**Title of Publication**: Encyclopedia of Cognitive Science.

**Article title**: Constraint Satisfaction.

**Article code**: 26.

**Authors**:
Rina Dechter
Department of Computer and Information Science
University of California, Irvine
Irvine, California, USA 92717
Telephone: 949-824-6556
Fax: 949-824-4056
E-mail: dechter@ics.uci.edu

Francesca Rossi
Dipartimento di Matematica Pura ed Applicata
Università di Padova
Via Belzoni 7, 35131 Padova, Italy
Telephone: +39-049-8275982
Fax: +39-049-8275892
E-mail: frossi@math.unipd.it

# Constraint Satisfaction

**Keywords**: Constraints, constraint satisfaction, constraint propagation, constraint programming.

## Contents

**Article definition**: Constraints are a declarative knowledge representation formalism that allows for a compact and expressive modeling of many real-life problems. Constraint satisfaction and propagation tools, as well as constraint programming languages, are successfully used to model, solve, and reason about many classes of problems, such as design, diagnosis, scheduling, spatio-temporal reasoning, resource allocation, configuration, network optimization, and graphical interfaces.

## 1 Introduction

**Constraint satisfaction problems.** A *constraint satisfaction problem (CSP)* consists of a finite set of *variables*, each associated with a *domain* of values, and a set of *constraints* . Each of the constraint is a relation, defined on some subset of the variables, called its *scope*, denoting their legal combinations of values. As well, constraints can be described by mathematical expressions or by computable procedures.

**Solutions.** A *solution* is an assignment of a value to each variable from its domain such that all the constraints are satisfied. Typical constraint satisfaction problems are to determine whether a solution exists, to find one or all solutions, and to find an optimal solution relative to a given cost function.

**Examples of CSPs.** An example of a constraint satisfaction problem is the well-known $k$-colorability problem. The task is to color, if possible, a given graph with $k$ colors only, such that any two adjacent nodes have different colors. A constraint satisfaction formulation of this problem associates the nodes of the graph with variables, the possible colors are their domains and the not-equal constraints between adjacent nodes are the constraints of the problem.

Another known constraint satisfaction problem concerns satisfiability (SAT), which is the task of finding a truth assignment to propositional variables such that a given set of clauses are satisfied. For example, given the two clauses $(A \vee B \vee \neg C), (\neg A \vee D)$, the assignment of *false* to $A$, *true* to $B$, *false* to C, and *false* to $D$ is a satisfying truth value assignment.

**The constraint graph.** The structure of a constraint problem is usually depicted by a *constraint graph* whose nodes represents the variables, and any two nodes are connected if the corresponding variables participate in the same constraint scope. In the $k$-colorability formulation, the graph to be colored is the constraint graph. In the SAT example above, the constraint graph has $A$ connected with $D$, and $A, B$ and $C$ are connected to each other.

**Applications areas.** Constraint problems have proven successful in modeling mundane cognitive tasks such as vision, language comprehension, default reasoning and abduction, as well as in applications such as scheduling, design, diagnosis, and temporal and spatial reasoning. The reason is that constraints allow for a natural, expressive and declarative formulation of *what* has to be satisfied, without the need to say *how* it has to be satisfied.

**Complexity of constraint-related tasks.** In general, constraint satisfaction tasks (like finding one or all solutions, or the best solution) are *computationally intractable* (*NP-hard*). Intuitively, this mean, that in the worst case all the possible variable instantiations may need to be considered before a solution (or best solution) can be found. However, there are some *tractable classes* of problems that allow for efficient solution algorithms even in the worst-case. Moreover, also for non-tractable classes, many techniques exhibit a good performance in practice in the average case.

**Constraint optimization problems.** Constraint processing tasks include not only the *satisfaction* task, but also constraint *optimization* problems. This occurs when the solutions are not equally preferred. The preferences among solutions can be expressed via an additional cost function (also called an *objective function*), and the task is to find a best-cost solution or a reasonable approximation of it.

**Soft constraints.** The notion of constraint optimization leads a more flexible view of constraints, where each constraint may have a level of importance. In this case, we talk about *soft constraints*, which allow for a faithful modeling of many applications and can accommodate user preferences and uncertainties within the constraint formalism.

