Introduction

The concept of *constraint* is central to a number of human activities. A constraint limits the field of possibilities in a certain universe. For example, a school timetable that coordinates students, teachers, lessons, rooms and time slots, must satisfy many constraints. Typically, for each group of students, the objective is to fill up one sheet such as the one shown[1] in Figure 1. In each time slot, you have to indicate who the teacher is, what the lesson is, and where it is located. Obviously, not all combinations are possible, since the constraints are numerous and various:

– no teacher can teach more than one class at the same time;

– different classes cannot be taught in the same room at the same time;

– classes cannot be taught in rooms that are too small, and preferably should not be taught in rooms that are much too big;

– some classes require specialized rooms such as science laboratories;

– some classes require consecutive periods in the same room with the same teacher;

– some part-time teachers need to have certain entire days off;

– students cannot have too far to travel between consecutive classes.

Besides school timetabling, constraint satisfaction problems arise in many enterprise and industrial tasks, ranging from scheduling to configuration, circuit design and molecular biology.

Constraint programming (CP) is a general framework providing simple, general and efficient models and algorithms for solving real-world and academic problems. The appeal of constraint programming mainly relates to the clear distinction between, on the one hand, its formalism, which facilitates the representation of various

1. All figures can be downloaded at http://www.iste.co.uk/Lecoutre/cn.zip

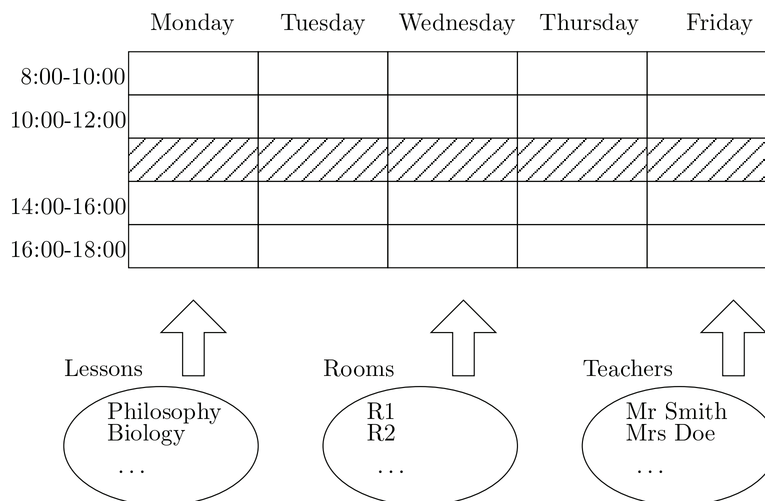


Figure 1. *The timetable assigned to a group of students. Filling timetable sheets is a constraint satisfaction problem*

problems by means of constraints, and, on the other hand, a vast range of algorithms and heuristics to solve them. Practical use of this framework involves two main stages. In the first of these, the user represents the problem abstractly by a *constraint network*, which is a set of variables together with a set of constraints, and perhaps also one or more objective functions. Ideally, this first stage is purely declarative, but in practice, some limited form of programming may be required (using e.g. an object-oriented or logic programming language). In the second stage, the problem represented by the constraint network is tackled by an available software tool, known as a *constraint solver*, that automatically obtains one solution, or all solutions, or an optimal solution, to the given problem. A *solution* is an assignment of values to all variables such that all constraints are satisfied.

A constraint network is a formulation of an *instance* of the *constraint satisfaction problem* (CSP) which is at the core of constraint programming. In a *discrete* instance, the domains, which are the sets of allowed values of variables, are finite. The discrete constraint satisfaction problem is not known to admit polynomial running time algorithms to solve its instances. More precisely, unless $\mathrm{P} = \mathrm{NP}$ (which is very unlikely to be the case), no such general algorithm can exist, since CSP is NP-hard[2]. This means that the worst-case time complexity of any algorithm for solving CSP instances is expected to be exponential. However, the worst case actually arises only

2. Complexity analysis is briefly introduced in Appendix A.2

within a limited range of situations, and outside this range efficient algorithms are already available. Efficiency is achieved by exploiting the structure of instances.

Although this book is focused on CSP, this problem or framework has many derivatives, mainly extensions, as indicated in Figure 2: temporal CSP (TCSP), weighted CSP (WCSP), valued CSP (VCSP), quantified CSP (QCSP), constraint optimization problem (COP), Max-CSP, distributed CSP (DisCSP), etc. Quite often, a concept or technique introduced for basic CSP has turned out to be relevant to its extensions. For example, the concept of arc consistency has been applied to most of these extensions.

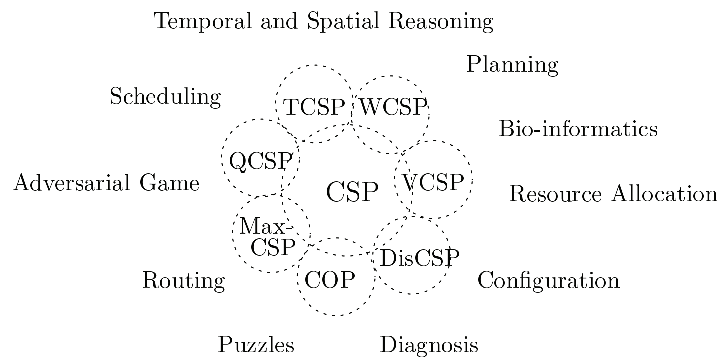


Figure 2. *The CSP framework and some of its extensions*

I. *Toward simplicity of use*

The ability to take heterogeneous constraints into account under a unifying framework has contributed to growing commercial interest in constraint programming since the 1990s. Modeling a problem may, however, turn out to be very difficult for the uninitiated user, as, for example, the number of specific patterns of constraints, called *global constraints*, may be unexpectedly large. In some cases, there is a need for specialized expertise to take full advantage of the efficiency of available techniques and algorithms.

A solver applies constraints so as to avoid exploring combinations of values that cannot possibly belong to any solution. Ideally, the operation of a solver should be totally transparent to the user: the user should not be aware of specific short-cuts used by the solver. Unfortunately, this idyllic vision is not exactly correct in reality because most of the currently available constraint toolkits require the user to guide search, to select algorithms to filter the search space, to break symmetries, etc. As pointed out by Puget [PUG 04], an important challenge for constraint programming is

to achieve greater simplicity of use: constraint programming should be made easier for non-specialist users. Enhanced ease of use will boost the impact of constraint programming on industry and academia, and will establish it more firmly as a key software technology for solution of combinatorial problems.

To take up the "simplicity of use" challenge, there is a need for robust and efficient solvers that users can regard as *black-boxes*. A black-box is a system such that the user sees only its input and output data, while its internal structure or mechanism remains invisible. This approach has recently been emphasized by some position papers [PUG 04, GEN 06a] as well as the holding of constraint solver competitions[3]. The black-box approach partially addresses the requirement for simplicity since the user does not have to be aware of (or modify or extend) embedded techniques and algorithms. However, a black-box constraint solver must have a default configuration that in most cases yields the best behavior that could be obtained by fine tuning of available options. This can be achieved by making the solver *robust*.

A solver is robust when it is able to produce similar results, consuming similar resources (time and space), given different but equivalent models of the same problem. It is important to note that the user of an ideally robust solver does not need to provide carefully chosen constraint network models. Robustness compensates for bad modeling by providing sophisticated solving techniques. Some of these certainly remain to be invented, but others are presented in this book: inferences from strong consistencies, adaptive heuristics, nogood recording, automatic symmetry breaking, state-based search, etc. These techniques enable a particularly clever exploration of the search space, learning much useful information before or during search so as to avoid exploring fruitless combinations of values of variables. Given different formulations of the same CSP instance, advanced learning and inference techniques reduce behavior disparities by increasing the efficiency of the solver. Thus robustness and efficiency are intimately interrelated.

