Handbook of Knowledge Representation
Edited by F. van Harmelen, V. Lifschitz and B. Porter

© 2008 Elsevier B.V. All rights reserved
DOI: 10.1016/S1574-6526(07)03004-0

Chapter 4

Constraint Programming

Francesca Rossi, Peter van Beek, Toby Walsh

4.1 Introduction

Constraint programming is a powerful paradigm for solving combinatorial search problems that draws on a wide range of techniques from artificial intelligence, operations research, algorithms, graph theory and elsewhere. The basic idea in constraint programming is that the user states the constraints and a general purpose constraint solver is used to solve them. Constraints are just relations, and a constraint satisfaction problem (CSP) states which relations should hold among the given decision variables. More formally, a constraint satisfaction problem consists of a set of variables, each with some domain of values, and a set of relations on subsets of these variables. For example, in scheduling exams at an university, the decision variables might be the times and locations of the different exams, and the constraints might be on the capacity of each examination room (e.g., we cannot schedule more students to sit exams in a given room at any one time than the room's capacity) and on the exams scheduled at the same time (e.g., we cannot schedule two exams at the same time if they share students in common). Constraint solvers take a real-world problem like this represented in terms of decision variables and constraints, and find an assignment to all the variables that satisfies the constraints. Extensions of this framework may involve, for example, finding optimal solutions according to one or more optimization criterion (e.g., minimizing the number of days over which exams need to be scheduled), finding all solutions, replacing (some or all) constraints with preferences, and considering a distributed setting where constraints are distributed among several agents.

Constraint solvers search the solution space systematically, as with backtracking or branch and bound algorithms, or use forms of local search which may be incomplete. Systematic method often interleave search (see Section 4.3) and inference, where inference consists of propagating the information contained in one constraint to the neighboring constraints (see Section 4.2). Such inference reduces the parts of the search space that need to be visited. Special propagation procedures can be devised to suit specific constraints (called global constraints), which occur often in real life. Such global constraints are an important component in the success of constraint pro-

gramming. They provide common patterns to help users model real-world problems. They also help make search for a solution more efficient and more effective.

While constraint problems are in general NP-complete, there are important classes which can be solved polynomially (see Section 4.4). They are identified by the connectivity structure among the variables sharing constraints, or by the language to define the constraints. For example, constraint problems where the connectivity graph has the form of a tree are polynomial to solve.

While defining a set of constraints may seem a simple way to model a real-world problem, finding a good model that works well with a chosen solver is not easy. A poorly chosen model may be very hard to solve. Moreover, solvers can be designed to take advantage of the features of the model such as symmetry to save time in finding a solution (see Section 4.5). Another problem with modeling real-world problems is that many are over-constrained. We may need to specify preferences rather than constraints. Soft constraints (see Section 4.6) provide a formalism to do this, as well as techniques to find an optimal solution according to the specified preferences. Many of the constraint solving methods like constraint propagation can be adapted to be used with soft constraints.

A constraint solver can be implemented in any language. However, there are languages especially designed to represent constraint relations and the chosen search strategy. These languages are logic-based, imperative, object-oriented, or rule-based. Languages based on logic programming (see Section 4.7) are well suited for a tight integration between the language and constraints since they are based on similar notions: relations and (backtracking) search.

Constraint solvers can also be extended to deal with relations over more than just finite (or enumerated) domains. For example, relations over the reals are useful to model many real-world problems (see Section 4.8). Another extension is to multi-agent systems. We may have several agents, each of which has their own constraints. Since agents may want to keep their knowledge private, or their knowledge is so large and dynamic that it does not make sense to collect it in a centralized site, distributed constraint programming has been developed (see Section 4.9).

This chapter necessarily covers some of the issues that are central to constraint programming somewhat superficially. A deeper treatment of these and many other issues can be found in the various books on constraint programming that have been written [5, 35, 53, 98, 70, 135–137].