**Techniques for solving CSPs.** The techniques for processing constraint problems can be roughly classified into two main categories: *search* and *consistency inference* (or *propagation*). However, such techniques can also be combined, and, in fact, in practice a constraint processing technique usually contains aspects of both categories. Search algorithms traverse the space of partial instantiations, building up a complete instantiation that satisfies all the constraints, or they determine that the problem is *inconsistent*. In contrast, consistency-inference algorithms reason through equivalent problems: at each step they modify the current problem to make it *more explicit* without loosing any information (that is, maintaining the same set of solutions). Search is either systematic and complete, or stochastic and incomplete. Likewise, consistency-inference has complete solution algorithms (e.g., variable-elimination), or incomplete versions, usually called *local consistency* algorithms because they operate on local portions of the constraint graph.

**Constraints in high-level languages.** The constraint satisfaction model is useful because of its mathematical simplicity on one hand, and its ability to capture many real-life problems on the other. Yet, to make this framework useful for many real-life applications, advanced tools for modeling and for implementation are necessary. For this reason, constraint systems (providing some built-in propagation and solution algorithms) are usually embedded within a high-level programming environment which assists in the modeling phase and which allows for some control over the solution method itself.

## 2 Constraint propagation

**The basic idea.** *Constraint propagation* (or local consistency) algorithms [11, 8, 4] transform a given constraint problem into an equivalent one which is more explicit, by inferring new constraints that are added to the problem. Therefore, they may make some inconsistencies, which were implicitly contained in the problem specification, explicitly expressed. Intuitively, given a constraint problem, a constraint propagation algorithm will make any solution of a small subproblem extensible to some surrounding variables and constraints. These algorithms are interesting because their worst-case time complexity is polynomial in the size of the problem, and they are often very effective in discovering many local inconsistencies.

**Arc and path consistency.** The most basic and most popular propagation algorithm, called *arc-consistency*, ensures that any value in the domain of a single variable has a legal match in the domain of any other selected variable. This means that any solution of a one-variable subproblem is extensible in a consistent manner to another variable. The time complexity of this algorithm is quadratic in the size of the problem. Another well-known constraint propagation algorithm is *path-consistency*. This algorithm ensures that any solution of a two-variables subproblem is extensible to any third variable, and, as expected, it is more powerful than arc-consistency, discovering and removing more inconsistencies. It also requires more time: its time complexity is cubic in the size of the problem.

**I-consistency.** Arc- and path-consistency can be generalized to $i$-consistency. In general, *$i$-consistency* algorithms guarantee that any locally consistent instantiation of $i-1$ variables is extensible to any $i^{th}$ variable. Therefore, arc-consistency coincides with 2-consistency, and path-consistency is 3-consistency. Enforcing $i$-consistency can be accomplished in time and space exponential in $i$: if the constraint problem has $n$ variables, the complexity of achieving $i$-consistency is $O(n^i)$. Algorithms for $i$-consistency can sometimes reveal that the whole problem is inconsistent.

**Global consistency.** A constraint problem is said to be *globally consistent*, if it is $i$-consistent for *every $i$*. When such a situation arises, a solution can be assembled by assigning values to variables (using any variable ordering) without encountering any dead-end, namely in a *backtrack-free* manner.

**Adaptive-consistency as complete inference.** In practice, global consistency is not really necessary to have backtrack-free assignment of values: it is enough to posses *directional* global consistency relative to a given variable ordering. For example, the *adaptive consistency* algorithm, which is a *variable elimination* algorithm, enforces global consistency in a given order only, such that every solution can be extracted with no dead-ends along this ordering. [3]. Another related algorithm, called *tree-clustering*, compiles the given constraint problem into an equivalent tree of subproblems whose respective solutions can be combined into a complete solution efficiently. Adaptive-consistency and tree-clustering are complete inference algorithms that can take time and space exponential in a parameter of the constraint graph called *induced-width* (or *tree-width*) [3].

**Bucket-elimination.** *Bucket-elimination* is a recently proposed framework for variable-elimination algorithms which generalizes adaptive-consistency to include dynamic programming for optimization tasks, directional-resolution for propositional satisfiability, Fourier elimination for linear inequalities, as well as algorithms for probabilistic inference in Bayesian networks.

**Constraint propagation and search.** When a problem is computationally too hard for adaptive-consistency, it can be solved by bounding the amount of consistency-enforcing (e.g. arc- or path-consistency) and embedding these constraint propagation algorithms within a search component, as described in the next section. This yields a trade-off, between the effort spent in constraint propagation and that spent on the search, which can be exploited and which is the focus of empirical studies (described later).