II. *Conceptual simplicity of techniques and algorithms*

Robust and efficient black-box solvers are intended to simplify the life of users. However, identifying and implementing appropriate state-of-the-art techniques and algorithms can be quite a hard task for black-box designers and developers. It is not easy to distinguish the most important algorithms among the large number that have been published. Moreover, certain algorithms require complex data structures and procedures that have not been disclosed in complete detail, so re-implementation is hazardous. Luckily, many of the substantial new developments that have appeared during the last decade are characterized by conceptual simplicity of techniques

3. See http://www.cril.univ-artois.fr/CPAI08/

and algorithms. This book attempts to present these developments comprehensively and rigorously, offering you a gentle introduction to this active field of research. Pragmatically, the book concentrates on general-purpose approaches that have proven to be effective in practice. These approaches are the source of a nascent generation of robust constraint solvers accessible to the average user.

If we insist on (conceptual) simplicity, this is because it has many nice features. Although these may be obvious, they deserve brief comment as follows. First, simplicity may be understood primarily as ease of comprehension. An easily understood principle is, from the master's point of view, easy to explain and, from the disciple's point of view, quick to assimilate. The difficulty in the comprehension of the world or of nature certainly lies in finding the elementary principles that enable explanation of the Creation. Modestly, in our context, the difficulty lies in finding the basic recipes that are at the origin of the efficiency of algorithms.

Another comment about simplicity is that it tends to make development easier. Proposed algorithmic solutions eventually become procedures written in programming languages. Software development time can be reduced, and more robust code can be written, if an algorithm is easy to code. Ease of coding usually depends on the complexity of the data structures that are employed. Generally, the shorter the code that implements an algorithm, the less the risk of bugs therein.

A final comment about simplicity concerns its impact on the reproduction of experiments. If a method is simple to understand and to implement, this simplicity substantially increases the probability that two people independently evaluating the method will develop similar (source) code and consequently obtain similar experimental results. Surely, science is nothing without the possibility of reproducing experiments (and, more generally, without the possibility of checking theoretical results).

III. *Organization of this book*

In the first chapter, constraint networks are introduced with the formalism that surrounds them. Formal foundations are then given, and several examples of constraint satisfaction problems are presented. In the second chapter, we study the nature of constraint networks, essentially discussing the presence or absence of structure in problems. The remainder of the book is divided into two parts.

The first part describes general inference methods based on local consistencies, which are relational and structural properties of constraint networks. The principle is to simplify the problem instance that must be solved by discarding some useless portions of the search space. This is made possible by propagating constraints following a targeted consistency that allows identification of inconsistent

instantiations. Chapter 3 provides an overview of the consistencies usually studied in constraint satisfaction. Following usual practice, we concentrate mainly on first-order (or domain-filtering) consistencies that identify globally inconsistent values. Chapter 4 describes generic algorithms proposed to enforce the central consistency in constraint programming, namely (generalized) arc consistency; such algorithms are universal, as they can theoretically be used for any type of constraints. In Chapter 5, we restrict our attention to table constraints, that is to say, constraints defined by explicitly listing allowed (or forbidden) combinations of values. We describe very recent propagation schemes that have led to significant progress. In Chapter 6, we are interested in singleton arc consistency, a consistency built upon (generalized) arc consistency. We introduce some recent approaches that make use of the incrementality of arc consistency algorithms in different ways. Finally, in Chapter 7, we study dual consistency, which is a consistency related to path consistency.

The second part of the book presents general search methods that cleverly explore the search space of combinatorial problems. The basic idea of these methods is to gather useful information, before and especially during a search, so as to guide the search efficiently. Chapter 8 presents the concept of backtrack search, together with classical look-back and look-ahead schemes. Chapter 9 explains how dead-ends encountered during a search can be quite helpful in guiding the search toward sources of conflicts. The guidance heuristics involve constraint weighting and last-conflict reasoning. Chapter 10 investigates nogood recording, in conjunction with the idea of regularly restarting search. Nogoods can easily be extracted from the current state of search before each restart, and exploited in subsequent runs to discard portions of the search space that have already been explored. Chapter 11 introduces the promising related approach of exploiting partial states extracted, using sophisticated operators, throughout the search. Finally, Chapter 12 addresses the automatic breaking of symmetries. This is an important reasoning mechanism that allows symmetric portions of the search space to be discarded.

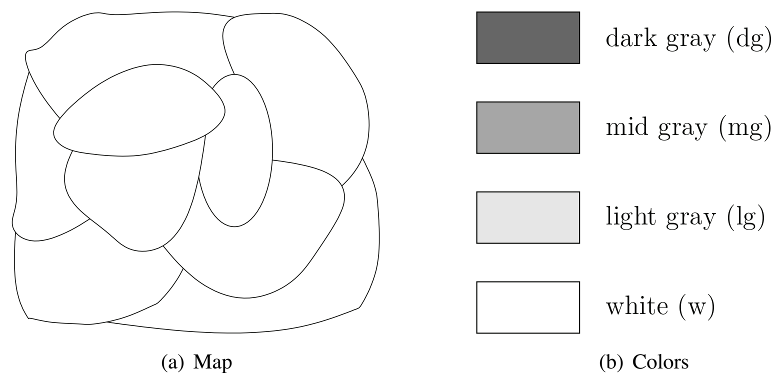
We wish to emphasize that many algorithms presented in this book have been implemented in our constraint solver Abscon. This solver is primarily intended to serve as a platform for the scientific development of research ideas. Incidentally, it participates in constraint solver competitions. We also wish to emphasize that this book does not attempt exhaustive coverage of all topics in the constraint processing field. It is intended mainly to promote the artificial intelligence approach to constraint programming, and is unsurprisingly built upon the experience of the author, making some sections rather personal.

IV. *Introductory example*

Most of the concepts introduced in this book refer to either inference or search. Nevertheless, sometimes concepts refer to both principles of inference and search.

This is the reason why we propose[4] an example to gently introduce the central notions of consistency and backtrack search. Map coloring is the problem chosen for this example.

The goal of a *map coloring problem* is to color a map so that adjacent regions, i.e. regions sharing a common border, have different colors. Figure 3(a) shows a map that has nine regions which need to be colored. The four color map theorem (e.g. see [WIL 05]) states that given any plane separated into regions, such as a political map of the states of a country, the regions can be colored using no more than four colors. Thus, we propose to color the map shown in Figure 3(a) with the four colors shown in Figure 3(b).



(a) Map (b) Colors

Figure 3. *A map with nine regions to be colored using four colors*

The map together with the colors shown in Figure 3 is an *instance* of the map coloring problem. We can represent this instance by a *constraint network* P which is a structure composed of variables and constraints. A *variable* is an unknown, which must be given, or *assigned*, a value from an associated *domain*. Naturally, the variables of our constraint network correspond to the nine regions of the map, and the domain of each variable contains the four available colors. The variables are $\{x_1, x_2, \ldots, x_9\}$ and the domains are $\{dg, mg, lg, w\}$, where dg stands for dark gray, mg stands for mid gray, etc. Figure 4 illustrates this. A *constraint* restricts the possible combinations of values of some variables. Since adjacent regions must be colored differently, we introduce a constraint on every pair of variables that represent adjacent regions. Such a *binary* constraint states that the values assigned to the two variables involved in this constraint must be different. We just use inequation constraints. For example, we have

4. I would like to thank Julian Ullmann for having suggested this to me.

$x_1 \neq x_2$ since x_1 and x_2 represent two adjacent regions located in the north of the map.

