuse notation and write $\Pi_t(B)$ and $\Pi_t(d)$, respectively, whenever the scheme reduces to the singleton scheme $x_j$, and refer to it as the projection over the variable $x_j$.

There is a sort of inverse operation to projection of domains and tuples, namely their join. To define this, let us consider two schemes $s := x_{i_1}, \ldots, x_{i_m}$ and
2.5. Combining and Comparing Problems

t := x_j, \ldots, x_k \text{ on } X, \text{ and let } r \text{ denote the scheme } s \cup t. \text{ Then, if } B \subseteq D[s] \text{ and } E \subseteq D[t], \text{ the join of } B \text{ and } E, \text{ denoted by }

B \bowtie E,

is the subset of } D[r] \text{ of tuples } d \text{ such that } d[s] \in B \text{ and } d[t] \in E. \text{ This implies that, if } r' \text{ stands for the scheme of the variables which are common to } s \text{ and } t, \text{ then } d \in D[r] \text{ yields } d[r'] \in B[r'] \cap E[r']. \text{ The join of tuples is defined similarly and, if } d \in D[s], \text{ } e \in D[t], \text{ their join is denoted by } d \bowtie e. 

Example 2.5.1. Consider the scheme } X := x_1, x_2, x_3, \text{ and the domains } D_1 := \{apple, banana\}, D_2 := \{chocolate, sugar\}, D_3 = \{dentist\}. \text{ If } s \text{ is the scheme } x_1, x_2, \text{ then } D[s] = D_1 \times D_2; \text{ i.e., the set }

\{(apple, chocolate), (banana, chocolate), (apple, sugar), (banana, sugar)\}.

Consider the subset } B := \{(apple, chocolate), (banana, chocolate)\} \text{ of } D[s]. \text{ Then the projection of } B \text{ over } x_2 \text{ is the set that only contains chocolate, whereas its projection over } x_1 \text{ collapses into } D_1. \text{ Similarly, if we consider the tuple } d := (banana, chocolate, dentist) \text{ from the Cartesian product } D_1 \times D_2 \times D_3, \text{ then } d[s] \text{ is } (banana, chocolate). \text{ While, if } t \text{ is the scheme } x_1, x_3, \text{ the tuple } d[t] \text{ is } (banana, dentist). 

Now, consider again the above subset } B \text{ of } D[s], \text{ the scheme } t = x_1, x_3, \text{ and the subset } E := \{(apple, dentist)\} \text{ of } D[t]. \text{ Then the join } B \bowtie E \text{ is the set }

\{(apple, chocolate, dentist)\};

whereas the join of } E \text{ and the subset } \{dentist\} \text{ of } D_3 \text{ is } E \text{ itself.}

2.5.2 Basic Orderings

As we shall clarify in Chapter 3, all the algorithms that we present in Chapter 4 transform a given CSP into another, but variables are neither added nor removed during this transformation process, so that only domains and constraints are modified.

Besides, those algorithms neither insert new values in the input domains, nor add domain elements to the input constraints. In other words, the algorithms in Chapter 4 transform CSP domains or constraints along a certain partial order, without backtracking in the order; in Chapter 3, we shall provide mathematical contents to these still vague claims. However, in the present section, we start introducing the main orders along which those algorithms transform CSPs.
The upward closure of CSPs

The notions of completion and its variants will allow us to easily define the orderings on CSPs that we shall encounter when dealing with constraint propagation.

Given two CSPs $P_1$ and $P_2$ on the same variable set, let us consider their completion $\bar{P}_1 := (X, D_1, C_1)$ and $\bar{P}_2 := (X, D_2, C_2)$. Then we write

$$P_1 \subseteq P_2,$$

if the following statements both hold:

- for each $x_i \in X$, we have $D_{1,i} \supseteq D_{2,i}$, where $D_{1,i}$ is the domain of $x_i$ in $P_1$ and $D_{2,i}$ the domain of $x_i$ in $P_2$;

- for each $C_1(s)$ in $\bar{P}_1$ and $C_2(s)$ in $\bar{P}_2$, the relation $C_1(s) \supseteq C_2(s)$ holds.

