The editors

Chapter 1

Constraint Programming

Francesca Rossi, Peter van Beek, Toby Walsh

1.1 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 1.3) and inference, where inference consists of propagating the information contained in one constraint to the neighboring constraints (see Section 1.2). 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 programming. 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 1.4). 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 1.5). 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 1.6) 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 1.7) 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 1.8). 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 1.9).

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, 136, 137].

1.2 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.

1.2.1 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* 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 tradeoff 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.

1.2.2 Global constraints

Although global constraints are an important aspect of constraint programming, there is no clear definition of what is and isn't 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 builtin 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 $O(r^2 d)$ 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.

1.3 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.

1.3.1 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 $S = (t_1, t_2, t_3, ...)$ 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.

1.3.2 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.

1.3.3 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].

1.4 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).

1.4.1 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 $k \leq 2$.

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 $\sum_i a_i X_i \geq b$ 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, 80, 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, $(X - Y \leq Z) \vee (X + Y + Z \neq 0)$ is linear Horn [88, 84].

1.4.2 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].

1.5 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.

1.5.1 CP $\vee$ $\neg$ 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.

1.5.2 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.

1.5.3 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.

1.6 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 considered as *preferences*, whose violation should be avoided as far as possible. *Soft constraints* provide one way to model such preferences.

1.6.1 Modeling soft constraints

There are many classes of soft constraints. The first one that was introduced concerns the so-called *fuzzy constraints* and it is based on fuzzy set theory [42, 41]. A fuzzy constraint is not a set (of allowed tuples of values to variables), but rather a *fuzzy set* [42], where each element has a graded degree of membership. For each assignment of values to its variables, we do not have to say whether it belongs to the set or not, but how much it does so. This allows us to represent the fact that a combination of values for the variables of the constraint is partially permitted. We can also say that the membership degree of an assignment gives us the *preference* for that assignment. In fuzzy constraints, preferences are between 0 and 1, with 1 being complete acceptance and 0 being total rejection. The preference of a solution is then computed by taking the minimal preference over the constraints. This may seem awkward in some scenarios, but it is instead very natural, for example, when we are reasoning about critical applications, such as space or medical applications, where we want to be as cautious as possible. *Possibilistic constraints* [122] are very similar to fuzzy constraints and they have the same expressive power: priorities are associated to constraints and the aim is to find an assignment which minimizes the priority of the most important violated constraint.

Lack of discrimination among solutions with the same minimal preferences is one of the main drawbacks of fuzzy constraints (the so-called *drowning effect*). To avoid this, one can use *fuzzy lexicographic constraints* [45]. The idea is to consider not just the least preference value, but all the preference values when evaluating a complete assignment, and to sort such values in increasing order. When two complete assignments are compared, the two sorted preference lists are then compared lexicographically.

There are situations where we are more interested in the damages we get by not satisfying a constraint rather than in the advantages we obtain when we satisfy it. A natural way to extend the classical constraint formalism to deal with these situations consists of associating a certain penalty or cost to each constraint, to be paid when the constraint is violated. A *weighted constraint* is thus just a classical constraint plus a weight. The cost of an assignment is the sum of all weights of those constraints which are violated. An optimal solution is a complete assignment with minimal cost. In the particular case when all penalties are equal to 1, this is called the MAX-CSP problem [50]. In fact, in this case the task consists of finding an assignment where the number of violated constraints is minimal, which is equivalent to say that the number of satisfied constraints is maximal.

Weighted constraints are among the most expressive soft constraint frameworks, in the sense that the task of finding an optimal solution for fuzzy, possibilistic, or lexicographic constraint problems can be efficiently reduced to the task of finding an optimal solution for a weighted constraint problem [124].

The literature contains also at least two general formalisms to model soft constraints, of which all the classes above are instances: *semiring-based constraints* [13, 14] and *valued constraints* [124]. Semiring-based constraints rely on a simple algebraic structure which is very similar to a semiring, and it is used to formalize the notion of preferences (or satisfaction degrees), the way preferences are ordered, and how to combine them. The minimum preference is used to capture the notion of absolute non-satisfaction, which is typical of hard constraints. Similarly for the maximal preference, which can model complete satisfaction. Valued constraints rely on a different algebraic structure, a positive totally ordered commutative monoid, and use a different syntax than semiring-based constraints. However, they have the same expressive power, if we assume preferences to be totally ordered [12]. Partially ordered preferences can be useful for example when we need to reason with more than one optimization criterion, since in this case there could be situations which are naturally not comparable.