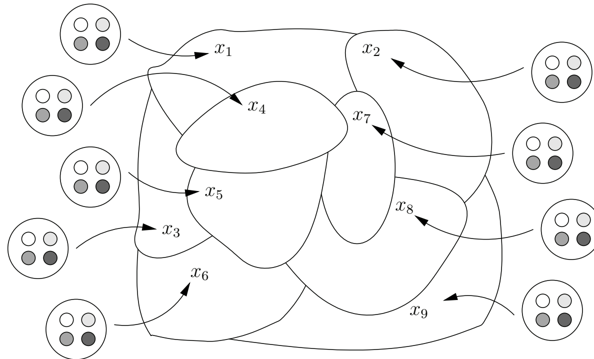


Figure 4. *Each region of the map is represented by a variable x whose domain is the set $\{dg, mg, lg, w\}$, that is, the four available colors*

It may be useful to associate a *constraint graph* with a (binary) constraint network so as to benefit from well-known results from graph theory. A constraint graph is an undirected graph built from a constraint network such that there is a vertex per variable, and there is an edge per pair of variables involved in a constraint. Figure 5 shows the constraint graph for our example. Using the constraint graph of the map coloring problem, we obtain an equivalent *graph coloring problem*: color the vertices of the graph such that adjacent vertices, i.e. vertices linked by an edge, have different colors.

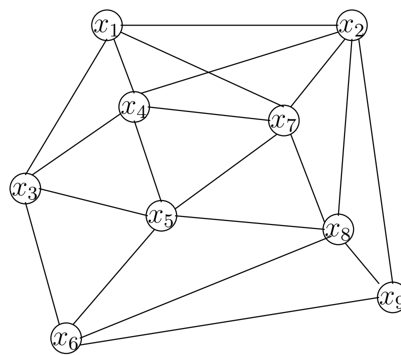


Figure 5. *The constraint graph associated with the constraint network partially depicted in Figure 4. Here, vertices are labeled with the variable names they represent*

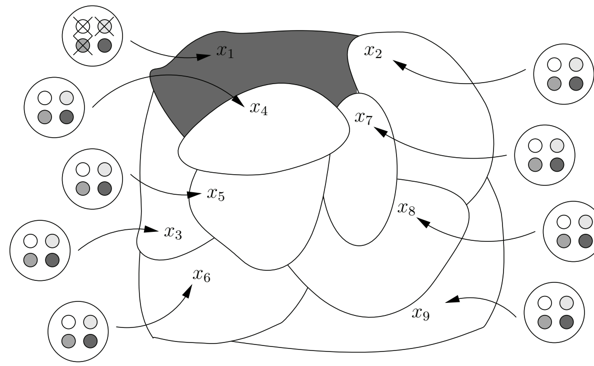
To find a solution for this problem, we need *search*. In its complete form, search performs an exhaustive exploration of the search space. The *search space* is basically the Cartesian product of the domains of the variables; here, as we have nine variables and four values per domain, we obtain a search space whose size is 4^9. This represents $262{,}144$ different configurations, or *complete instantiations*, for the constraint network. Enumerating every complete instantiation in turn and checking each one to see whether it satisfies all the constraints appears to be quite inefficient; this is a method called *generate and test*.

To improve the performance of the "generate and test" approach, it is possible to perform a depth-first exploration of the search space, verifying at each step that it may still be possible to find a solution. Variables are assigned, or *instantiated*, in turn, thereby forming *partial* instantiations. At each step, the local consistency of the partial instantiation can be checked: the partial instantiation is *locally consistent* iff each constraint *covered* by it (i.e. each constraint only involving instantiated variables) is satisfied.

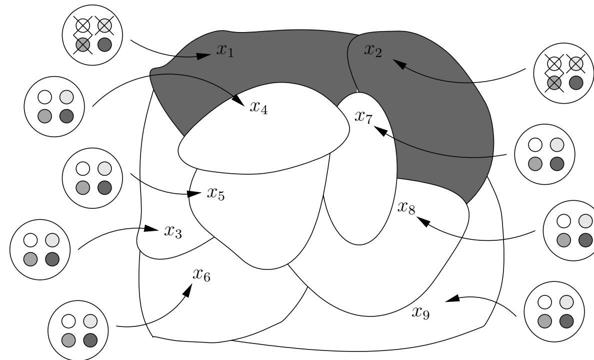
For our example, a *depth-first search* (DFS) starts by assigning dg to x_1; see Figure 6(a). The partial instantiation $\{x_1 = dg\}$ is locally consistent because no constraint is covered by it (all constraints are binary). Then, DFS assigns dg to x_2; see Figure 6(b). This time, the partial instantiation is not locally consistent because the constraint $x_1 \neq x_2$ is covered and violated. No solution can be found by extending this partial instantiation, which corresponds to a *dead-end* situation and is called a *nogood*. This is why another value for x_2 is tried by the search; see Figure 6(c).

Assume now that the (locally consistent) partial instantiation $\{x_1 = dg, x_2 = mg, x_3 = mg, x_4 = w, x_5 = lg, x_6 = dg\}$ must be extended over x_7; see Figure 7(a). It is easy to see that any assignment to x_7 yields an inconsistent instantiation because x_1, x_2, x_4 and x_5 are adjacent to x_7 and have all been assigned different colors. Otherwise stated, no color remains possible for x_7. Consequently, after four tentative assignments for x_7 (because the domain of x_7 is composed of four values), the search has to return to the variable that was instantiated before x_7, which is x_6. When the search returns to a previous variable, we say that the search algorithm backtracks; this general principle is called *backtracking*. Depth-first search (with backtracking) is also called *backtrack search*. In our example, after backtracking from x_7, another value for x_6 must be tried; this is color mg as shown in Figure 7(b). This new assigned color is immediately discarded because the constraint $x_3 \neq x_6$ is violated. For a similar reason, lg is discarded, and so the only remaining possibility is to try w for x_6; see Figure 7(c). However, after assigning w to x_6, the algorithm again performs the same useless tentative instantiations of x_7, although the value of x_6 has no bearing on these failures. Rediscovering the same failure situations during search is called *thrashing*.

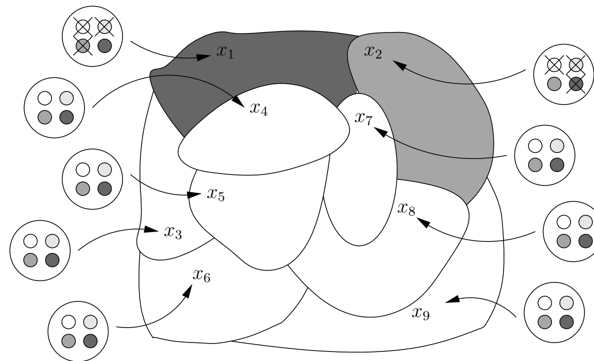
Finally, it seems reasonable to prevent conflicts that can easily be anticipated (so as to prevent, or at least reduce, thrashing). For example, if at the beginning of



(a) DFS assigns dg to x_1

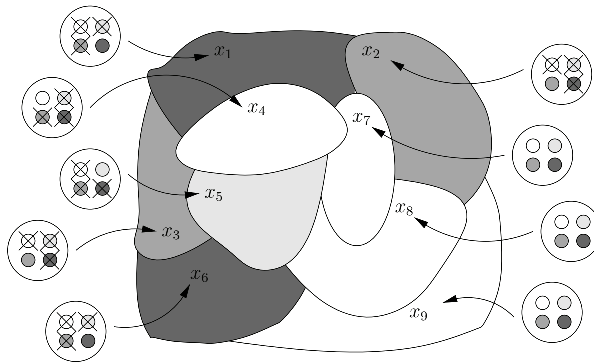


(b) DFS assigns dg to x_2. The partial instantiation is not locally consistent because the constraint $x_1 \neq x_2$ is violated.