Therefore, two CSPs on the same variable scheme are comparable through $\subseteq$ iff the domains and constraints of their respective completions are comparable through the $\supseteq$ relation. Notice that, here and in the remainder of this thesis, we consistently choose to adopt $\subseteq$ instead of its reverse $\supseteq$ to denote the above relation. The motivation for this choice is that it has become standard in the mathematics and computer science literature (see [DP90]) to deal with partial orders or pre-orders with bottom, or complete partially ordered sets (CPOs) with bottom, least common fixpoints etc.; in other words, to use the $\supseteq$ relation.

Algorithms as in Chapter 4 receive in input a CSP $P$, and transform it into a CSP $P'$ such that the relation $P \subseteq P'$ holds. Therefore, it is sensible to circumscribe the set of problems those algorithms can produce, so to speak, starting from the input CSP $P$, as specified in the following definition.

**Definition 2.5.2.** Consider a CSP $P := (X, D, C)$ and the family $P^\uparrow$ of all problems $P'$ on $X$ for which the relation $P \subseteq P'$ holds. We call the family $P^\uparrow$ the upward closure or closure of $P$.

The above definition is commonly known in the theory of partial orders and pre-orders as the upward closure with respect to the given relation, see [DP90]. The first of the following results holds in general for all upward closures; the other follows immediately from Definition 2.5.2.

**Fact 2.5.3.** Consider a CSP $P$ and its closure $P^\uparrow$. Then the following statements hold:

1. if $P_1 \subseteq P_2$ and $P_1 \in P^\uparrow$, then $P_2 \in P^\uparrow$;

2. if $P_1 \subseteq P_2$, then $P_2^\uparrow \subseteq P_1^\uparrow$. \qed
2.5. Combining and Comparing Problems

Example 2.5.4. Consider the CSP $P$ with three variables, $x_1$, $x_2$, and $x_3$, whose domains are equal to $\{0, 1\}$, and with only two constraints, defined as follows: $C(x_1, x_2)$ states that $x_1 \neq x_2$, so it only contains the pairs $(0, 1)$ and $(1, 0)$; $C(x_2, x_3)$ states that $x_2 \neq x_3$, so it contains the same pairs as $C(x_1, x_2)$. The completion $\bar{P}$ has constraints on all the schemes of $x_1, x_2, x_3$. Therefore, a part from the $P$ constraints, $\bar{P}$ also has the following additional constraints:

- unary constraints on the given variables, $C(x_1), C(x_2)$, and $C(x_3)$, all equal to $\{0, 1\}$;
- $C(x_1, x_3)$ is equal to $\{0, 1\}^2$; i.e., it contains all pairs of 0 and 1;
- $C(x_1, x_2, x_3)$ is the set $\{0, 1\}^3$.

Problems in $P \uparrow$ can differ in domains or constraints from $P$. An instance of a CSP in $P \uparrow$ is the problem that is as $P$, except that its constraint $C(x_1, x_3)$ is equal to $\{(0,1), (1,0)\}$ — corresponding to the inequality $x_1 \neq x_3$. Another example is the $P \uparrow$ problem that has all domains empty, and constraints as $P$.

Unfortunately, the family $P \uparrow$ is too large, as the above example suggests: it still contains too many subproblems which are not related to any of the algorithms in Chapter 4. Thereby, in the remainder of this subsection, we carve out those subfamilies of $P \uparrow$ that are related to specific classes of algorithms in Chapter 4: i.e., domain orderings and constraint orderings.

Domain orderings

There are some algorithms for CSPs, such as arc consistency ones in Section 4.2, that only modify domains. Thus, consider a CSP $P := (X, D, C)$, and the closure $P \uparrow$. Let $\mathcal{F}(P)$ be a subfamily of $P \uparrow$, and assume that all problems in $\mathcal{F}(P)$ differ at most in their domains, but have the same constraints: i.e.,

$$
\text{if } P' \in \mathcal{F}(P) \text{ and } P' = (X, D', C'), \text{ then } C' = C.
$$

(2.2)

Then, if $P$ belongs to $\mathcal{F}(P)$, a partial ordering

$$(\mathcal{F}(P), \subseteq, P)$$

is a domain ordering over $P$. Notice that $P$ is the bottom (i.e., the least element) of such orderings.

When the family $\mathcal{F}(P)$ contains all the problems $P'$ that satisfy the property (2.2), we refer to the structure $\langle \mathcal{F}(P), \subseteq, P \rangle$ as the domain ordering on $P$.