## 3 Constraint satisfaction as search

**Backtracking search.** The most common algorithm for performing systematic search for a solution of a constraint problem is the so-called *backtracking* search algorithm. This algorithm traverses the space of partial solutions in a depth-first manner, and at each step it extends a partial solution (that is, a variable instantiation to a subset of variables which satisfies all the relevant constraints) by assigning a value to one more variable. When a variable is encountered such that none of its values are consistent with the current partial solution (a situation referred to as a *dead-end*), backtracking takes place, and the algorithm reconsiders one of the previous assignments. The best case occurs when the algorithm is able to successfully assign a value to each variable in a backtrack-free manner, without encountering any dead-end. In this case, the time complexity is linear in the size of the problem (often identified with the number of its variables). In the worst case, the time complexity of this algorithm is exponential in the size of the given problem. However, even in this case the algorithm requires only linear space.

**Look-ahead schemes.** Several improvements of backtracking have focused on one or both of the two phases of the algorithm: moving forward to a new variable (*look-ahead* schemes) and backtracking to a previous assignments (*look-back* schemes) [3]. When moving forward to extend a partial solution, some computation (e.g., arc-consistency) may be carried out to decide which variable, and which of the variable's values, to choose next in order to decrease the likelihood of a dead-end. For deciding on the next variable, variables that maximally constrain the rest of the search space are usually preferred, and therefore, the most highly constrained variable is selected. For value selection, instead, the least constraining value is preferred, in order to maximize future options for instantiations [6]. A well-known look-ahead method is *forward-checking*, which performs a limited form of arc-consistency at each step, ruling out some values that would lead to a dead-end. Currently, a popular form of look-ahead scheme, called *MAC* (for Maintaining Arc-Consistency), performs arc-consistency at each step and uses the revealed information for variable and value selection [5].

**Look-back schemes.** *Look-back* schemes are invoked when the algorithm encounters a dead-end. These schemes perform two functions. The first one is to decide how far to backtrack, by analyzing the reasons for the current dead-end, a process often referred to as *backjumping* [5]. The second one is to record the reasons for the dead-end in the form of new constraints so that the same conflict will not arise again, a process known as *constraint learning* and *no-good recording* [13].

**Local search.** *Stochastic local search (SLS)* strategies have been introduced into the constraint satisfaction literature in the 1990's under the name *GSAT (Greedy SATisfiability)* and are popular especially for solving propositional satisfiability problems. These methods move in a hill-climbing manner in the space of all variables' instantiations, and at each step they improve the current instantiation by changing (also called "flipping") the value of a variable so as to maximize the number of constraints satisfied. Such search algorithms are incomplete, since they may get stuck in a local maxima and thus might not be able to discover that a constraint problem is inconsistent. Nevertheless, when equipped with some heuristics for randomizing the search (e.g., WalkSat) or for revising the guiding criterion function, (e.g., constraint re-weighting), they have been shown to be reasonably successful in solving large problems that are frequently too hard to be handled by a backtracking-style search [12]. A well known local search algorithm for optimization tasks is *simulated annealing*.

**Evaluation of the algorithms.** The *theoretical evaluation* of constraint satisfaction algorithms is accomplished primarily by worst-case analysis, that is, determining a function of the problem's size that represents an upper bound of the algorithm's performance over all problems of that size. In particular, the tradeoff between constraint inference and search is hardly captured by such analysis. In addition, worst-case analysis, by its nature, is very pessimistic, and often it does not reflect the actual performance. Thus in most case an *empirical evaluation* is also necessary. Normally, an algorithm is evaluated empirically on a set of randomly generated problems, chosen in a way that they are reasonably hard to solve (this is done by selecting them from the *phase transition* region [12]). Several benchmarks, based on real-life applications such as scheduling, are also used to empirically evaluate an algorithm.

## 4 Tractable classes

In between search and constraint propagation algorithms, we may find the so-called *structure-driven algorithms*. These techniques emerged from an attempt to topologically characterize constraint problems that are tractable (that is, polynomially solvable). *Tractable classes* are generally recognized by realizing that enforcing low-level consistency (in polynomial time) guarantees global consistency for some problems.

**Graph-based tractability.** The basic constraint graph structure that support tractability is a tree. This has been observed repeatedly in constraint networks, complexity theory and database theory. In particular, enforcing arc-consistency on a tree-structured constraint problem ensures global consistency along an ordering. Most other graph-based techniques can be viewed as transforming a given network into a meta-tree. Among these, we find methods such as *tree-clustering* and *adaptive-consistency*, the *cycle-cutset scheme*, as well as the *bi-connected component decomposition*. These lead to a general characterization of tractability that uses the notion of *induced-width* [3].