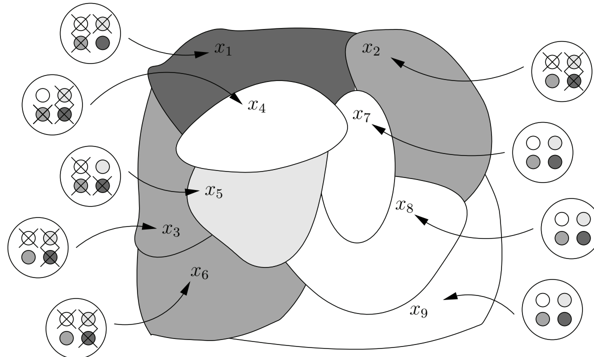


(c) DFS tries another assignment for x_2 (mg is assigned to x_2). The new partial instantiation is locally consistent.

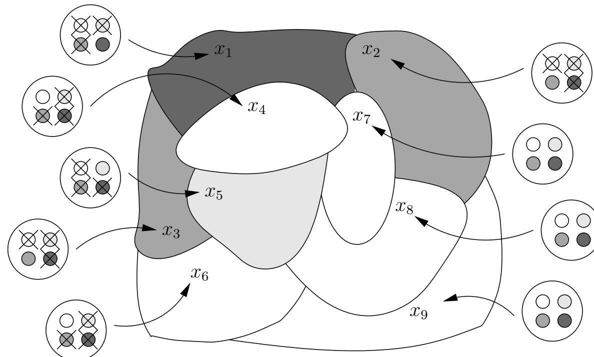
Figure 6. *The early steps performed by DFS (depth-first search)*



(a) The partial instantiation $\{x_1 = dg, x_2 = mg, x_3 = mg, x_4 = w, x_5 = lg, x_6 = dg\}$ must be extended over x_7. No extension is locally consistent: search has to backtrack to x_6.



(b) After backtracking to x_6, a new value has been assigned to x_6. This value (as well as lg) is immediately discarded because the new partial instantiation is not locally consistent.



(c) The value w is now assigned to x_6. Four useless tentative assignments to x_7 will be performed again. This is a phenomenon called thrashing.

Figure 7. *Illustration of backtracking and thrashing*

search the value dg is assigned to x_1, then clearly this value can be removed from the domain of the variables in the neighborhood of x_1, namely x_2, x_3, x_4 and x_7; see Figure 8. A value for an uninstantiated variable is incompatible with the value of the last instantiated variable if there is a constraint that prevents these two variables from taking these values simultaneously. Such incompatible values are not *arc-consistent* and can be safely deleted without losing any solutions. Deletion of inconsistent values is called *filtering* of the domains.

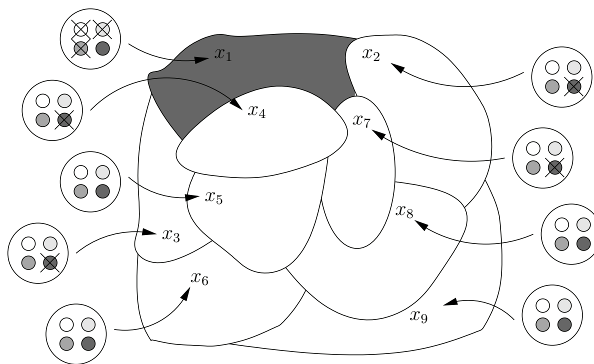


Figure 8. *By reasoning locally from constraints after* dg *is assigned to* x_1*, we deduce (infer) that the value* dg *can be safely removed from the domains of* x_2*,* x_3*,* x_4 *and* x_7

Sophisticated backtrack search algorithms interleave search steps and filtering inference processes that can identify inconsistent partial instantiations of arbitrary size. Before starting search, constraint networks are usually processed during a so-called *preprocessing* stage. Typically, inferences such as removing inconsistent values are performed at preprocessing time. Sometimes preprocessing alone is sufficient to solve a problem instance.

Chapter 1

Constraint Networks

This chapter introduces the formalism of constraint networks, which can abstractly represent many academic and real-world problems. Section 1.1 introduces variables and constraints, which are the main ingredients of constraint networks. This introduction includes different representations of constraints as well as the vital concept of constraint support. In section 1.2, we formally define constraint networks. Moreover, we present the (hyper)graphs that can be associated with any constraint network, and introduce instantiations. Section 1.3 provides some illustrative examples of problems that can be easily represented by means of constraint networks. For simplicity and entertainment, these examples are based on logic puzzles. Section 1.4 is concerned with partial orders in constraint networks, decisions and general properties of values and variables. Finally, section 1.5 introduces some data structures that can be employed to represent constraint networks in computer programs.

1.1. Variables and constraints

Here we will define variables and constraints, which are the main ingredients of constraint networks. They constitute the surface part of a problem representation, whereas domains and relations constitute the underlying part. In object-oriented design, we would certainly build up a class for variables and another for constraints, and represent all relevant information about variables and constraints in terms of attributes (maybe introducing additional classes) for these objects: identifier, domain, scope, relation, etc.

DEFINITION 1.1.– *[Variable] A variable, which is a component of an abstract system, is an object that has a name and is able to take different values. In our context, a* variable *(whose name is) x must be given a value from a set, which is called the* current

domain *of* x *and is denoted by* $\text{dom}(x)$*. The domain of a variable* x *may evolve over time, but it is always included in a set called the* initial domain *of* x*.*[1] *This initial domain, which is denoted by* $\text{dom}^{\text{init}}(x)$*, represents the full universe of the variable* x*.*

A *continuous* variable has an infinite initial domain, usually defined in terms of real intervals. Continuous variables are outside the scope of this book, which only considers discrete variables. A *discrete variable* is a variable whose initial domain contains a finite number of values.

We use letters x, y, z (and when necessary u, v, w), possibly subscripted or primed, to denote variables. Without any loss of generality, our variables can be assumed to have integer values in their domains when necessary. Quite often, letters a, b, c, possibly subscripted or primed, will be used to denote values. For example, x and y such that $\text{dom}^{\text{init}}(x) = \{a, b\}$ and $\text{dom}^{\text{init}}(y) = \{1, 2, \ldots, 100\}$ are two discrete variables whose initial domains contain 2 and 100 values, respectively.

Domains are dynamic sets, i.e. they may change over time. A variable is said to be *fixed* when its current domain only contains one value, and *unfixed* otherwise. A variable can be fixed either explicitly or implicitly (incidentally). When a variable x is explicitly given a value a from its current domain $\text{dom}(x)$ during the progression of a scenario or an algorithm, every other value $b \neq a$ is considered to be removed from $\text{dom}(x)$. In this case we say that the variable x is *instantiated*; otherwise, we say that x is *uninstantiated*. We also say that the variable x is *assigned* (the value a) or that the value a is assigned to x. Assigning a value to a variable is called a *variable assignment*. Implicitly fixed variables occur when deduction (inference) mechanisms are used. For example, consider the equality $x = y$ between two variables x and y whose (common) current domain is $\{1, 2\}$. If the variable x is assigned the value 1, by reasoning from the equality we can deduce that y must also be equal to 1, i.e. the value 2 can be removed from $\text{dom}(y)$ by deduction. The two variables are then fixed, the first one explicitly and the second one implicitly. However, only the first variable is considered to be instantiated (or assigned).