4.2 Constraint Propagation

One of the most important concepts in the theory and practice of constraint programming is that of local consistency. A *local inconsistency* is an instantiation of some of the variables that satisfies the relevant constraints but cannot be extended to one or more additional variables and so cannot be part of any solution. If we are using a backtracking search to find a solution, such an inconsistency can be the reason for many deadends in the search and cause much futile search effort. This insight has led to: (a) the definition of conditions that characterize the level of local consistency of a CSP (e.g., [49, 95, 104]), (b) the development of constraint propagation algorithms—algorithms which enforce these levels of local consistency by removing inconsistencies from a CSP (e.g., [95, 104]), and (c) effective backtracking algorithms

for finding solutions to CSPs that maintain a level of local consistency during the search (e.g., [30, 54, 68]). In this section, we survey definitions of local consistency and constraint propagation algorithms. Backtracking algorithms integrated with constraint propagation are the topic of a subsequent section.

4.2.1 Local Consistency

Currently, arc consistency [95, 96] is the most important local consistency in practice and has received the most attention. Given a constraint, a value for a variable in the constraint is said to have a *support* if there exists values for the other variables in the constraint such that the constraint is satisfied. A constraint is *arc consistent* or if every value in the domains of the variables of the constraint has a support. A constraint can be made arc consistent by repeatedly removing unsupported values from the domains of its variables. Removing unsupported values is often referred to as *pruning* the domains. For constraints involving more than two variables, arc consistency is often referred to as *hyper arc consistency* or *generalized arc consistency*. For example, let the domains of variables x and y be $\{0, 1, 2\}$ and consider the constraint $x + y = 1$. Enforcing arc consistency on this constraint would prune the domains of both variables to just $\{0, 1\}$. The values pruned from the domains of the variables are locally inconsistent—they do not belong to any set of assignments that satisfies the constraint—and so cannot be part of any solution to the entire CSP. Enforcing arc consistency on a CSP requires us to iterate over the domain value removal step until we reach a fixed point. Algorithms for enforcing arc consistency have been extensively studied and refined (see, e.g., [95, 11] and references therein). An optimal algorithm for an arbitrary constraint has $O(rd^r)$ worst case time complexity, where r is the arity of the constraint and d is the size of the domains of the variables [103].

In general, there is a trade-off between the cost of the constraint propagation performed at each node in the search tree, and the amount of pruning. One way to reduce the cost of constraint propagation, is to consider more restricted local consistencies. One important example is bounds consistency. Suppose that the domains of the variables are large and ordered and that the domains of the variables are represented by intervals (the minimum and the maximum value in the domain). With bounds consistency, instead of asking that each value in the domain has a support in the constraint, we only ask that the minimum value and the maximum value each have a support in the constraint. Although bounds consistency is weaker than arc consistency, it has been shown to be useful for arithmetic constraints and global constraints as it can sometimes be enforced more efficiently (see below).

For some types of problems, like temporal constraints, it may be worth enforcing even stronger levels of local consistency than path consistency [95]. A problem involving binary constraints (that is, relations over just two variables) is *path consistent* if every consistent pair of values for two variables can be extended to any third variables. To make a problem path consistent, we may have to add additional binary constraints to rule out consistent pairs of values which cannot be extended.

4.2.2 Global Constraints

Although global constraints are an important aspect of constraint programming, there is no clear definition of what is and is not a global constraint. A global constraint is

a constraint over some sequence of variables. Global constraints also usually come with a constraint propagation algorithm that does more pruning or performs pruning cheaper than if we try to express the global constraint using smaller relations. The canonical example of a global constraint is the `all-different` constraint. An `all-different` constraint over a set of variables states that the variables must be pairwise different. The `all-different` constraint is widely used in practice and because of its importance is offered as a built-in constraint in most, if not all, major commercial and research-based constraint programming systems. Starting with the first global constraints in the CHIP constraint programming system [2], hundreds of global constraints have been proposed and implemented (see, e.g., [7]).

The power of global constraints is two-fold. First, global constraints ease the task of modeling an application as a CSP. Second, special purpose constraint propagation algorithms can be designed which take advantage of the semantics of the constraint and are therefore much more efficient. As an example, recall that enforcing arc consistency on an arbitrary has $O(rd^r)$ worst case time complexity, where r is the arity of the constraint and d is the size of the domains of the variables. In contrast, the `all-different` constraint can be made arc consistent in $O(r^2d)$ time in the worst case [116], and can be made bounds consistent in $O(r)$ time [100].