The CSP algorithms presented in Section 4.2 are explained via iterations of functions that only modify CSP domain sets. So, given a domain ordering over $P$, say $\langle \mathcal{F}(P), \subseteq, \bot \rangle$, we name a function of the form

$$f : \mathcal{F}(P) \mapsto \mathcal{F}(P)$$

(2.3)
a domain function. However, all the problems in a domain ordering differ at most in their domains. Therefore, it is natural to regard a domain function as defined on the domains of $\mathcal{F}(P)$; i.e., if we introduce the family of domains

$$\mathcal{D}(P) := \{D' : \text{there exists } P' \in \mathcal{F}(P) \text{ such that } P' := (X, D', C)\}$$

and restrict the ordering $\subseteq$ on $\mathcal{F}(P)$ to the domains in $\mathcal{D}(P)$, then we can equivalently regard a function as in (2.3) as a function of the form

$$f : \mathcal{D}(P) \mapsto \mathcal{D}(P)$$

on the structure $\langle \mathcal{D}(P), \subseteq, D \rangle$, in which $D$ is the domain set of $P$. Notice that $D$ is the bottom of such orderings.

**Example 2.5.5.** Consider the problem $P$ in Example 2.5.4. The domain ordering on $P$ only contains problems that have the same constraints as $P$, and differ in their domains. Thus problems whose domains contain 0 or 1; problems that have some or all domains empty. The domain ordering on $P$ contains all such problems. A domain function is $\sigma(x_1; x_1, x_2)$ defined as follows on the domain family $\mathcal{D}(P)$ of $P$: if $B := B_1, B_2, B_3$ is in $\mathcal{D}(P)$, then $\sigma(x_1; x_1, x_2)(B)$ has domains $B_1', B_2'$ and $B_3'$ defined as follows:

$$B_1' := \Pi_1(C(x_1, x_2) \cap B_1 \times B_2),$$
$$B_2' := B_2,$$
$$B_3' := B_3.$$ 

The set $B'$ is still in the domain ordering of $P$, hence $\sigma(x_1; x_1, x_2)$ is a domain function. Analogously, functions of the form $\sigma(x_i; x_i, x_j)$ and $\sigma(x_j; x_i, x_j)$ can be defined for each pair of the problem variables $x_i$ and $x_j$ such that $i < j$. Such functions return in Section 4.2, where they are used to characterise certain algorithms for CSPs.

**Constraint orderings**

Algorithms for CSPs such as those in Section 4.3 (the so-called path consistency algorithms) do not modify domains, but alter constraints. These algorithms usually require to first complete the input CSP (see Definition 2.4.8).

Thus, consider a CSP $P := (X, D, C)$ with $n$ variables, and the closure $P\uparrow$ of $P$. Let $\mathcal{F}(P)$ be a subfamily of $P\uparrow$, and assume that all the problems in $\mathcal{F}(P)$ are $k$ complete, for $k \leq n$, and differ at most in their constraint sets: i.e.,

$$\text{if } P' := (X, D', C') \in \mathcal{F}(P), \text{ then } P' = \bar{P}'_k \text{ and } D' = D. \quad (2.4)$$

Then, if $\bar{P}_k$ belongs to $\mathcal{F}(P)$, the structure

$$\langle \mathcal{F}(P), \subseteq, \bar{P}_k \rangle \quad (2.5)$$

is a $k$-constraint ordering over $P$. Notice that $\bar{P}_k$ is the bottom of such orderings.
EXAMPLE 2.5.6. Consider the problem in Example 2.5.4. All 2-constraint orderings will contain $\tilde{P}_2$, that is the problem that has the same scheme, domain and constraints as $P$, plus the additional constraint $C(x_1, x_3)$ equal to the whole set $\{0, 1\}^2$. Those constraint orderings will differ for the binary constraints, that must be subsets of those of $\tilde{P}_2$. An example is the 2-constraint ordering that only contain $\tilde{P}_2$ and the problem whose constraints are all equal to $\{(0, 1)\}$.