A value a is said to be *valid* for a variable x iff $a \in \text{dom}(x)$. Because of changes in $\text{dom}(x)$, a value that is valid for x at time t may be invalid at another time t'. To keep track of those changes, it can be helpful to use a superscript t to denote the time at which we refer to a domain: $\text{dom}^t(x)$ is the domain of x at time t. With t_0 representing the time origin we have, for every variable x, $\text{dom}^{t_0}(x) = \text{dom}^{\text{init}}(x)$. Actually, as we shall see later, instead of using time, we use constraint networks as superscript for domains. Indeed, when we reason about several related constraint networks, it is

---

1. We can imagine situations where initial domains could be enlarged. However, no technique presented in this book allows us to do that.

expedient to write $\text{dom}^P(x)$ to denote the domain of x in constraint network P. When the context is unambiguous, we simply use $\text{dom}(x)$.

In this book, without any loss of generality, we assume that (names of) discrete variables belong to an infinite totally ordered set, with the (strict) total order denoted by $\lhd$; thus $x \lhd y$ means that variable x (strictly) precedes y within this order. Consequently, any set of variables handled in the remainder of this book is assumed to be totally ordered by $\lhd$.

REMARK 1.2.– *[Total Order on Variables] Any set X of variables is totally ordered according to the relation $\lhd$.*

Similarly, without any loss of generality, we assume that values are always taken from a totally ordered set, with the (strict) total order denoted by $<$; thus $a < b$ means that the value a (strictly) precedes b within this order. Consequently, any set of values handled in the remainder of this book is assumed to be totally ordered by $<$.

REMARK 1.3.– *[Total Order on Values] Any set V of values is totally ordered according to the relation $<$.*

To define constraints, we introduce *tuples*, *Cartesian product* and *relations*. More information about sets, relations, etc. can be found in Appendix A.1.

DEFINITION 1.4.– *[Tuple] A* tuple *τ is a sequence, usually enclosed between parentheses, of values separated by commas. A tuple containing r values is called an* r-tuple. *The ith value of an r-tuple, with $1 \leq i \leq r$, is denoted by $\tau[i]$.*

As values are taken from a totally ordered set, r-tuples can be lexicographically ordered by extending the relation $<$. The new strict total order is denoted by $<_{\text{lex}}$, and the corresponding non-strict total order is denoted by $\leq_{\text{lex}}$.

DEFINITION 1.5.– *[Lexicographic Order] Let τ and τ' be two r-tuples.*

– *$\tau <_{\text{lex}} \tau'$ iff $\exists i \in 1..r$ such that $\tau[i] < \tau'[i]$ and $\forall j \in 1..i-1, \tau[j] = \tau'[j]$.*

– *$\tau \leq_{\text{lex}} \tau'$ iff $\tau <_{\text{lex}} \tau'$ or $\tau = \tau'$.*

EXAMPLE.– Considering values taken from $\mathbb{N}$, we have:

– $(2,4,7,6) <_{\text{lex}} (3,3,3,8)$;

– $(2,4,7,6) <_{\text{lex}} (2,4,8,2)$;

– $(2,4,7,6) <_{\text{lex}} (2,4,7,8)$.

A Cartesian product is a set composed of all tuples that can be built from a sequence of sets.

DEFINITION 1.6.– *[Cartesian Product] Let $D_1, D_2, \ldots, D_r$ be a sequence of r sets. The* Cartesian product $D_1 \times D_2 \times \cdots \times D_r$*, also written $\prod_{i=1}^{r} D_i$, is the set $\{(a_1, a_2, \ldots, a_r) \mid a_1 \in D_1, a_2 \in D_2, \ldots, a_r \in D_r\}$. Each element of $\prod_{i=1}^{r} D_i$ is an r-tuple.*

EXAMPLE.– We can define Cartesian products of domains of variables. For example, if x, y and z are three variables such that $\mathrm{dom}(x) = \mathrm{dom}(y) = \{a, b\}$ and $\mathrm{dom}(z) = \{a, c\}$, we have:

$$\mathrm{dom}(x) \times \mathrm{dom}(y) \times \mathrm{dom}(z) = \left\{ \begin{array}{c} (a, a, a), \\ (a, a, c), \\ (a, b, a), \\ (a, b, c), \\ (b, a, a), \\ (b, a, c), \\ (b, b, a), \\ (b, b, c) \end{array} \right\}$$

A relation is simply a subset of a Cartesian product.

DEFINITION 1.7.– *[Relation] A* relation *R defined over a sequence of r sets $D_1, D_2, \ldots, D_r$ is a subset of the Cartesian product $\prod_{i=1}^{r} D_i$, so $R \subseteq \prod_{i=1}^{r} D_i$.*

We also say that R is defined on $\prod_{i=1}^{r} D_i$.

EXAMPLE.– Here is a relation defined on $\mathrm{dom}(x) \times \mathrm{dom}(y) \times \mathrm{dom}(z)$:

$$R_{xyz} = \left\{ \begin{array}{c} (a, a, c), \\ (b, a, a), \\ (b, a, c), \\ (b, b, c) \end{array} \right\}$$

We can now introduce the central concept of *constraint*.

DEFINITION 1.8.– *[Constraint] A constraint, which is a component of an abstract system, is represented by a name and is a restriction on combinations of values that can be taken simultaneously by a set of variables. In our context, a* constraint *(whose name is) c is defined over a (totally ordered) set of variables, which constitute the* scope *of c and are denoted by $\mathrm{scp}(c)$. A constraint c is defined by a relation, denoted by $\mathrm{rel}(c)$, comprising exactly the set of tuples allowed by c for the variables of its scope; we have $\mathrm{rel}(c) \subseteq \prod_{x \in \mathrm{scp}(c)} \mathrm{dom}^{\mathrm{init}}(x)$.*

The letter c, possibly subscripted with the sequence of scope variables, or possibly primed, is used to denote a constraint. For example, the constraint c_{xyz} is such that

$\text{scp}(c_{xyz}) = \{x, y, z\}$. Sometimes we use the symbol c to denote a value for a variable, but the context is always sufficient to distinguish between a constraint and a value.

A tuple τ allowed by c is also said to be *accepted* by c, and we say that τ *satisfies* c. A tuple that is not allowed by c is said to be *disallowed* or *forbidden* by c, and we say that c is unsatisfied, or *violated*, by τ. For example, if c_{xyz} is a constraint such that $\text{rel}(c_{xyz}) = R_{xyz}$, where R_{xyz} is the relation introduced above, then (b, a, c) is an allowed tuple, whereas (a, b, a) is disallowed by c_{xyz}.

A variable x that belongs to $\text{scp}(c)$ is said to be *involved* in c. Note that $\text{scp}(c)$ is totally ordered according to the relation $\lhd$; see Remark 1.2. Consequently, in Definition 1.8 the order of the domains in the Cartesian product corresponds to the order of the variables for which they are the domains. We use $\text{scp}(c)[i]$ in some algorithms to denote the ith variable involved in $\text{scp}(c)$, with $1 \leq i \leq |\text{scp}(c)|$. Two constraints c and c' such that $\text{scp}(c) \cap \text{scp}(c') \neq \emptyset$ are said to *intersect*. For example, c_{xyz} and c_{wy} are two constraints that intersect on variable y. The *arity* of a constraint c is the number of variables involved in c, i.e. $|\text{scp}(c)|$. A constraint is:

– unary iff its arity is 1;

– binary iff its arity is 2;

– ternary iff its arity is 3;

– non-binary iff its arity is strictly greater than 2.

Notice that a non-binary constraint is considered as being neither binary nor (more surprisingly) unary. The reason is that, as we shall see later, unary constraints defined on discrete variables can easily be discarded (and so ignored).

Definition 1.8 is a little bit more general than the one usually employed, which is confined to *tailored constraints*. This has to do with the concept of embedded constraint networks introduced in [BES 06].