Other examples of widely applicable global constraints are the global cardinality constraint (`gcc`) [117] and the `cumulative` constraint [2]. A `gcc` over a set of variables and values states that the number of variables instantiating to a value must be between a given upper and lower bound, where the bounds can be different for each value. A `cumulative` constraint over a set of variables representing the time where different tasks are performed ensures that the tasks are ordered such that the capacity of some resource used at any one time is not exceeded. Both of these types of constraint commonly occur in rostering, timetabling, sequencing, and scheduling applications.

4.3 Search

The main algorithmic technique for solving constraint satisfaction problems is search. A search algorithm for solving a CSP can be either complete or incomplete. Complete, or systematic algorithms, come with a guarantee that a solution will be found if one exists, and can be used to show that a CSP does not have a solution and to find a provably optimal solution. Incomplete, or non-systematic algorithms, cannot be used to show a CSP does not have a solution or to find a provably optimal solution. However, such algorithms are often effective at finding a solution if one exists and can be used to find an approximation to an optimal solution. In this section, we survey backtracking and local search algorithms for solving CSPs, as well as hybrid methods that draw upon ideas from both artificial intelligence (AI) and operations research (OR). Backtracking search algorithms are, in general, examples of systematic complete algorithms. Local search algorithms are examples of incomplete algorithms.

4.3.1 Backtracking Search

A backtracking search for a solution to a CSP can be seen as performing a depth-first traversal of a search tree. This search tree is generated as the search progresses. At a

node in the search tree, an uninstantiated variable is selected and the node is extended where the branches out of the node represent alternative choices that may have to be examined in order to find a solution. The method of extending a node in the search tree is often called a branching strategy. Let x be the variable selected at a node. The two most common branching strategies are to instantiate x in turn to each value in its domain or to generate two branches, $x = a$ and $x \neq a$, for some value a in the domain of x. The constraints are used to check whether a node may possibly lead to a solution of the CSP and to prune subtrees containing no solutions.

Since the first uses of backtracking algorithms in computing [29, 65], many techniques for improving the efficiency of a backtracking search algorithm have been suggested and evaluated. Some of the most important techniques include constraint propagation, nogood recording, backjumping, heuristics for variable and value ordering, and randomization and restart strategies. The best combinations of these techniques result in robust backtracking algorithms that can now routinely solve large, and combinatorially challenging instances that are of practical importance.

Constraint propagation during search

An important technique for improving efficiency is to maintain a level of local consistency during the backtracking search by performing constraint propagation at each node in the search tree. This has two important benefits. First, removing inconsistencies during search can dramatically prune the search tree by removing many dead ends and by simplifying the remaining subproblem. In some cases, a variable will have an empty domain after constraint propagation; i.e., no value satisfies the unary constraints over that variable. In this case, backtracking can be initiated as there is no solution along this branch of the search tree. In other cases, the variables will have their domains reduced. If a domain is reduced to a single value, the value of the variable is forced and it does not need to be branched on in the future. Thus, it can be much easier to find a solution to a CSP after constraint propagation or to show that the CSP does not have a solution. Second, some of the most important variable ordering heuristics make use of the information gathered by constraint propagation to make effective variable ordering decisions. As a result of these benefits, it is now standard for a backtracking algorithm to incorporate some form of constraint propagation.

The idea of incorporating some form of constraint propagation into a backtracking algorithm arose from several directions. Davis and Putnam [30] propose unit propagation, a form of constraint propagation specialized to SAT. McGregor [99] and Haralick and Elliott proposed the *forward checking* backtracking algorithm [68] which makes the constraints involving the most recently instantiated variable arc consistent. Gaschnig [54] suggests *maintaining arc consistency* on all constraints during backtracking search and gives the first explicit algorithm containing this idea. Mackworth [95] generalizes Gaschnig's proposal to backtracking algorithms that interleave case-analysis with constraint propagation.

Nogood recording

One of the most effective techniques known for improving the performance of backtracking search on a CSP is to add implied constraints or nogoods. A constraint is *implied* if the set of solutions to the CSP is the same with and without the constraint.

