# Constraint Satisfaction Problems

Marco Kuhlmann Guido Tack

14th July 2005

The purpose of this document is to set up a simple formal framework for constraint programming. We hope that this gives you a good idea of the fundamental concepts involved, and helps us motivate various components of the processing architecture we will develop during the course of this lecture. Note that none of the concepts defined here is original; everything has been taken from existing literature.

## 1 Basic definitions

**Finite domain** A *finite domain* is a finite set of values. Examples for finite domains are: the set of Boolean values $\{\top, \bot\}$; finite subsets of the natural numbers (such as $\{0, 1, 2, 3\}$); the set of subsets of a given finite set of objects (such as the set of subsets of the set $\{0, 1, 2, 3\}$).

**Finite-domain variable** A *finite-domain variable* is a variable that can range over values from a finite domain. We will assume that these variables are taken from a (countably infinite) set of typed variables $\mathfrak{X}$, and that each variable $x : \tau$ is associated with a finite domain of type $\tau$ via a mapping $\mathsf{dom} \in \mathfrak{X} \to \mathcal{P}_{fin}(\tau)$. We will usually leave this mapping implicit and write $x : D$ to signify that $x$ is a finite-domain variable (of some type $\tau$) for which $\mathsf{dom}(x) = D$. For example, we write $x : [0, 3]$ to say that $x$ is a finite-domain variable (of type $\mathbf{N}$) for which $\mathsf{dom}(x) = [0, 3]$ (the interval containing the natural numbers from 0 to 3).

**Valuations and assignments** *Valuations* map variables to values of the right type. *Assignments* are valuations that respect the domains of the variables.

**Definition 1 (Valuation; assignment)** Let $X$ be a finite set of finite-domain variables. A *valuation* for $X$ is a function $v$ from $X$ to values such that for each $(x : \tau) \in X$, $vx \in \tau$. An *assignment* for $X$ is a valuation $\alpha$ for $X$ such that for all $(x : \tau) \in X$, $\alpha x \in \mathsf{dom}(x)$. The value $\alpha x$ is called the value *assigned* to $x$. □

Note that assignments for $X$ can be seen as a set of ordered pairs, where the first component is a finite-domain variable, and the second component is a value from the domain of that variable. We use the notations $\mathsf{val}(X)$ to refer to the set of all valuations and $\mathsf{ass}(X)$ to refer to the set of all assignments for $X$. Also, note that while $\mathsf{val}(X)$ may be infinite (depending on the cardinality of the types of the variables in $X$), $\mathsf{ass}(X)$ will always be a finite set (as long as $X$ itself is finite).

**Constraints** Intuitively, a constraint restricts the possible values for (a set of) variables. We can make this intuition precise by viewing constraints as sets of valuations: each constraint contains those valuations that it 'allows'.

**Definition 2 (Constraint)** Let $X$ be a set of finite-domain variables. A *constraint* (for $X$) is a set of valuations for $X$. □

Different constraints may differ in the sets of variables they constrain. For example, given variables $\{x, y, z\}$, the binary constraint $x < y$ only constrains the two variables $x$ and $y$, not the variable $z$. To formalise this idea, we introduce the concept of the *scope* of a constraint. Intuitively, a constraint with scope $S$ may choose to map the variables in $S$ to values from a subset of the types associated to these variables, but must map all variables not in $S$ to all possible values of their type.

**Definition 3 (Scope)** Let $X$ be a set of finite-domain variables, and $S \subseteq X$. A *constraint with scope S* is a constraint $C$ that can be defined according to the scheme

$$C := \{\, v \in \mathsf{val}(X) \mid \textbf{if } (x : \tau) \in S \textbf{ then } vx \in (\tau' \subseteq \tau) \textbf{ else } vx \in \tau \,\}.$$ □

**Constraint satisfaction problems** A constraint satisfaction problem specifies a finite set of finite-domain variables, a mapping from variables to domains (which we, as we said above, will leave implicit), and a set of constraints on these variables:

**Definition 4 (CSP)** A *constraint satisfaction problem* (CSP) $\langle X ; \mathfrak{C} \rangle$ is characterised by a finite set $X$ of finite-domain variables with associated domains and a finite set $\mathfrak{C}$ of constraints for $X$. The set $X$ is called the set of *problem variables* of the CSP. □

**Solutions** The intuitive task associated to a CSP is this: 'Find assignments for the problem variables such that all constraints hold.' Since we defined constraints as sets of valid valuations, a constraint $C$ holds for an assignment $\alpha$ if $\alpha \in C$.