DEFINITION 1.9.– *[Tailored Constraint] A constraint c is said to be* tailored *iff* $\text{rel}(c) \subseteq \prod_{x \in \text{scp}(c)} \text{dom}(x)$.

When a constraint is tailored, every allowed tuple only involves valid values, i.e. values in current domains. When it is not tailored, we may have $\text{rel}(c) \not\subseteq \prod_{x \in \text{scp}(c)} \text{dom}(x)$, but by definition we know that $\text{rel}(c) \subseteq \prod_{x \in \text{scp}(c)} \text{dom}^{\text{init}}(x)$. The general definition 1.8 is useful in dynamic situations, as we shall see later. In practice, constraints are tailored when they are defined; but when domains of variables change, constraints do not systematically remain tailored.

It is important to note that constraint relations may also change over time; this is a feature of various approaches such as enforcing path consistency or pairwise consistency, which are introduced later. The state of a constraint c at time t is given

by the state of $\text{rel}(c)$ at time t, and also, indirectly, by the state of the domains of the variables involved in c at time t. To keep track of changes, if any, in a constraint relation, we can use a superscript t so that $\text{rel}^t(c)$ is the relation of c at time t.

EXAMPLE.– Figure 1.1 illustrates the dynamic aspect of constraint relations with a ternary constraint c_{xyz} (with $\text{scp}(c_{xyz}) = \{x, y, z\}$). We have $\text{dom}^{\text{init}}(x) = \text{dom}^{\text{init}}(y) = \text{dom}^{\text{init}}(z) = \{a, b, c\}$. At time t_0, the initial tailored constraint is defined. At time t_1, two allowed tuples of the initial relation have here been (arbitrarily) removed. At time t_2, some values have been (arbitrarily) removed from the domains of the variables involved in c_{xyz}, making c_{xyz} no longer tailored. For example, $(b, a, a) \in \text{rel}^{t_2}(c_{xyz})$ but $(b, a, a) \notin \text{dom}^{t_2}(x) \times \text{dom}^{t_2}(y) \times \text{dom}^{t_2}(z)$.

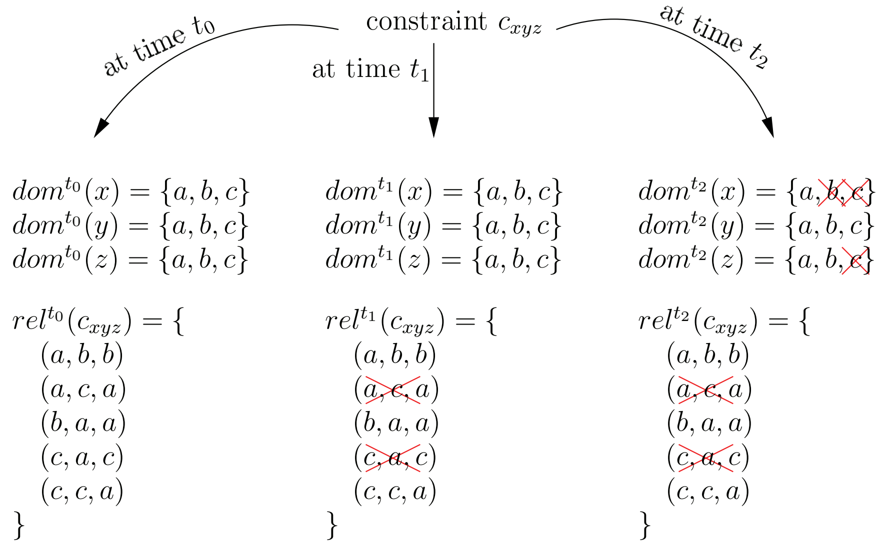


Figure 1.1. *Three (arbitrary) successive states of a constraint c_{xyz}*

The initial relation of c is denoted by $\text{rel}^{\text{init}}(c)$; this is the relation defined at the time origin t_0. As for domains, when we are concerned with constraints in more than one constraint network, e.g. when analyzing the dynamic behavior of an algorithm, we write $\text{rel}^P(c)$ to denote the relation of c in constraint network P. When the context is unambiguous, we simply use $\text{rel}(c)$.

Although we have defined a constraint in terms of an associated relation, this imposes no restriction on the practical prescription of constraints. In practice, a constraint may be defined either intensionally or extensionally.

DEFINITION 1.10.– *[Intensional Constraint] A constraint c is* intensional*, or defined* in intension*, iff* $\text{rel}(c)$ *is implicitly described by a predicate*[2]*, i.e. by a characteristic function that is defined from* $\prod_{x \in \text{scp}(c)} \text{dom}^{\text{init}}(x)$ *to* $\{false, true\}$ *and based on a Boolean expression or formula.*

Examples of Boolean expressions are $x \neq y$ and $|x * y| < |z|$. Clearly, the semantics of constraints intensionally defined by Boolean expressions is immediately understood. We usually refer to an intensional constraint c as $c : expr$ where $expr$ is the predicate expression of c (also denoted by $expr[c]$).

DEFINITION 1.11.– *[Extensional Constraint] A constraint c is* extensional*, or defined* in extension*, iff* $\text{rel}(c)$ *is explicitly described, either positively by listing the tuples allowed by c or negatively by listing the tuples disallowed by c.*

For an extensional constraint c, we use $table[c]$ and $\overline{table}[c]$ to denote the set of tuples allowed and disallowed by c, respectively. Of course, we have $table[c] = \text{rel}(c)$ and $\overline{table}[c] = \prod_{x \in \text{scp}(c)} \text{dom}^{\text{init}}(x) \setminus \text{rel}(c)$. The use of these special terms shows clearly that we are dealing with extensional constraints.

EXAMPLE.– Consider a ternary constraint c_{xyz}. Imagine that this constraint means that the values which can be assigned simultaneously to x, y and z must all be different. The constraint c_{xyz} can be defined in intension by using $x \neq y \wedge x \neq z \wedge y \neq z$ as a predicate expression, denoted by $c_{xyz} : x \neq y \wedge x \neq z \wedge y \neq z$. Note that this representation remains stable, irrespective of the initial domains of variables in $\text{scp}(c_{xyz})$. If $\text{dom}^{\text{init}}(x) \times \text{dom}^{\text{init}}(y) \times \text{dom}^{\text{init}}(z) = \{0,1,2\} \times \{0,1,2\} \times \{0,1,2\}$, then c_{xyz} can be represented in extension by one of the two following sets:

$$table[c_{xyz}] = \left\{ \begin{array}{c} (0,1,2), \\ (0,2,1), \\ (1,0,2), \\ (1,2,0) \\ (2,0,1), \\ (2,1,0) \end{array} \right\} \qquad \overline{table}[c_{xyz}] = \left\{ \begin{array}{c} (0,0,0), \\ (0,0,1), \\ (0,0,2), \\ \ldots \\ (2,2,1), \\ (2,2,2) \end{array} \right\}$$

The number of allowed tuples is 6, whereas the number of disallowed tuples is 21. For simplicity and for space efficiency, it is better in this case to employ a representation of allowed tuples. If we generalize the ternary constraint c_{xyz} to an r-ary constraint c such that the initial domain of any involved variable is $\{0, 1, \ldots, r-1\}$ while keeping the same semantics, the number of allowed and disallowed tuples become $r!$ and $r^r - r!$, respectively. It is then essential to represent such a constraint in

2. Note that an intensional constraint cannot always easily be defined by a Boolean formula, because it sometimes corresponds to use of a computer function.

intension, and even better, by a so-called global constraint whose meaning is implicit. Actually, the constraint introduced in our example is (an instance of) the well-known global constraint (pattern) allDifferent.

