Chapter 2
Constraints

A constraint is something that restricts, limits, or regulates. From this notion of constraint, a new field of computer science has been created: Constraint Programming (CP).

CP has attracted high attention among experts from many areas because of its potential for solving hard real-life problems and because it is based on a strong theoretical foundation. The success of CP derives from the fact that on one hand it allows to model a problem in a simple way and on the other hand it provides efficient problem solving algorithms.

There are a lot of surveys on CP. One of the best and complete works on CP is [108]. To be short, however, we will follow [10]. After giving in Sect. 2.1, a brief history of CP we will first define what a Constraint Satisfaction Problem (CSP) is in Sect. 2.2 and how these problems are solved in Sects. 2.3, 2.4, 2.5, and 2.6. In Sects. 2.8 and 2.7 we will describe some extensions of CP while in Sects. 2.9 and 2.10 we will present some of its applications and limitations.

2.1 Short History

The earliest ideas leading to CP may be found in the Artificial Intelligence (AI) dating back to the 1960s and 1970s. The scene labeling problem [129] where the goal is to recognize the objects in a 3D scene by interpreting lines in the 2D drawings is probably the first CSP that was formalized. The main algorithms developed in those years were related to achieving some form of consistency. Another application for constraints is interactive graphics where Ivan Sutherland's Sketchpad [120], developed in the 1960s, was the pioneering system. Sketchpad and its follower were interactive graphics applications that allowed the user to draw and manipulate constrained geometric figures on the computer's display. These systems contribute to developing local propagation methods and constraint compiling. The main step toward CP was achieved when Gallaire [51] and Jaffar and Lassez [66] noted that

J. Mauro, *Constraints Meet Concurrency*, Atlantis Studies in Computing 5,
DOI: 10.2991/978-94-6239-067-6_2, © Atlantis Press and the authors 2014

logic programming was just a particular kind of CP. The basic idea behind Logic Programming is that the user states what has to be solved instead of how to solve it, which is very close to the idea of constraints. Therefore, the combination of constraints and logic programming is very natural and Constraint Logic Programming (CLP) makes a nice declarative environment for solving problems by means of constraints. However, it does not mean that CP is restricted to CLP. Constraints were integrated to typical imperative languages like c++ [111] or Java [77] as well.

CP has an inner interdisciplinary nature. It combines and exploits ideas from a number of fields including AI, Combinatorial Algorithms, Computational Logic, Discrete Mathematics, Neural Networks, Operations Research, Programming Languages, and Symbolic Computation.

2.2 Constraint Satisfaction Problems

CSPs have been a subject of research in AI for many years. A CSP is defined as:

- a set of variables $X = x_1, \ldots, x_n$
- for each variable x_i a finite set D_i of possible values like naturals, reals or strings. D_i is called domain
- a set of constraints restricting the values that the variables can simultaneously take.

Example 2.1 *Let us show how the famous "send more money" mathematical game published in the July 1924 issue of Strand Magazine by Henry Dudeney can be modeled. This game is an alphametics, i.e., a type of mathematical game consisting of a mathematical equation among unknown numbers, whose digits are represented by letters. To solve the game, we need to associate to every letter in "send more money" a different number from 0 to 9 in a way that the sum of "send" and "more" is equal to "money." We also require that the leading digit of a multidigit number is not zero.*

Since we need to find what numbers are associated to every letter we can model these numbers with variables which domain is the set $\{0, \ldots, 9\}$. *Let be* c_v *the variable associated to the letter* c.

The set of constraint to consider are:

- $s_v e_v n_v d_v + m_v o_v r_v e_v = m_v o_v n_v e_v y_v$
- $c_v \neq c'_v$ *if* $c \neq c'$
- $s_v \neq 0$
- $m_v \neq 0$.

A solution to a CSP is a labeling, i.e., an assignment of a value from its domain to every variable, in such a way that all constraints are satisfied at once. We may want to find:

- just one solution, with no preference
- all solutions

- an optimal, or at least a good solution, given some objective function defined in terms of some or all of the variables.

Solutions to a CSP can be found by searching (systematically) through the possible assignments of values to variables. Search methods divide into two broad classes, those that traverse the space of partial solutions (or partial value assignments), and those that explore the space of complete value assignments (to all variables) stochastically.

2.3 CSP Solving: Systematic Search

From the theoretical point of view, solving CSP is trivial using systematic exploration of the solution space. Even if systematic search methods without additional improvements look very simple and nonefficient, they are important and worth mentioning because they make the foundation of more advanced and efficient algorithms.