Soft constraint problems are as expressive, and as difficult to solve, as constraint optimization problems, which are just constraint problems plus an objective function. In fact, given any soft constraint problem, we can always build a constraint optimization problem with the same solution ordering, and vice versa.

1.6.2 Searching for the best solution

The most natural way to solve a soft constraint problem, or a constraint optimization problem, is to use Branch and Bound. *Depth First Branch and bound* (DFBB) performs a depth-first traversal of the search tree. At each node, it keeps a lower bound lb and an upper bound ub. The *lower bound* is an underestimation of the violation degree of any complete assignment below the current node. The *upper bound* ub is the maximum violation degree that we are willing to accept. When $ub \leq lb(t)$, the subtree can be pruned because it contains no solution with violation degree lower than ub. The time complexity of DFBB is exponential, while its space complexity is linear. The efficiency of DFBB depends largely on its pruning capacity, that relies on the quality of its bounds: the higher lb and the lower ub, the better DFBB performs, since it does more pruning, exploring a smaller part of the search tree. Thus many efforts have been made to improve (that is, to increase) the lower bound.

While the simplest lower bound computation takes into account just the past variables (that is, those already assigned), more sophisticated lower bounds include contributions of other constraints or variables. For example, a lower bound which considers constraints among past and future variables has been implemented in the *Partial Forward Checking* (PFC) algorithm [50]. Another lower bound, which includes contributions from constraints among future variables, was first implemented in [143] and then used also in [89, 90], where the algorithm PFC-MRDAC has been shown to give a substantial improvement in performance with respect to previous approaches. An alternative lower bound is presented within the *Russian doll search* algorithm [140] and in the *specialized* RDS approach [101],

1.6.3 Inference in soft constraints

Inference in classical constraint problems consists of computing and adding implied constraints, producing a problem which is more explicit and hopefully easier to solve. If this process is always capable of solving the problem, then inference is said to be complete. Otherwise, inference is incomplete and it has to be complemented with search. For classical constraints, adaptive consistency enforcing is complete while local consistency (such as arc or path consistency) enforcing is in general incomplete. Inference in soft constraints keeps the same basic idea: adding constraints that will make the problem more explicit without changing the set of solutions nor their preference. However, with soft constraints, the addition of a new constraint could change the semantics of the constraint problem. There are cases though where an arbitrary implied constraint can be added to an existing soft constraint problem while getting an equivalent problem: when preference combination is idempotent.

Bucket elimination

Bucket elimination (BE) [34, 35] is a complete inference algorithm which is able to compute all optimal solutions of a soft constraint problem (as opposed to one optimal solution, as usually done by search strategies). It is basically the extension of the *adaptive consistency* algorithm [37] to the soft case. BE has both a time and a space complexity which are exponential in the induced width of the constraint graph, which essentially measures the graph cyclicity. The high memory cost, that comes from the high arity of intermediate constraints that have to be stored as tables in memory, is the main drawback of BE to be used in practice. When the arity of these constraints remains reasonable, BE can perform very well [91]. It is always possible to limit the arity of intermediate constraints, at the cost of losing optimality with respect to the returned level and the solution found. This approach is called *mini-bucket elimination* and it is an approximation scheme for BE.

Soft constraint propagation

Because complete inference can be extremely time and space intensive, it is often interesting to have simpler processes which are capable of producing just a lower bound on the violation degree of an optimal solution. Such a lower bound can be immediately useful in Branch and Bound algorithms. This is what soft constraint propagation does.

Constraint propagation is an essential component of any constraint solver. A *local consistency* property is identified (such as arc or path consistency), and an associated enforcing algorithm (usually polynomial) is developed to transform a constraint problem into a unique and equivalent network which satisfies the local consistency property. If this equivalent network has no solution, then the initial network is obviously inconsistent too. This allows one to detect some inconsistencies very efficiently. A similar motivation exists for trying to adapt this approach to soft constraints: the hope that an equivalent locally consistent problem may provide a better lower bound during the search for an optimal solution. The first results in the area were obtained on fuzzy networks [129, 122]. Then, [13, 14] generalized them to semiring-based constraints with idempotent combination.