A *nogood* is a special type of implied constraint, a set of assignments for some subset of variables which do not lead to a solution. Adding the "right" implied constraints to a CSP can mean that many deadends are removed from the search tree and other deadends are discovered after much less search effort. Three main techniques for adding implied constraints have been investigated. One technique is to add implied constraints by hand during the modeling phase. A second technique is to automatically add implied constraints by applying a constraint propagation algorithm. Both of the above techniques rule out local inconsistencies or deadends *before* they are encountered during the search. A third technique is to automatically add implied constraints *after* a local inconsistency or deadend is encountered in the search. The basis of this technique is the concept of a nogood—a set of assignments that is not consistent with any solution.

Once a nogood for a deadend is discovered, it can be ruled out by adding a constraint. The technique, first informally described by Stallman and Sussman [130], is often referred to as nogood or constraint recording. The hope is that the added constraints will prune the search space in the future. Dechter [31] provides the first formal account of discovering and recording nogoods. Ginsberg's [61] dynamic backtracking algorithm performs nogood recording coupled with a strategy for deleting nogoods in order to use only a polynomial amount of space. Schiex and Verfaillie [125] provide the first formal account of nogood recording within an algorithm that performs constraint propagation.

Backjumping

Upon discovering a deadend in the search, a backtracking algorithm must uninstantiate some previously instantiated variable. In the standard form of backtracking—called chronological backtracking—the most recently instantiated variable becomes uninstantiated. However, backtracking chronologically may not address the reason for the deadend. In backjumping, the algorithm backtracks to and retracts the decision which bears some responsibility for the deadend. The idea is to (sometimes implicitly) record nogoods or explanations for failures in the search. The algorithms then reason about these nogoods to determine the highest point in the search tree that can safely be jumped to without missing any solutions. Stallman and Sussman [130] were the first to informally propose a non-chronological backtracking algorithm—called dependency-directed backtracking—that discovered and maintained nogoods in order to backjump. The first explicit backjumping algorithm was given by Gaschnig [55]. Subsequent generalizations of Gaschnig's algorithm include Dechter's [32] graph-based backjumping algorithm and Prosser's [113] conflict-directed backjumping algorithm.

Variable and value ordering heuristics

When solving a CSP using backtracking search, a sequence of decisions must be made as to which variable to branch on or instantiate next and which value to give to the variable. These decisions are referred to as the variable and the value ordering. It has been shown that for many problems, the choice of variable and value ordering can be crucial to effectively solving the problem (e.g., [58, 62, 68]). When solving a CSP using backtracking search interleaved with constraint propagation, the domains

of the unassigned variables are pruned using the constraints and the current set of branching constraints. Many of the most important variable ordering heuristics are based on choosing the variable with the smallest number of values remaining in its domain (e.g., [65, 15, 10]). The principle being followed in the design of many value ordering heuristics is to choose next the value that is most likely to succeed or be a part of a solution (e.g., [37, 56]).

Randomization and restart strategies

It has been widely observed that backtracking algorithms can be brittle on some instances. Seemingly small changes to a variable or value ordering heuristic, such as a change in the ordering of tie-breaking schemes, can lead to great differences in running time. An explanation for this phenomenon is that ordering heuristics make mistakes. Depending on the number of mistakes and how early in the search the mistakes are made (and therefore how costly they may be to correct), there can be a large variability in performance between different heuristics. A technique called randomization and restarts has been proposed for taking advantage of this variability (see, e.g., [69, 66, 144]). A restart strategy $S = (t_1, t_2, t_3, \ldots)$ is an infinite sequence where each t_i is either a positive integer or infinity. The idea is that a randomized backtracking algorithm is run for t_1 steps. If no solution is found within that cutoff, the algorithm is restarted and run for t_2 steps, and so on until a solution is found.