The basic constraint satisfaction algorithm that searches the space of complete labelings, is called generate-and-test. The idea is simple: a complete labeling of variables is generated and, consequently, if this labeling satisfies all the constraints then the solution is found, otherwise, another labeling is generated. The generate-and -test algorithm is a weak generic algorithm that is used if everything else failed. Its efficiency is poor because of noninformed generator and late discovery of inconsistencies. Consequently, there are two ways to improve its efficiency:

- the generator of valuations is smart, i.e., it generates the complete valuation in such a way that the conflict found by the test phase is minimized
- the generator is merged with the tester, i.e., the validity of the constraint is tested as soon as its respective variables are instantiated. This method is used by the backtracking approach. Backtracking [87] is a method of solving CSP by incrementally extending a partial solution that specifies consistent values for some of the variables, toward a complete solution, by repeatedly choosing a value for another variable consistent with the values in the current partial solution. Clearly, whenever a partial instantiation violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of all variable domains. Consequently, backtracking is strictly better than generate-and-test. However, its running complexity for most nontrivial problems is still NP-hard.

There are three major drawbacks of the standard backtracking:

1. thrashing, i.e., repeated failure due to the same reason
2. redundant work, i.e., conflicting values of variables are not remembered
3. late detection of the conflict, i.e., conflict is not detected before it really occurs.

We will now present some of the improvements of backtracking studied in the literature.

2.4 CSP Solving: Consistency Techniques

One alternative approach for solving CSP is based on removing inconsistent values from variables' domains till the solution has been founded. These methods are called consistency techniques. There are several consistency techniques [76, 83] but most of them are not complete, i.e., they can not be used alone to solve a CSP completely. The names of basic consistency techniques are derived from the graph notions. The CSP is usually represented as a constraint graph or hypergraph (sometimes called constraint network) where nodes correspond to variables and edges/hyperedges are labeled by constraints.

The simplest consistency technique is referred to as a node consistency. It removes values from variable domains that are inconsistent with unary constraints on respective variables. The most widely used consistency technique is called arc consistency. This technique removes values from variables domains that are inconsistent with binary constraints. There exist several arc consistency algorithms starting from AC-1 based on repeated revisions of arcs till a consistent state is reached or some domain become empty. The most popular among them are AC-3 and AC-4.

Even more inconsistent values can be removed by path consistency techniques. Path consistency is a property similar to arc consistency, but considers pairs of variables instead of only one. A pair of variables is path-consistent with a third variable if each consistent evaluation of the pair can be extended to the other variable in such a way that all binary constraints are satisfied. There exist several path consistency algorithms like PC-1 and PC-2 but, compared to algorithms for arc consistency, they need an extensive representation of constraints that is memory consuming.

All above-mentioned consistency techniques are covered by a general notion of k-consistency [39] and strong k-consistency. A constraint graph is k-consistent if for every system of values for $k-1$ variables satisfying all the constraints among these variables, there exist a value for arbitrary kth variable such that the constraints among all k variables are satisfied. A constraint graph is strongly k-consistent if it is j-consistent for all $j \leq k$. We have that:

- node consistency is equivalent to strong 1-consistency
- arc consistency is equivalent to strong 2-consistency
- path consistency is equivalent to strong 3-consistency.

Algorithms exist for making a constraint graph strongly k-consistent for $k > 2$ but in practice they are rarely used because of efficiency issues.

Although these algorithms remove more inconsistent values than any arc-consistency algorithm they do not eliminate the need for search in general. Restricted forms of these algorithms removing a similar amount of inconsistencies with a greater efficiency have been proposed. For example, directional arc consistency revises each arc only once, requires less computation than AC-3 and less space than AC-4 but is still able to achieve full arc consistency in some problems. It is also possible to weaken the path consistency in a similar way.

2.5 Constraint Propagation

Both systematic search and consistency techniques can be used alone to completely solve the CSP but this is rarely done in practice. A combination of both approaches is more commonly used. To avoid some problems of backtracking like thrashing or redundant work, look back schemes were proposed. Backjumping [52], for instance, is a method to avoid thrashing. The control of backjumping is exactly the same as backtracking except when backtracking takes place. Both algorithms pick one variable at a time and look for a value for this variable making sure that the new assignment is compatible with values committed so far. However, if backjumping finds an inconsistency, it analyses the situation in order to identify the source of inconsistency. It uses the violated constraints as guidance to find out the conflicting variable. If all the values in the domain are explored then the backjumping algorithm backtracks to the most recent conflicting variable. This is a main difference from the backtracking algorithm that backtracks to the immediate past variable.