If we take the usual notions of local consistency like arc or path consistency and replace constraint conjunction by preference combination, and tuple elimination by preference lowering, we immediately obtain a soft constraint propagation algorithm. If preference combination is idempotent, then this algorithm terminates and yields a unique equivalent arc consistent soft constraints problem. Idempotency is only sufficient, and can be slightly relaxed, for termination. It is however possible to show that it is a necessary condition to guarantee equivalence.

However, many real problems do not rely on idempotent operators because such operators provide insufficient discrimination, and rather rely on frameworks such as weighted or lexicographic constraints, which are not idempotent. For these classes of soft constraints, equivalence can still be maintained, compensating the addition of new constraints by the ”subtraction” of others. This can be done in all *fair* classes of soft constraints [24], where it is possible to define the notion of ”subtraction”. In this way, arc consistency has been extended to fair valued structures in [123, 26]. While equivalence and termination in polynomial time can be achieved, constraint propagation on non-idempotent classes of soft constraints does not assure the uniqueness of the resulting problem.

Several global constraints and their associated algorithms have been extended to handle soft constraints. All these proposals have been made using the approach of [111] where a soft constraint is represented as a hard constraint with an extra variable representing the cost of the assignment of the other variables. Examples of global constraints that have been defined for soft constraints are the soft `all-different` and soft `gcc` [112, 138, 139].

1.7 Constraint Logic Programming

Constraints can, and have been, embedded in many programming environments, but some are more suitable than others. The fact that constraints can be seen as relations or predicates, that their conjunction can be seen as a *logical and*, and that backtracking search is a basic methodology to solve them, makes them very compatible with logic programming [94], which is based on predicates, logical conjunctions, and depth-first search. The addition of constraints to logic programming has given the *constraint logic programming* paradigm [77, 98].

1.7.1 Logic programs

Logic programming (LP) [94] is based on a unique declarative programming idea where programs are not made of statements (like in imperative programming) nor of functions (as in functional programming), but of logical implications between collections of predicates. A logic program is thus seen as a logical theory and has the form of a set of rules (called *clauses*) which relate the truth value of a literal (the *head* of the clause) to that of a collection of other literals (the *body* of the clause).

Executing a logic program means asking for the truth value of a certain statement, called the *goal*. Operationally, this is done by repeatedly transforming the goal via a sequence of *resolution steps*, until we either end up with the empty goal (in this case the proof is successful), or we cannot continue and we don't have the empty goal (and in this case we have a failure), or we continue forever (and in this case we have an infinite computation). Each resolution step involves the unification between a literal which is part of a goal and the head of a clause.

Finite domain CSPs can always be modeled in LP by using one clause for the definition of the problem graph and many facts to define the constraints. However, this modeling is not convenient, since LP's execution engine corresponds to depth-first search with chronological backtracking and this may not be the most efficient way to solve the CSP. Also, it ignores the power of constraint propagation in solving a CSP.

Constraint logic programming languages extend LP by providing many tools to improve the solution efficiency using constraint processing techniques. They also extend CSPs by accommodating constraints defined via formulas over a specific language of constraints (like arithmetic equations and disequations over the reals, or term equations, or linear disequations over finite domains).

1.7.2 Constraint logic programs

Syntactically, constraints are added to logic programming by just considering a specific constraint type (for example, linear equations over the reals) and then allowing constraints of this type in the body of the clauses. Besides the usual resolution engine of logic programming, one has a (complete or incomplete) constraint solving system, which is able to check the consistency of constraints of the considered type. This simple change provides many improvements over logic programming. First, the concept of unification is generalized to constraint solving: the relationship between a goal and a clause (to be used in a resolution step) can be described not just via term equations but via more general statements, that is, constraints. This allows for a more general and flexible way to control the flow of the computation. Second, expressing constraints by some language (for example, linear equations and disequations) gives more compactness and structure. Finally, the presence of an underlying constraint solver, usually based on incomplete constraint propagation of some sort, allows for the combination of backtracking search and constraint propagation, which can give more efficient complete solvers.