**Constraint-based tractability.** Some tractable classes have also been characterized by special properties of the constraints, without any regard to the topology of the constraint graph. For example, tractable classes of temporal constraints include subsets of the qualitative interval algebra, expressing relationships such as "time interval $A$ overlaps or precedes time interval $B$", as well as quantitative binary linear inequalities over the Real numbers of the form $X - Y \leq a$ [10]. In general, we exploit notions such as tight domains and tight constraints, row-convex constraints [14], implicational and max-ordered constraints, as well as causal networks. A connection between tractability and algebraic closure was recently discovered [2].

## 5 Constraint optimization and soft constraints

**Searching for a most preferred solution.** While constraint satisfaction tasks involve finding any solution which satisfies all constraints, constraint optimization seeks the *best* solution relative to one (or more) criteria, usually represented via a cost- or an objective function. For example, we may have the constraints $X \leq Y, Y \leq 10$ with the objective function $f = X + Y$, to be maximized. Then, the best solution (only one for this example) is $X = 10, Y = 10$. All other solutions, (like $X = 5, Y = 6$), although satisfying all constraints, are less preferred. Frequently, cost functions are specified additively, as a sum of cost components, each defined on subsets of variables.

**Branch & bound.** Extending backtracking search for the task of selecting the most preferred (best-cost) solution, yields the well-known *branch and bound* algorithm. Like backtracking, branch and bound traverses the search tree in a depth-first manner, pruning not only partial instantiations that are inconsistent, but also those that are evaluated to be inferior to the current best solution. Namely, at each node, the value of the current partial solution is estimated (by an evaluation function) and compared with the current best solution; if it is inferior, search along the path is terminated. When the evaluation function is accurate, branch and bound prunes substantial portions of the search tree.

**Soft constraints.** One way to specify preferences between solutions is to attach a level of importance to each constraint or to each of its tuples. This type of constraints was introduced because real problems often cannot be described by a set of true/false statements only. Often relationships are associated with features such as preferences, probabilities, costs, and uncertainties. Moreover, many real problems, even when modeled correctly, are often over-constrained. This type of constraints is currently investigated under the formalism of *soft constraints*. There are several frameworks for soft constraints, such as the *semi-ring based formalism* [1], where each tuple in each constraint has an associated element taken from a partially ordered set (a semi-ring), and the *valued constraint formalism*, where each constraint is associated with an element from a totally ordered set. These formalisms are general enough to model classical constraints, weighted constraints, *fuzzy constraints*, and *over-constrained* problems. Current research effort is focused on extending propagation and search techniques to this more general framework.

## 6 Constraint programming

In a constraint solving or optimization system, it is frequently desired to have a certain level of flexibility in choosing the propagation technique and the search control method that suits the particular application. For this reason, constraint systems are usually embedded into high-level programming environments which allow for such a control.

**Logic programming.** Although many programming paradigms have recently been augmented with constraints, the concept of constraint programming is mainly linked to the *Logic Programming* (LP) framework. Logic programming is a declarative programming paradigm where a program is seen as a logical theory and has the form of a set of rules (called *clauses*) which relate the truth value of an atom (the *head* of the clause) to that of a set of other atoms (the *body* of the clause). The clause
`p(X,Y) :- q(X), r(X,Y,Z).`
says that if atoms `q(X)` and `r(X,Y,Z)` are true, then also atom `p(X,Y)` is true. For example, the clauses
`reach(X,Y) :- flight(X,Y).` (namely, there is a direct flight)
`reach(X,Y) :- flight(X,Z), reach(Z,Y).`
describe the reachability between two cities (X and Y) via a sequence of direct flights.

**Logic programming and search.** Executing a logic program means asking for the truth value of a certain predicate, called the *goal*. For example, the goal `:- p(X,Y)` asks whether there are values for the variables `X` and `Y` such that `p(X,Y)` is true in the given logic program. The answer is found by recursively *unifying* the current goal with the head of a clause (by finding values for the variables which make the two atoms equal). As with constraint solving, the algorithm that searches for such an answer in LP involves a backtracking search.