Another look back schema called backchecking [63] avoids redundant work. Backchecking and its evolution backmarking are useful algorithms for reducing the number of compatibility checks. If the algorithm finds that some label Y/b is incompatible with any recent label X/a then it remembers this incompatibility. As long as X/a is still committed, the Y/b will not be considered again. Backmarking is an improvement over backchecking since it reduces the number of compatibility checks by remembering for every label the incompatible recent labels and avoids repeating compatibility checks which have already been performed.

All look back schemes share the disadvantage of late detection of the conflict. Indeed, they solve the inconsistency when it occurs but they do not prevent the inconsistency to occur. For this reason, look ahead schemes were proposed. For instance forward checking, the simplest example of look ahead strategy, performs arc-consistency between pairs of a non instantiated variable and an instantiated one removing temporarily the values that the noninstantiated variable can not assume. It maintains the invariance that for every unlabeled variable there exists at least one value in its domain that is compatible with the values of instantiated/labeled variables. Even though forward checking does more work than backtracking when each assignment is added to the current partial solution, it is almost always a better choice than chronological backtracking.

Further, future inconsistencies are removed by the partial look ahead method. While forward checking performs only the checks of constraints between the current variable and the not defined variables, the partial look ahead extends this consistency checking even to variables that have not direct connection with labeled variables, using directional arc consistency. The approach that uses full arc-consistency after each labeling step is called (full) look ahead.

2.6 Stochastic and Heuristic Algorithms

In the last few years, greedy local search strategies have become popular. These algorithms alter incrementally inconsistent value assignments to all the variables. They use a "repair" or "hill-climbing" metaphor to move toward more and more complete solutions. To avoid getting stuck at "local minimum," they are equipped with various heuristics to randomize the search but this stochastic nature generally avoids the guarantee of "completeness" provided by the systematic search methods.

Hill-climbing is probably the most famous algorithm of local search [87]. It starts from a randomly generated labeling of variables and, at each step, it changes a value of some variable in such a way that the resulting labeling satisfies more constraints. If a strict local minimum is reached then the algorithm restarts using other randomly generated states. The algorithm stops as soon as a global minimum has been found, i.e., all constraints are satisfied, or some resource (e.g., time, memory) is exhausted.

To avoid exploring the whole neighborhood of a state, the min-conflicts heuristic was proposed [92]. This heuristic chooses randomly any conflicting variable, i.e., the variable that is involved in any unsatisfied constraint, and then picks a value which minimizes the number of violated constraints (break ties randomly). If no such value exists, it picks randomly one value that does not increase the number of violated constraints. Because the pure min-conflicts algorithm cannot go beyond a local-minimum, some noise strategies were introduced. Among them worth presenting is the random walk heuristics [112] that for a given conflicting variable picks randomly a value with probability p and applies the min-conflicts heuristic with probability $1 - p$. The hill-climbing algorithm using the random-walk heuristic is also known as Steepest-Descent-Random-Walk.

Tabu search is another method to avoid cycling and getting trapped in local minimum [54, 55]. It is based on the notion of tabu list, a special short-term memory that maintains a selective history, composed of previously encountered configurations or, more generally, pertinent attributes of such configurations. A simple tabu search strategy consists in preventing configurations of tabu list from being recognized for the next k iterations. Such a strategy prevents algorithm from being trapped in short-term cycling and allows the search process to go beyond local optima.

2.7 Constraint Optimization

In many real-life applications, we do not want to find any solution but a good solution. The quality of solutions is usually measured by an application dependent function called objective function. The goal is to find a solution that satisfies all the constraints and minimize or maximize the objective function. Such problems are referred to as Constraint Optimization Problems (COP). A COP consists of a standard CSP and an optimization function that maps every solution to a numerical value.

The most used algorithm for finding optimal solutions is called branch and bound [81]. It needs a heuristic function mapping a partial labeling to a numerical value that represents an under estimate (in case of minimization) of the objective function for the best complete labeling obtained from the partial labeling. The branch and bound algorithm searches for solutions in a depth first manner and behaves like chronological backtracking except that, as soon as a value is assigned to the variable, the heuristic function is applied to the partial labeling. If case the under estimate obtained exceeds the bound of the best solution encountered so far the subtree under the partial labeling is pruned.

The efficiency of branch and bound is determined by two factors: the quality of the heuristic function and whether a good bound is found early. Observations of real-life problems show improving a good solution are usually the most computationally expensive part of the solving process.