To execute a CLP program, at each step we must find a most general unifier between the selected subgoal and the head. Moreover, we have to check the consistency of the current set of constraints with the constraints in the body of the clause. Thus two solvers are involved: unification, and the specific constraint solver for the constraints in use. The constraint consistency check can use some form of constraint propagation, thus applying the principle of combining depth-first backtracking search with constraint propagation, as usual in complete constraint solvers for CSPs.

Exceptional to CLP (and LP) is the existence of three different but equivalent semantics for such programs: declarative, fixpoint, and operational [98]. This means that a CLP program has a declarative meaning in terms of set of first-order formulas but can also be executed operationally on a machine.

CLP is not a single programming language, but a programming *paradigm*, which is parametric with respect to the class of constraints used in the language. Working with a particular CLP language means choosing a specific class of constraints (for example, finite domains, linear, or arithmetic) and a suitable constraint solver for that class. For example, CLP over finite domain constraints uses a constraint solver which is able to perform consistency checks and projection over this kind of constraints. Usually, the consistency check is based on constraint propagation similar to, but weaker than, arc consistency (called *bounds consistency*).

1.7.3 LP and CLP languages

The concept of logic programming [94, 132] was first developed in the ’70s, while the first constraint logic programming language was Prolog II [23], which was designed by Colmerauer in the early 80’s. Prolog II could treat term equations like Prolog, but in addition could also handle term disequations. After this, Jaffar and Lassez observed that both term equations and disequations were just a special form of constraints, and developed the concept of a constraint logic programming scheme in 1987 [76]. From then on, several instances of the CLP scheme were developed: Prolog III [22], with constraints over terms, strings, booleans, and real linear arithmetic; CLP(R) [75], with constraints over terms and real arithmetics; and CHIP [39], with constraints over terms, finite domains, and finite ranges of integers.

Constraint logic programming over finite domains was first implemented in the late 80’s by Pascal Van Hentenryck [70] within the language CHIP [39]. Since then, newer constraint propagation algorithms have been developed and added to more recent CLP(FD) languages, like GNU Prolog [38] and ECLiPSe [142].

1.7.4 Other programming paradigms

Whilst constraints have been provided in declarative languages like CLP, constraint-based tools have also been provided for imperative languages in the form of libraries. The typical programming languages used to develop such solvers are C++ and Java. ILOG [1] is one the most successful companies to produce such constraint-based libraries and tools. ILOG has C++ and Java based constraint libraries, which uses many of the techniques described in this chapter, as well as a constraint-based configurator, scheduler and vehicle routing libraries.

Constraints have also been successfully embedded within concurrent constraint programming [120], where concurrent agents interact by posting and reading constraints in a shared store. Languages which follow this approach to programming are AKL [78] and Oz [71]. Finally, high level modeling languages exist for modeling constraint problems and specifying search strategies. For example, OPL [135] is a modeling language similar to AMPL in which constraint problems can be naturally modeled and the desired search strategy easily specified, while COMET is an OO programming language for constraint-based local search [136]. CHR (Constraint Handling Rules) is instead a rule-based language related to CLP where constraint solvers can be easily modeled [52].

1.8 Beyond Finite Domains

Real-world problems often take us beyond finite domain variables. For example, to reason about power consumption, we might want a decision variable to range over the reals. Constraint programming has therefore been extended to deal with more than just finite (or enumerated) domains of values. In this section, we consider three of the most important extensions.

1.8.1 Intervals

The constraint programming approach to deal with continuous decision variables is typically via intervals [20, 28, 74, 107]. We represent the domain of a continuous variable by a set of disjoint intervals. In practice, the bounds on an interval are represented by machine representable numbers such as floats. We usually solve a continuous problem by finding a covering of the solution space by means of a finite set of multi-dimensional interval boxes with some required precision. Such a covering can be found by means of a branch-and-reduce algorithm which branches (by splitting an interval box in some way into several interval boxes) and reduces (which applies some generalization of local consistency like box or hull consistency to narrow the size of the interval boxes [8]). If we also have an optimization criteria, a bounding procedure can compute bounds on the objective within the interval boxes. Such bounds can be used to eliminate interval boxes which cannot contain the optimal objective value. Alternatively, direct methods for solving a continuous constraint problem involve replacing the classical algebraic operators in the constraints by interval operators and using techniques like Newton's methods to narrow the intervals [137].