**Definition 5 (Solution to a CSP)** Let $P := \langle X ; \mathfrak{C} \rangle$ be a CSP, and let $C$ be a constraint in $\mathfrak{C}$. An assignment $\alpha$ *satisfies* $C$ if $\alpha \in C$; it is a *solution to* $P$ if $\alpha \in \bigcap_{C \in \mathfrak{C}}$. □

In the context of a CSP $P$ with problem variables $X$, we will use the notation $\mathsf{sat}(C)$ to denote the set of assignments (for $X$) satisfying a constraint $C$, and the notation $\mathsf{sol}(P)$ to denote the set of solutions to $P$. A CSP that does not have any solutions is called *inconsistent*.

Note that the definition of a CSP that we use here takes an *extensional* view on constraints: they are specified as sets of assignments. This view is good because it allows us to keep the definitions of the relevant concepts simple—but the extensional view is very bad as a representation technique, since even rather simple constraints may have infinite extensions. If we want to write down a constraint on paper or in a computer program, we must adopt an *intensional* view, and use some kind of constraint language to specify constraints.

## 2 Domain reduction

**Determined sets** Here is an alternative view on solutions to a CSP: A finite-domain variable $x : D$ is said to be *determined*, if $D$ is a singleton. For sets of determined variables, we usually write $x : d$ instead of $x : \{d\}$. Note that a CSP in which all variables are determined corresponds to an assignment $\alpha$ as follows:

$$\langle \{x_1 : \{d_1\}, \ldots, x_n : \{d_n\}\} ; \mathfrak{C} \rangle \;\leftrightsquigarrow\; \alpha = \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\} .$$

Thus, to find a solution to a CSP, we can try to reduce the domains of the problem variables such that they become determined, and verify that all constraints are satisfied. This process is known as *domain reduction*.

**Definition 6 (Reduction)** Consider two CSPs $P$ and $P'$ that are equal up to the associated domains—that is $P$ and $P'$ share the same set of problem variables $X$ and the same set of constraints, but may have different domain mappings $\mathsf{dom}$ and $\mathsf{dom}'$. We then say that $P$ is a *reduction* of $P'$, written $P \prec P'$, if $\mathsf{dom}(x) \subseteq \mathsf{dom}'(x)$ for all variables $x \in X$, and $\mathsf{dom}(x) \subset \mathsf{dom}'(x)$ for at least one variable $x \in X$. □

**Generate and test** There is a very dumb (but straightforward) algorithm that uses domain reduction to solve a CSP:

- Take any problem variable $x : D$ that is not yet determined, partition its domain into disjoint sets $D_1, \ldots, D_k$, choose an $1 \leq i \leq k$, and create a reduction of the old problem by replacing $x : D$ by $x : D_i$.
- Proceed recursively by reducing the new problem until all of the problem variables are determined. Finally, test whether the induced assignment satisfies all the constraints.

If the final test succeeds for at least one set of domain choices, we have found a solution for the original problem; otherwise, the original problem was inconsistent.

The outlined algorithm is called *generate-and-test*. It is dumb because it might do a lot of unnecessary guessing: inconsistencies in the problem are found only after all variables are reduced. Of course, in the worst case (where the CSP under consideration is NP-complete), we cannot even hope to do any better than this. But it would be good if we could adapt our algorithm to detect inconsistencies as early as possible—to do better when we actually can do better. This requires two changes: First, we need to eliminate inconsistent values before finding an assignment—this is what we will call a *propagation* rule. Second, we need to use a control strategy that ensures that guessing is done only as a last resort. In the remainder of this section, we will present a system of inference rules for the revised algorithm.