DEFINITION 1.12.– *[Global Constraint] A* global constraint *is a constraint pattern that captures precise relational semantics and can be applied over an arbitrary number of variables.*

For example, the semantics of allDifferent is that every variable must take a different value. When the allDifferent constraint pattern is applied to three variables x, y and z, we obtain a constraint denoted by c_{xyz} : allDifferent(x, y, z). Clearly, the allDifferent constraint pattern can be applied to any number of variables. For more information about global constraints, see e.g. [HOE 06, BEL 08].

x	y
a	b
a	c
b	a
b	b
c	c

(a) table of c_{xy}

x \ y	a	b	c
a	0	1	1
b	1	1	0
c	0	0	1

(b) $(0, 1)$-matrix of c_{xy}

Figure 1.2. *Extensional representation of a binary constraint c_{xy} by a table and a $(0, 1)$-matrix*

An alternative representation for extensional constraints is to use multi-dimensional Boolean arrays, also called $(0, 1)$-matrices when constraints are binary. For example, assume that x and y are two variables such that $\mathrm{dom}(x) = \mathrm{dom}(y) = \{a, b, c\}$, and c_{xy} is a binary constraint defined in extension by the table[3] given in Figure 1.2(a). The constraint c_{xy} can equivalently be represented by the $(0, 1)$-matrix given in Figure 1.2(b). An entry of 0 (resp. 1) means that the tuple composed of the value labeling the row and the value labeling the column is disallowed (resp. allowed) by the constraint. For example, we find 1 at the intersection of row b and column a, meaning that (b, a) is allowed by c_{xy}. The space complexity of a table representation is $O(tr)$, where t denotes the number of tuples in the table, and r the arity of the constraint[4]. The space complexity of a multi-dimensional array representation is $O(d^r)$, where d denotes the greatest domain size, which shows that arrays can be used

3. Henceforth, tables are presented as a simple enumeration (list) of tuples.

4. Asymptotic notation is presented in Appendix A.2.1.

only for small-arity constraints. In the remainder of the book, we always consider extensional constraints implemented by tables.

As explained above, when constraints are not tailored we have $\text{rel}(c) \not\subseteq \prod_{x \in \text{scp}(c)} \text{dom}(x)$. For example, consider a binary intensional constraint $c_{xy} : x = y$ such that $\text{dom}^{\text{init}}(x) = \text{dom}^{\text{init}}(y) = \{0, 1, \ldots, 9\}$. We have $\text{rel}(c_{xy}) = \{(i, j) \in \text{dom}^{\text{init}}(x) \times \text{dom}^{\text{init}}(y) \mid i = j\}$. When the membership of domains is changed, we can implicitly update the relation associated with c_{xy}, e.g. as in [BAC 02a], so that $\text{rel}(c_{xy}) = \{(i, j) \in \text{dom}(x) \times \text{dom}(y) \mid i = j\}$, and constraints always remain tailored. However, it may not be practical to update a constraint relation represented in extension; in our example, an extensional representation of c_{xy} is $table[c_{xy}] = \text{rel}^{\text{init}}(c_{xy}) = \{(0, 0), \ldots, (9, 9)\}$. If 0 and 1 are removed from $\text{dom}(x)$, then in principle, (the table associated with) the relation of c_{xy} can be reduced to $\text{rel}(c_{xy}) = \{(2, 2), \ldots, (9, 9)\}$. In practice, updating $table[c_{xy}]$ may be expensive and not very helpful, and implicitly considering such an update may be unsafe in the development and/or complexity analysis of some algorithms. Therefore, unless explicitly mentioned, constraint relations will be considered as invariant, i.e. $\text{rel}(c) = \text{rel}^{\text{init}}(c)$ for all constraints c.

The distinction between what is allowed (i.e. what can be accepted by a constraint) and what is valid (i.e. what can be built from the variable domains of a constraint) is important for understanding the dynamic aspect of some algorithms.

Let $\tau = (a_1, \ldots, a_r)$ be an r-tuple of values of a (totally ordered) set of r variables $X = \{x_1, \ldots, x_r\}$. The value a_i will be denoted by $\tau[x_i]$. By extension, for any subset $X' \subseteq X$, the restriction of τ to the variables in X' will be denoted by $\tau[X']$. For example, let $X = \{w, x, y, z\}$ and $\tau = (a, b, b, c)$. We have $\tau[w] = a$, $\tau[x] = b$, $\ldots$, and $\tau[\{w, z\}] = (a, c)$. A valid tuple for a constraint is a tuple containing a valid value for every variable in the scope of the constraint.

DEFINITION 1.13.– *[Valid Tuple] Let c be an r-ary constraint. An r-tuple τ is* valid *on c iff $\forall x \in \text{scp}(c)$, $\tau[x] \in \text{dom}(x)$. The set of valid tuples on c is* $\text{val}(c) = \prod_{x \in \text{scp}(c)} \text{dom}(x)$.

By definition of variables, we always have $\text{val}(c) \subseteq \prod_{x \in \text{scp}(c)} \text{dom}^{\text{init}}(x)$. Moreover, when c is tailored, we have $\text{rel}(c) \subseteq \text{val}(c)$. Recall that a tuple τ is allowed by a constraint c iff $\tau \in \text{rel}(c)$. Supports and conflicts are defined as follows.

DEFINITION 1.14.– *[Support and Conflict] Let c be an r-ary constraint. An r-tuple τ is a* support *(resp. a* conflict*) on c iff τ is a valid tuple on c which is allowed (resp. disallowed) by c.*

If τ is a support (resp. a conflict) on a constraint c involving a variable x and such that $\tau[x] = a$, we say that τ is a support (resp. a conflict) for (x, a) on c; we also say

that (x, a) is supported (resp. not supported) by c. When (a, b) is a support on a binary constraint c_{xy}, we sometimes say that (x, a) *supports* (y, b) on c_{xy}, and symmetrically that (y, b) supports (x, a) on c_{xy}.

NOTATION 1.15.– *Let c be a constraint.*

– *The set of supports on c is* $\text{sup}(c) = \text{val}(c) \cap \text{rel}(c)$.

– *The set of conflicts on c is* $\text{con}(c) = \text{val}(c) \setminus \text{sup}(c)$.

For a tailored constraint c, we have $\text{sup}(c) = \text{rel}(c)$ since $\text{rel}(c) \subseteq \text{val}(c)$. Determining if a tuple is allowed is called a *constraint check*, and determining if a tuple is valid is called a *validity check*. We often need to make such checks when looking for supports; search of supports represents a basic operation in constraint reasoning. Figure 1.3 summarizes the different sets introduced so far; Figure 1.4 provides a detailed example.

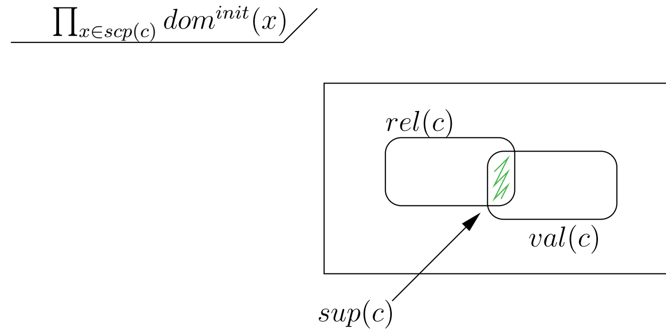


Figure 1.3. *A constraint c whose "universe" is* $\prod_{x \in \text{scp}(c)} \text{dom}^{\text{init}}(x)$. *The set of tuples allowed by c is* $\text{rel}(c)$. *The set of valid tuples on c is* $\text{val}(c)$. *The set of supports on c is* $\text{sup}(c) = \text{rel}(c) \cap \text{val}(c)$

The following notation will be useful in situations where we need to deal with tuples that involve a particular value.