**From logic programming to constraint logic programming.** To use constraints within LP, one just has to treat some of the predicates in a clause as constraints and to replace unification with constraint solving. The resulting programming paradigm is called *Constraint Logic Programming* (CLP) [7, 9]. A typical example of a clause in CLP is

```
p(X,Y) :- X < Y+1, q(X), r(X,Y,Z).
```

which states that `p(X,Y)` is true if `q(X)` and `r(X,Y,Z)` are true, and if the value of $X$ is smaller than that of $Y + 1$. While the regular predicates are treated as in LP, constraints are manipulated using specialized constraint processing tools. The shift from LP to CLP permits the choice from among several constraint domains, yielding an effective scheme that can solve many more classes of real-life problems.

**Specialized algorithms for CLP.** CLP languages are reasonably efficient, due to the use of a collection of specialized solving and propagation algorithms for frequently used constraints and for special variable domain shapes. *Global constraints* and *bounds-consistency* are two aspects of such techniques which are incorporated into most current CLP languages.

**Global constraints.** Global constraints are just regular constraints, usually non-binary, for which there exist specialized, powerful propagation methods. They are called "global" because they are normally used, in the modeling phase, in place of a collection of smaller constraints. Their purpose is to overcome the ineffectiveness associated with propagation methods in the presence of certain small constraints. A typical example is the constraint `alldifferent`, which requires all the involved variables to assume a different value, and for which there is an effective propagation algorithm based on bi-partite matching. This constraint is used in place of a set of binary disequality constraints, which seldom give rise to useful constraint propagation. Most current CLP languages are equipped with several kinds of global constraints.

**Bounds consistency** . Bounds consistency is an approximated version of arc-consistency which is applicable to integer domains, and which was presented and is being used within most CLP languages. Rather than expressing integer domains explicitly, only their minimum and maximum bounds are stored. To maintain this compact representation, arc-consistency is restricted to generate shrinked domains having also an interval representation (i.e., no holes are allowed). Bounds consistency has a major impact on the efficiency of constraint propagation over integer domains and it is therefore an integral part of most constraint languages.

## 7 Summary

This survey provides the main notions underlying most CSP-related research and discusses the main issues. The CSP area is very interdisciplinary, since it embeds ideas from many research fields, like Artificial Intelligence (where it first started), Databases, Programming Languages, and Operation Research. Thus it is not possible to cover all the lines of work related to CSPs. However, we feel that the information contained in this survey constitutes a good starting point for those who are interested in either using CSPs for their purposes, or actively working to make CSPs more useful.

Ongoing investigations related to constraints currently focus on many issues, among which: identification of new tractable classes; studying the relationship between search and propagation; extending propagation techniques to soft constraints; developing more flexible and efficient constraint languages.

## References

[1] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2):201–236, March 1997.

[2] D. Cohean P. Jeavons and M. Gyssens. A unifying framework for tractable constraints. In *Constraint Programming (CP'95)*, France, 1995.

[3] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.

[4] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[5] J. Gaschnig. Performance measurement and analysis of search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University, 1979.

[6] M. Haralick and G. L. Elliot. Increasing tree-search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[7] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19 and 20, 1994.

[8] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[9] K. Marriott and P. J. Stuckey. *Programming with constraints: an introduction.* MIT Press, 1998.

[10] I. Meiri R. Dechter and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1990.

[11] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(66):95–132, 1974.

pages 17–24, Anaheim, CA, 1990.

[12] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.

[13] M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196.

[14] P. van Beek and R. Dechter. On the minimality and decomposability of row-convex constraint networks. *Journal of the ACM*, 42:543–561, 1995.

## Further readings

- S. Arnborg and A. Proskourowski. Linear time algorithms for np-hard problems restricted to partial $k$-trees. *Discrete and Applied Mathematics*, 23:11–24, 1989.
- N. Beldicenau and E. Contejean. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modeling*, 20(12), 1994.
- R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
- R. Dechter. Constraint networks. *Encyclopedia of Artificial Intelligence*, 2nd Ed. John Wiley and Sons, pages 276–285, 1992.
- M. C. Golumbic and R. Shamir. Complexity and algorithms for reasoning about time: a graph-theoretic approach. *Journal of the ACM*, 40:1108–1133, 1993.
- J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1993.
- A. K. Mackworth. Constraint Satisfaction. *Encyclopedia of Artificial Intelligence*, 2nd Ed. John Wiley and Sons, pages 285–293, 1992.
- D. Maier. The theory of relational databases. In *Computer Science Press, Rockville, MD*, 1983.
- E. Tsang. Foundation of Constraint Satisfaction. *Academic press*, 1993.
- M. Wallace. Practical Applications of Constraint Programming. *Constraints: An International Journal*, Vol. 1, 1996.