4.3.2 Local Search

In backtracking search, the nodes in the search tree represent partial sets of assignments to the variables in the CSP. In contrast, a local search for a solution to a CSP can be seen as performing a walk in a directed graph where the nodes represent complete assignments; i.e., every variable has been assigned a value from its domain. Each node is labeled with a cost value given by a cost function and the edges out of a node are given by a neighborhood function. The search graph is generated as the search progresses. At a node in the search graph, a neighbor or adjacent node is selected and the algorithm "moves" to that node, searching for a node of lowest cost. The basic framework applies to both satisfaction and optimization problems and can handle both hard (must be satisfied) and soft (desirable if satisfied) constraints (see, e.g., [73]). For satisfaction problems, a standard cost function is the number of constraints that are not satisfied. For optimization problems, the cost function is the measure of solution quality given by the problem. For example, in the Traveling Salesperson Problem (TSP), the cost of a node is the cost of the tour given by the set of assignments associated with the node.

Four important choices must be made when designing an effective local search algorithm. First is the choice of how to start search by selecting a starting node in the graph. One can randomly pick a complete set of assignments or attempt to construct a "good" starting point. Second is the choice of neighborhood. Example neighborhoods include picking a single variable/value assignment and assigning the variable a new value from its domain and picking a pair of variables/value assignments and swapping the values of the variables. The former neighborhood has been shown to work well in SAT and n-queens problems and the latter in TSP problems. Third is the choice of "move" or selection of adjacent node. In the popular min-conflicts heuristic [102],

a variable x is chosen that appears in a constraint that is not satisfied. A new value is then chosen for x that minimizes the number of constraints that are not satisfied. In the successful GSAT algorithm for SAT problems [127], a best-improvement move is performed. A variable x is chosen and its value is flipped (true to false or vice versa) that leads to the largest reduction in the cost function—the number of clauses that are not satisfied. Fourth is the choice of stopping criteria for the algorithm. The stopping criteria is usually some combination of an upper bound on the maximum number of moves or iterations, a test whether a solution of low enough cost has been found, and a test whether the number of iterations since the last (big enough) improvement is too large.

The simplest local search algorithms continually make moves in the graph until all moves to neighbors would result in an increase in the cost function. The final node then represents the solution to the CSP. However, note that the solution may only be a local minima (relative to its neighbors) but not globally optimal. As well, if we are solving a satisfaction problem, the final node may not actually satisfy all of the constraints. Several techniques have been developed for improving the efficiency and the quality of the solutions found by local search. The most important of these include: multi-starts where the algorithm is restarted with different starting solutions and the best solution found from all runs is reported and threshold accepting algorithms that sometimes move to worse cost neighbors to escape local minima such as simulated annealing [83] and tabu search [63]. In simulated annealing, worse cost neighbors are moved to with a probability that is gradually decreased over time. In tabu search, a move is made to a neighbor with the best cost, even if it is worse than the cost of the current node. However, to prevent cycling, a history of the recently visited nodes called a tabu list is kept and a move to a node is blocked if it appears on the tabu list.

4.3.3 Hybrid Methods

Hybrid methods combine together two or more solution techniques. Whilst there exist interesting hybrids of systematic and local search methods, some of the most promising hybrid methods combine together AI and OR techniques like backtracking and linear programming. Linear programming (LP) is one of the most powerful techniques to have emerged out of OR. In fact, if a problem can be modeled by linear inequalities over continuous variables, then LP is almost certainly a better method to solve it than CP.

One of the most popular approaches to bring linear programming into CP is to create a *relaxation* of (some parts of) the CP problem that is linear. Relaxation may be both dropping the integrality requirement on some of the decision variables or on the tightness of the constraints. Linear relaxations have been proposed for a number of global constraints including the `all different`, `circuit` and `cumulative` constraints [72]. Such relaxations can then be given to a LP solver. The LP solution can be used in a number of ways to prune domains and guide search. For example, it can tighten bounds on a variable (e.g., the variable representing the optimization cost). We may also be able to prune domains by using reduced costs or Lagrange multipliers. In addition, the continuous LP solution may by chance be integral (and thus be a solution to the original CP model). Even if the LP solution is not integral, we can use it to guide search (e.g., branching on the most non-integral variable). One of the advantages of

using a linear relaxation is that the LP solver takes a more global view than a CP solver which just makes "local" inferences.

Two other well-known OR techniques that have been combined with CP are branch and price and Bender's decomposition. With branch and price, CP can be used to perform the column generation, identifying variables to add dynamically to the search. With Bender's decomposition, CP can be used to perform the row generation, generating new constraints (nogoods) to add to the model. Hybrid methods like these have permitted us to solve problems beyond the reach of either CP or OR alone. For example, a CP based branch and price hybrid was the first to solve the 8-team traveling tournament problem to optimality [43].