NOTATION 1.16.– *Let c be a constraint,* $x \in \text{scp}(c)$ *and* $a \in \text{dom}(x)$.

– *The set of valid tuples for* (x, a) *on c is* $\text{val}(c)_{x=a} = \{\tau \in \text{val}(c) \mid \tau[x] = a\}$.

– *The set of supports for* (x, a) *on c is* $\text{sup}(c)_{x=a} = \text{val}(c)_{x=a} \cap \text{rel}(c)$.

– *The set of conflicts for* (x, a) *on c is* $\text{con}(c)_{x=a} = \text{val}(c)_{x=a} \setminus \text{sup}(c)$.

– *The set of* strict supports *for* (x, a) *on c is* $\text{sup}(c)\downarrow_{x=a} = \{\tau[\text{scp}(c) \setminus \{x\}] \mid \tau \in \text{sup}(c)_{x=a}\}$.

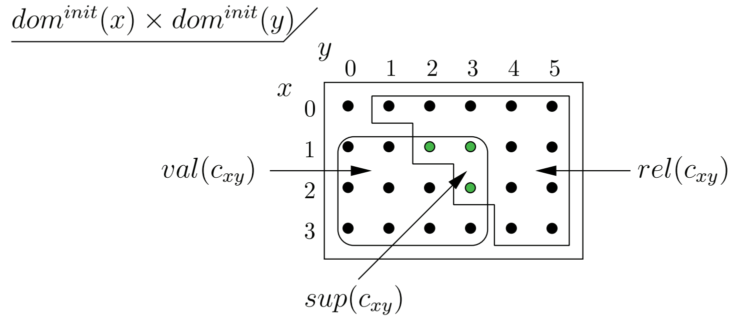


Figure 1.4. *A constraint $c_{xy} : x < y$ whose "universe" is $\mathrm{dom}^{\mathrm{init}}(x) \times \mathrm{dom}^{\mathrm{init}}(y) = \{0, \ldots, 3\} \times \{0, \ldots, 5\}$. The set of tuples allowed by c_{xy} is $\mathrm{rel}(c_{xy}) = \{(i, j) \in \mathrm{dom}^{\mathrm{init}}(x) \times \mathrm{dom}^{\mathrm{init}}(y) \mid i < j\}$. When $\mathrm{dom}(x) = \{1, 2, 3\}$ and $\mathrm{dom}(y) = \{0, 1, 2, 3\}$, the set of valid tuples on c_{xy} is $\mathrm{val}(c_{xy}) = \{1, 2, 3\} \times \{0, 1, 2, 3\}$. The set of supports on c_{xy} is $\sup(c_{xy}) = \mathrm{rel}(c_{xy}) \cap \mathrm{val}(c_{xy}) = \{(1, 2), (1, 3), (2, 3)\}$*

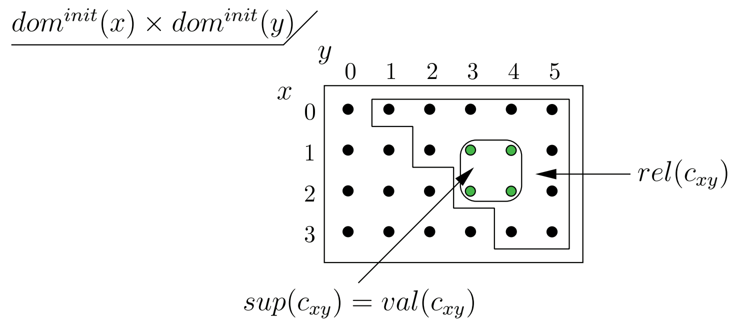


Figure 1.5. *The constraint from Figure 1.4 in a different state, since we now have $\mathrm{dom}(x) = \{1, 2\}$ and $\mathrm{dom}(y) = \{3, 4\}$. Here, we have $\sup(c_{xy}) = \mathrm{val}(c_{xy}) = \{1, 2\} \times \{3, 4\}$. Hence, c_{xy} is entailed: we have a guarantee that $x < y$*

When we have $\sup(c)_{x=a} \neq \emptyset$, we say that c (currently) supports (x, a). Note here that a strict support for a value (x, a) on a constraint c is a tuple composed of $|\operatorname{scp}(c)| - 1$ values, whereas a "classical" support contains $|\operatorname{scp}(c)|$ values. We need strict supports to define some properties later. For example, if c_{xyz} is such that $\sup(c_{xyz}) = \{(a, b, a), (a, b, c), (b, a, b), (c, c, b)\}$, then $\sup(c_{xyz})_{x=a} = \{(a, b, a), (a, b, c)\}$, and $\sup(c_{xyz})\downarrow_{x=a} = \{(b, a), (b, c)\}$.

We can now introduce constraint *tightness* (and *looseness*), which is an important feature. The greater the tightness of a constraint, the more difficult it is to satisfy the constraint.

DEFINITION 1.17.– *[Constraint Tightness and Looseness] Let* c *be a constraint.*

– *The* looseness *of* c *is equal to the ratio*

$$\frac{|\operatorname{rel}^{\text{init}}(c)|}{|\prod_{x\in\operatorname{scp}(c)}\operatorname{dom}^{\text{init}}(x)|}.$$

– *The* tightness *of* c *is equal to the ratio*

$$\frac{|\prod_{x\in\operatorname{scp}(c)}\operatorname{dom}^{\text{init}}(x)\setminus\operatorname{rel}^{\text{init}}(c)|}{|\prod_{x\in\operatorname{scp}(c)}\operatorname{dom}^{\text{init}}(x)|}.$$

Looseness and tightness above are defined from initial domains and relation; this corresponds to the classical usage. Sometimes it is useful to compute tightness or looseness from current domains and relation. The *current* constraint tightness (resp. looseness) of a constraint c is the ratio $|\operatorname{con}(c)|/|\operatorname{val}(c)|$ (resp. $|\operatorname{sup}(c)|/|\operatorname{val}(c)|$). Current constraint tightness corresponds to the ratio "number of conflicts on c over number of valid tuples on c". For example, the constraint tightness of the constraint c_{xy} depicted in Figure 1.4 is $\frac{10}{24}$, assuming that $\operatorname{rel}(c_{xy}) = \operatorname{rel}^{\text{init}}(c_{xy})$, and its current constraint tightness is $\frac{9}{12}$.

Universal and *empty* constraints correspond to extreme values of $\operatorname{rel}(c)$. A universal constraint can be safely ignored (but may be introduced for special purposes), whereas an empty constraint can never be satisfied.

DEFINITION 1.18.– *[Universal and Empty Constraints] Let* c *be a constraint.*

– c *is* universal *iff* $\operatorname{rel}^{\text{init}}(c) = \prod_{x\in\operatorname{scp}(c)}\operatorname{dom}^{\text{init}}(x)$.

– c *is* empty *iff* $\operatorname{rel}^{\text{init}}(c) = \emptyset$.

After domains have been reduced, constraints sometimes seem to be universal or empty; they are said to be *entailed* or *disentailed*:

DEFINITION 1.19.– *[Entailed and Disentailed Constraints] Let* c *be a constraint.*

– c *is* entailed *iff* $\operatorname{sup}(c) = \operatorname{val}(c)$.

– c *is* disentailed *iff* $\operatorname{sup}(c) = \emptyset$.

As long as no value is restored to any domain, an entailed constraint is guaranteed to be satisfied (provided that at least one value remains in each domain). Similarly, a disentailed constraint is guaranteed to be unsatisfied. An illustration of an entailed constraint is given in Figure 1.5.