Suppose that all the problems in a subfamily $\mathcal{F}(P)$ of $P_\uparrow$ are strongly $k$ complete and differ at most in their constraints: i.e.,

$$\text{if } P' := (X, D', C') \in \mathcal{F}(P), \text{ then } P' = P_k^x \text{ and } D' = D. \quad (2.6)$$

If the strong $k$ completion of $P$, namely $P_k^x$, belongs to $\mathcal{F}(P)$, then the structure

$$\langle \mathcal{F}(P), \subseteq, P_k^x \rangle \quad (2.7)$$

is a strong $k$-constraint ordering over $P$. Clearly, $P_k^x$ is the bottom of such orderings.

EXAMPLE 2.5.7. Let us consider Example 2.5.6. Every strong 2-constraint ordering will contain $\tilde{P}_2$, namely the problem that is as $\tilde{P}_2$ in Example 2.5.6, and has in addition the unary constraints on $x_1$, $x_2$ and $x_3$, all equal to $\{0, 1\}$. An instance of a strong 2-constraint ordering is given by the family of problems $\tilde{P}_2^x$ and $P'$, which is defined as follows: $P'$ only differs from $\tilde{P}_2^x$ in its unary constraints, which are all equal to the empty set.

Suppose that a family $\mathcal{F}(P)$ contains all the $k$ complete problems that satisfy the property (2.4). In this case, $\langle \mathcal{F}(P), \subseteq, P_k \rangle$ is called the $k$-constraint ordering on $P$. If $\mathcal{F}(P)$ contains all the strongly $k$ complete problems that satisfy (2.6), then we refer to the structure $\langle \mathcal{F}(P), \subseteq, P_k^x \rangle$ as the strong $k$-constraint ordering on $P$.

Domain orderings are useful to characterise some functions for certain CSP algorithms, and so are constraint orderings. Thus, given a constraint ordering (2.5) or (2.7), we name a function of the form

$$f : \mathcal{F}(P) \mapsto \mathcal{F}(P) \quad (2.8)$$

a constraint function. Since all the problems in a constraint ordering (2.5) or (2.7) differ at most in their constraints, we can redefine a constraint function as a function on the constraint set $C$ of the constraint ordering (2.5) or (2.7). In fact, if we introduce the family of constraint sets

$$C(P) := \{C' : \text{there exists } P' \in \mathcal{F}(P) \text{ such that } P' := (X, D, C')\}$$

and restrict the ordering $\subseteq$ on $\mathcal{F}(P)$ to the constraint sets in $C(P)$, then we can equivalently regard a function as in (2.8) as a function of the form

$$f : C(P) \mapsto C(P)$$

on the structure $\langle C(P), \subseteq, C \rangle$. Notice that $C$ is the bottom of such orderings.
Example 2.5.8. A constraint function on the 2-constraint ordering in Example 2.5.6 is the function \( \sigma(x_1, x_2; x_3) \) defined as follows. Consider the constraint set \( B := B(x_1, x_2), B(x_2, x_3), B(x_1, x_3) \) in \( C(P) \); then \( \sigma(x_1, x_2; x_3)(B) \) has constraints \( B'_1(x_1, x_2), B'_2(x_2, x_3) \) and \( B'_3(x_1, x_3) \) defined as

\[
\begin{align*}
B'_1(x_1, x_2) &:= B(x_1, x_2) \cap \Pi_{x_1, x_2}(B(x_1, x_2) \uplus B(x_2, x_3)), \\
B'_2(x_2, x_3) &:= B(x_2, x_3), \\
B'_3(x_1, x_3) &:= B(x_1, x_3).
\end{align*}
\]

Thus \( \sigma(x_1, x_2; x_3)(B) \) is still in the 2-constraint ordering on \( P \), hence \( \sigma(x_1, x_2; x_3) \) is a constraint function. Analogously, functions like \( \sigma(x_i, x_j; x_k) \) can be defined for each pairwise distinct \( i < j \) and \( k \), from 1 to 3. These functions return in Section 4.3, where they are used to characterise certain algorithms for CSPs.

2.6 Conclusions

In this chapter, CSPs are introduced: these are shown to generalise a number of well-known problems, such as map colourability, \( n \)-SAT, temporal and spatial reasoning, scheduling and planning problems. In general, the task of finding a solution to these problems is intractable. In the CSP community, a number of approximate algorithms were devised for removing inconsistencies from the solution search space of CSPs; these algorithms are used before the search for solutions, or to this alternated.

In the following chapter, we present a simple theory to describe and analyse all those algorithms, and we put it at work in Chapters 4 and 5.