4.4 Tractability

Constraint satisfaction is NP-complete and thus intractable in general. It is easy to see how to reduce a problem like graph 3-coloring or propositional satisfiability to a CSP. Considerable effort has therefore been invested in identifying restricted classes of constraint satisfaction problems which are tractable. For Boolean problems where the decision variables have just one of two possible values, Schaefer's dichotomy theorem gives an elegant characterization of the six tractable classes of relations [121]: those that are satisfied by only true assignments; those that are satisfied by only false assignments; Horn clauses; co-Horn clauses (i.e., at most one negated variable); 2-CNF clauses; and affine relations. It appears considerably more difficult to characterize tractable classes for non-Booleans domains. Research has typically broken the problem into two parts: tractable languages (where the relations are fixed but they can be combined in any way), and tractable constraint graphs (where the constraint graph is restricted but any sort of relation can be used).

4.4.1 Tractable Constraint Languages

We first restrict ourselves to instances of constraint satisfaction problems which can be built using some limited language of constraint relations. For example, we might consider the class of constraint satisfaction problems built from just the not-equals relation. For k-valued variables, this gives k-coloring problems. Hence, the problem class is tractable iff $k \leqslant 2$.

Some examples

We consider some examples of tractable constraint languages. Zero/one/all (or ZOA) constraints are binary constraints in which each value is supported by zero, one or all values [25]. Such constraints are useful in scene labeling and other problems. ZOA constraints are tractable [25] and can, in fact, be solved in $O(e(d+n))$ where e is the number of constraints, d is the domain size and n is the number of variables [149]. This results generalizes the result that 2-SAT is linear since every binary relation on a Boolean domain is a ZOA constraint. Similarly, this result generalizes the result that functional binary constraints are tractable. The ZOA constraint language is maximal in the sense that, if we add any relation to the language which is not ZOA, the language becomes NP-complete [25].

Another tractable constraint language is that of connected row-convex constraints [105]. A binary constraint C over the ordered domain D can be represented by a 0/1 matrix M_{ij} where $M_{ij} = 1$ iff $C(i, j)$ holds. Such a matrix is row-convex iff the non-zero entries in each row are consecutive, and connected row-convex iff it is row-convex and, after removing empty rows, it is connected (non-zero entries in consecutive rows are adjacent). Finally a constraint is connected row-convex iff the associated 0/1 matrix and its transpose are connected row-convex. Connected row-convex constraints include monotone relations. They can be solved without backtracking using a path-consistency algorithm. If a constraint problem is path-consistent and only contains row-convex constraints (not just connected row-convex constraints), then it can be solved in polynomial time [133]. Row-convexity is not enough on its own to guarantee tractability as enforcing path-consistency may destroy row-convexity.

A third example is the language of max-closed constraints. Specialized solvers have been developed for such constraints in a number of industrial scheduling tools. A constraint is max-closed iff for all pairs of satisfying assignments, if we take the maximum of each assignment, we obtain a satisfying assignment. Similarly a constraint is min-closed iff for all pairs of satisfying assignments, if we take the minimum of each assignment, we obtain a satisfying assignment. All unary constraints are max-closed and min-closed. Arithmetic constraints like $aX = bY + c$, and $\sum_i a_i X_i \geqslant b$ are also max-closed and min-closed. Max-closed constraints can be solved in quadratic time using a pairwise-consistency algorithm [82].

Constraint tightness

Some of the simplest possible tractability results come from looking at the constraint tightness. For example, Dechter shows that for a problem with domains of size d and constraints of arity at most k, enforcing strong $(d(r-1)+1)$-consistency guarantees global consistency [33]. We can then construct a solution without backtracking by repeatedly assigning a variable and making the resulting subproblem globally consistent. Dechter's result is tight since certain types of constraints (e.g., binary inequality constraints in graph coloring) require exactly this level of local consistency.