**Failure detection** The first rule that we will use detects obvious inconsistencies: a problem is obviously inconsistent if one of the involved domains has been reduced to the empty set. In this case, we cannot build *any* assignment, save one that satisfies all the constraints.

$$\frac{\langle X \cup \{x : \varnothing\} \,;\, \mathfrak{C}\rangle}{\textbf{fail}} \tag{1}$$

**Branching** The second rule captures the 'generate' aspect of the generate-and-test algorithm: To reduce a CSP $P$ that is not yet determined, we can take any variable $x : D$ from $P$, partition its domain into disjoint sets $D_1, \ldots, D_k$, guess an $1 \leq i \leq k$, and continue the reduction with the problem obtained from $P$ by replacing $D$ by $D_i$. This rule is known as *branching*:

$$\frac{\langle X \cup \{x : D\} \,;\, \mathfrak{C}\rangle \qquad |D| > 1 \qquad D = D_1 \uplus \cdots \uplus D_k}{\langle X \cup \{x : D_1\} \,;\, \mathfrak{C}\rangle \mid \cdots \mid \langle X \cup \{x : D_k\} \,;\, \mathfrak{C}\rangle} \tag{2}$$

**Constraint propagation** Finally, we need a rule to eliminate inconsistent values for the variables. From an extensional perspective, what we could do to test the consistency of a candidate assignment is to check whether it appears in the intersection of all the constraints in the problem. From an intensional perspective, this is not generally possible anymore, since (a) the CSP under consideration may still allow an exponential number of possible assignments and (b) it may be very hard to compute the 'intersection' of a set of intensional constraints. We make the compromise that we only consider one constraint at a time, and test the *local consistency* of *potential values* of the problem variables: For each constraint $C$, problem variable $x : D$ and potential value $d \in D$ for that variable, we check whether there are any assignments

that satisfy $C$ (rather than all the constraints in the problem) and map $x$ to $d$. If that test fails, we can eliminate $d$ as a potential value for $x$. A rule of this type is called a *constraint propagation rule*:

$$\frac{\langle Y = X \cup \{x : D\} \,;\, \mathfrak{C} \cup \{C\}\rangle \qquad d \in D \qquad \mathsf{sat}(C) \cap \{\, \alpha \in \mathsf{ass}(Y) \mid \alpha x = d \,\} = \varnothing}{\langle X \cup \{x : D - \{d\}\} \,;\, \mathfrak{C} \cup \{C\}\rangle} \tag{3}$$

Notice that, if we applied this rule only to assignments, it would—together with the rule for failure—have the same effect as the 'testing' part of the generate-and-test algorithm: if **fail** cannot be derived, the current problem defines a solution.

Of course, in order to perform the test in the third premise of the rule in its completeness, we might still be required to generate an exponential number of assignments, so that any implementation of the rule might need exponential running time. This is something that we do not want to do. It is important to point out though, that we do not need to be too pedantic about the test as long as the problem is not yet determined: if we can eliminate *any* value at all, we will still be better off than with generate-and-test. Therefore, it is no problem to only approximate the test, and exclude only those values that we can exclude without spending too much time. (The implementations of the propagation rules in a constraint system like GeCoDE typically have polynomial running time.) What we *should* be efficient at is performing the test in the case where all of the problem variables are determined, and the set $\mathsf{ass}(X)$ is a singleton.

**Control strategy** The complete algorithm now works as follows: In a first step, it exhaustively applies the propagation rule. When no further propagation is possible, we have one of three cases:

- At least one of the domains of the problem variables has been reduced to the empty set. In this case, the algorithm terminates and reports 'failure'.
- All of the domains of the problem variables are singletons. In this case, no further rule can be applied, and the algorithm terminates; the current constraint problem induces an assignment that is a solutionto the original problem.
- At least one of the domains of the problem variables contains more than one value. In this case, the branching rule is applied, and the algorithm recurses on one of the reduced problems.

This (abstract) algorithm is a complete solver for constraint satisfaction problems. In the course of the next lectures, we will instantiate its various components to see how it can be implemented efficiently.