Note that the branch and bound algorithm can be used to find suboptimal solutions too. For instance, the algorithm can compute all the solution and return a solution that reaches an acceptable bound even though this solution is not proved to be optimal.

2.8 Constraint Programming Extensions

Problems in which it is not possible to satisfy all the constraints are called over-constrained. Several approaches were proposed to handle these problems. Here we present two of these approaches, namely Partial Constraint Satisfaction and Constraint Hierarchies.

Partial Constraint Satisfaction [40] involves finding a solution to a CSP problem where some constraints are "weaken" to permit additional acceptable value combinations. Formally, a Partial Constraint Satisfaction problem is defined as a standard CSP with some evaluation function that maps every labeling of variables to a numerical value. The goal is to find a labeling with the best value of the evaluation function. A Partial Constraint Satisfaction problem looks like a COP with the difference that the satisfaction of all the constraint is not required. In fact, the global satisfaction is described by the evaluation function and constraints are used as a guide to find an optimal value of the evaluation function. Many standard algorithms like backjumping, backmarking, arc-consistency, forward checking, and branch and bound were customized to work with Partial Constraint Satisfaction problems.

Constraint hierarchies [24] is another approach of handling over-constrained problems. The constraint is weakened explicitly here by specifying its strength or preference. It allows one to specify not only the constraints that are required to hold, but also weaker constraints (usually called soft constraints) . Intuitively, the hierarchy does not permit to the weakest constraints to influence the result at the expense of dissatisfaction of a stronger constraint. Currently, two groups of constraint hierarchy solvers can be identified, namely refining method and local propagation. While the refining methods solve the constraints starting from the strongest level and continuing to

weaker levels, the local propagation algorithms gradually solve constraint hierarchies by repeatedly selecting uniquely satisfiable constraints.

2.9 Applications

CP has been successfully applied to many areas as diverse as DNA structure analysis, time-tabling for hospitals or industry scheduling. It proved to be well adapted for solving real-life problems because many application domains evoke constraint descriptions naturally.

Assignment problems were perhaps the first type of industrial application that were solved with the constraint tools. A typical example is the stand allocation for airports, where aircraft must be parked on the available stand during the stay at airport or counter allocation for departure halls. Another example is berth allocation to ships in the harbor or refinery berth allocation.

Another typical constraint application area is personnel assignment where work rules and regulations impose difficult constraints. The important aspect in these problems is the requirement to balance work among different persons. Systems like Gymnaste were developed for production of rosters for nurses in hospitals, for crew assignment to flights or stuff assignment in railways companies.

Successful applications for finite domain constraint are the once that solve scheduling problems, where, again, constraints express naturally the real-life limitations. Constraint-based software is used for well-activity scheduling, forest treatment scheduling, production scheduling in plastic industry or for planning production of military and business jets. The usage of constraints in Advanced Planning and Scheduling systems is increasing due to current trends of on-demand manufacturing.

Another large area of constraint application is network management and configuration. These problems include planning of cabling of the telecommunication networks in the building or electric power network reconfiguration for maintenance scheduling without disrupting customer services. Another example is optimal placement of base stations in wireless indoor telecommunication networks [47]. There are many other areas that have been tackled using constraints. Recent applications of CP were used in computer graphics, natural language processing, database systems, molecular biology, business applications, electrical engineering, and transport problems.

2.10 Limitations

Since many problems solved by CP are NP-hard problems, the identification of restrictions that make the problem tractable is very important for both the theoretical and the practical points of view. Unfortunately, the efficiency of constraint programs

is still unpredictable and the intuition is usually the most important part of deciding when and how to use constraints. A common problem for CP users is the stability of the constraint model. Even small changes in a program or in the data can lead to a dramatic change in performance. The process of performance debugging for a stable execution over a variety of input data is currently not well understood.

Another problem is choosing the right constraint satisfaction technique for a particular problem. Sometimes fast blind search like chronological backtracking is more efficient than more expensive constraint propagation and vice versa.

A particular problem in many constraint models is the cost optimization. Sometimes, it is very difficult to improve an initial solution, and a small improvement takes much more time than finding the initial solution.

Finally, constraint programs can add constraints dynamically but they do not support the on-line constraint solving required, for instance, in a changing environment. For instance, the possibility of deleting a constraint at runtime has been considered by some extensions like the ones described in [124] but this kind of operation is yet too costly to be performed.



http://www.springer.com/978-94-6239-066-9

Constraints Meet Concurrency

Mauro, J.

2014, XV, 148 p. 13 illus., Hardcover
ISBN: 978-94-6239-066-9
A product of Atlantis Press