Stronger results can be obtained by looking more closely at the constraints. For example, a k-ary constraint is m-tight iff given any assignment for $k-1$ of the variables, there are at most m consistent values for the remaining variable. Dechter and van Beek prove that if all relations are m-tight and the network is strongly relational $(m+1)$-consistent, then it is globally consistent [134]. A complementary result holds for constraint looseness. If constraints are sufficiently loose, we can guarantee that the network must have a certain level of local consistency.

Algebraic results

Jeavons et al. have given a powerful algebraic treatment of tractability of constraint languages using relational clones, and polymorphisms on these cones [79–81]. For example, they show how to construct a so-called "indicator" problem that determines whether a constraint language over finite domains is NP-complete or tractable. They are also able to show that the search problem (where we want to find a solution) is no harder than the corresponding decision problem (where we want to just determine if a solution exists or not).

Dichotomy results

As we explained, for Boolean domains, Schaefer's result completely characterizes the tractable constraint languages. For three valued variables, Bulatov has provided a more complex but nevertheless complete dichotomy result [16]. Bulatov also has given a cubic time algorithm for identifying these tractable cases. It remains an open question if a similar dichotomy result holds for constraint languages over any finite domain.

Infinite domains

Many (but not all) of the tractability results continue to hold if variable domains are infinite. For example, Allen's interval algebra introduces binary relations and compositions of such relations over time intervals [3]. This can be viewed as a binary constraint problem over intervals on the real line. Linear Horn is an important tractable class for temporal reasoning. It properly includes the point algebra, and ORD-Horn constraints. A constraint over an infinite ordered set is *linear Horn* when it is equivalent to a finite disjunction of linear disequalities and at most one weak linear inequality. For example, $(X - Y \leqslant Z) \vee (X + Y + Z \neq 0)$ is linear Horn [88, 84].

4.4.2 Tractable Constraint Graphs

We now consider tractability results where we permit any sort of relation but restrict the constraint graph in some way. Most of these results concern tree or tree-like structures. We need to distinguish between three types of constraint graph: the *primal* constraint graph has a node for each variable and edges between variables in the same constraint, the *dual* constraint graph has a node for each constraint and edges between constraints sharing variables, and the constraint *hypergraph* has a node for each variable and a hyperedge between all the variables in each constraint.

Mackworth gave one of the first tractability results for constraint satisfaction problems: a binary constraint networks whose primal graph is a tree can be solved in linear time [97]. More generally, a constraint problem can be solved in a time that is exponential in the induced width of the primal graph for a given variable ordering using a join-tree clustering or (for space efficiency) a variable elimination algorithm. The induced width is the maximum number of parents to any node in the induced graph (in which we add edges between any two parents that are both connected to the same child). For non-binary constraints, we tend to obtain tighter results by considering the constraint hypergraph [67]. For example, an acyclic non-binary constraint problem will have high tree-width, even though it can be solved in quadratic time. Indeed, results based on hypertree width have been proven to strongly dominate those based on cycle cutset width, biconnected width, and hinge width [67].

4.5 Modeling

Constraint programming is, in some respects, one of the purest realizations of the dream of declarative programming: you state the constraints and the computer solves them using one of a handful of standard methods like the maintaining arc consistency backtracking search procedure. In reality, constraint programming falls short of this dream. There are usually many logically equivalent ways to model a problem. The

model we use is often critical as to whether or not the problem can be solved. Whilst modeling a problem so it can be successfully solved using constraint programming is an art, a number of key lessons have started to be identified.

4.5.1 CP ∨ ¬ CP

We must first decide if constraint programming is a suitable technology in which to model our problem, or whether we should consider some other approach like mathematical programming or simulation. It is often hard to answer this question as the problem we are trying to solve is often not well defined. The constraints of the problem may not have been explicitly identified. We may therefore have to extract the problem constraints from the user in order to build a model. To compound matters, for economic and other reasons, problems are nearly always over-constrained. We must therefore also identify the often conflicting objectives (price, speed, weight, ...) that need to be considered. We must then decide which constraints to consider as hard, which constraints to compile into the search strategy and heuristics, and which constraints to ignore.

Real world combinatorial search problems are typically much too large to solve exactly. Problem decomposition is therefore a vital aspect of modeling. We have to decide how to divide the problem up and where to make simplifying approximations. For example, in a production planning problem, we might ignore how the availability of operators but focus first on scheduling the machines. Having decided on a production schedule for the machines, we can then attempt to minimize the labor costs.

Another key concern in modeling a problem is stability. How much variability is there between instances of the problem? How stable is the solution method to small changes? Is the problem very dynamic? What happens if (a small amount of) the data changes? Do solutions need to be robust to small changes? Many such questions need to be answered before we can be sure that constraint programming is indeed a suitable technology.

4.5.2 Viewpoints

Having decided to use constraint programming, we then need to decide the variables, their possible domains and the constraints that apply to these variables. The concept of viewpoint [57, 19] is often useful at this point. There are typically several different viewpoints that we can have of a problem. For example, if we are scheduling the next World Cup, are we assigning games to time slots, or time slots to games? Different models can be built corresponding to each of these viewpoints. We might have variables representing the games with their values being time slots, or we might have variables representing the time slots with their values being games.

A good rule of thumb is to choose the viewpoint which permits the constraints to be expressed easily. The hope is that the constraint solver will then be able to reason with the constraints effectively. In some cases, it is best to use multiple viewpoints and to maintain consistency between them with *channeling* constraints [19]. One common viewpoint is a *matrix model* in which the decision variables form a matrix or array [48, 47]. For example, we might need to decide which factory processes which order. This can be modeled with an 0/1 matrix O_{ij} which is 1 iff order i is processed in factory j.

The constraint that every order is processed then becomes the constraint that every row sums to 1.

To help specify the constraints, we might introduce auxiliary variables. For example, in the Golomb ruler problem (prob006 in CSPLib.org), we wish to mark ticks on an integer ruler so that all the distances between ticks are unique. The problem has applications in radio-astronomy and elsewhere. One viewpoint is to have a variable for each tick, whose value is the position on the ruler. To specify the constraint that all the distances between ticks are unique, we can an introduce auxiliary variable D_{ij} for the distance between the ith and jth tick [128]. We can then post a global `all-different` constraint on these auxiliary variables. It may be helpful to permit the constraint solver to branch on the auxiliary variables. It can also be useful to add implied (or redundant) constraints to help the constraint solver prune the search space. For example, in the Golomb ruler problem, we can add the implied constraint that $D_{ij} < D_{ik}$ for $j < k$ [128]. This will help reduce search.

4.5.3 Symmetry

A vital aspect of modeling is dealing with symmetry. Symmetry occurs naturally in many problems (e.g., if we have two identical machines to schedule, or two identical jobs to process). Symmetry can also be introduced when we model a problem (e.g., if we name the elements in a set, we introduce the possibility of permuting their order). We must deal with symmetry or we will waste much time visiting symmetric solutions, as well as parts of the search tree which are symmetric to already visited parts. One simple but highly effective mechanism to deal with symmetry is to add constraints which eliminate symmetric solutions [27]. Alternatively, we can modify the search procedure to avoid visiting symmetric states [44, 59, 118, 126].

Two common types of symmetry are variable symmetries (which act just on variables), and value symmetries (which act just on values) [21]. With variable symmetries, there are a number of well understood symmetry breaking methods. For example, many problems can be naturally modeled using a matrix model in which the rows and columns of the matrix are symmetric and can be freely permuted. We can break such symmetry by lexicographically ordering the rows and columns [47]. Efficient constraint propagation algorithms have therefore been developed for such ordering constraints [51, 17]. Similarly, with value symmetries, there are a number of well understood symmetry breaking methods. For example, if all values are interchangeable, we can break symmetry by posting some simple precedence constraints [92]. Alternatively, we can turn value symmetry into variable symmetry [47, 93, 114] and then use one of the standard methods for breaking variable symmetry.

4.6 Soft Constraints and Optimization

It is often the case that, after having listed the desired constraints among the decision variables, there is no way to satisfy them all. That is, the problem is *over-constrained*. Even when all the constraints can be satisfied, and there are several solutions, such solutions appear equally good, and there is no way to discriminate among them. These scenarios often occur when constraints are used to formalize desired properties rather than requirements that cannot be violated. Such desired properties should rather be