1.8.2 Temporal problems

A special class of continuous constraint problems for which they are specialized and often more efficient solving methods are temporal constraint problems. Time may be represented by points (e.g. the point algebra) or by interval of time points (e.g. the interval algebra). Time points are typically represented by the integers, rationals or reals (or, in practice, by machine representations of these). For the interval algebra (IA), Allen introduced [3] an influential formalism in which constraints on time intervals are expressed in terms of 13 mutually exclusive and exhaustive binary relations (e.g. this interval is before this other interval, or this interval is during this other interval). Deciding the consistency of a set of such interval constraints is NP-complete. In fact, there are 18 maximal tractable (polynomial) subclasses of the interval algebra (e.g. the ORD-Horn subclass introduced by Nebel and Bürckert) [106]. The point algebra (PA) introduced by Vilain and Kautz [141] is more tractable. In this algebra, time points can be constrained by ordering, equality, or a disjunctive combination of ordering and equality constraints. Koubarakis proved that enforcing strong 5-consistency is a necessary and sufficient condition for achieving global consistency on the point algebra. Van Beek gave an $O(n^2)$ algorithm for consistency checking and finding a solution. Identical results hold for the pointisable subclass of the IA (PIA) [141]. This algebra consists of those elements of the IA that can be expressed as a conjunction of binary constraints using only elements of PA. A number of richer representations of temporal information have also been considered including disjunctive binary difference constraints [36] (i.e. $\bigvee_i a_i \leq x_j - x_k \leq b_i$), and simple disjunctive problems [131] (i.e. $\bigvee_i a_i \leq x_i - y_i \leq b_i$). Naturally, such richer representations tend to be more intractable.

1.8.3 Sets and other datatypes

Many combinatorial search problems (e.g. bin packing, set covering, and network design) can be naturally represented in the language of sets, multisets, strings, graphs and other structured objects. Constraint programming has therefore been extended to deal with variables which range over such datatypes. For example, we can represent a decision variable which ranges over sets of integers by means of an upper and lower bound on the possible and necessary elements in the set (e.g. [60]). This is more compact both to represent and reason with than the exponential number of possible sets between these two bounds. Such a representation necessarily throws away some information. We cannot, for example, represent a decision variable which takes one of the two element sets: $\{1, 2\}$ or $\{3, 4\}$. To represent this, we need an empty lower bound and an upper bound of $\{1, 2, 3, 4\}$. Two element sets like $\{2, 3\}$ and $\{1, 4\}$ also lie within these bounds. Local consistency techniques have been extended to deal with such set variables. For instance, a set variable is bound consistent iff all the elements in its lower bound occur in every solution, and all the elements in its upper bound occur in at least one solution. Global constraints have also been defined for such set variables [119, 9, 115] (e.g. a sequence of set variables should be pairwise disjoint). Variables have also been defined over other richer datatypes like multisets (or bags) [87, 145], graphs [40], strings [64] and lattices [46].

1.9 Distributed Constraint Programming

Constraints are often generated by several different agents. Typical examples are scheduling meetings, where each person has his own constraints and all have to be satisfied to find a common time to meet. It is natural in such problems to have a decentralized solving algorithm. Of course, even when constraints are produced by several agents, one could always collect them all in one place and solve them by using a standard centralized algorithm. This certainly saves the time to exchange messages among agents during the execution of a distributed algorithm, which could make the execution slow. However, this is often not desirable, since agents may want to keep some constraints as private. Moreover, a centralized solver makes the whole system less robust.

Formally, a distributed CSP is just a CSP plus one agent for each variable. The agent controls the variable and all its constraints (see, e.g., [147]). Backtracking search, which is the basic form of systematic search for constraint solving, can be easily extended to the distributed case by passing a partial instantiation from an agent to another one, which will add the instantiation for a new variable, or will report the need to backtrack. Forward checking, backjumping, constraint propagation, and variable and value ordering heuristics can also be adapted to this form of distributed *synchronous backtracking*, by sending appropriate messages. However, in synchronous backtracking one agent is active at any given time, so the only advantage with respect to a centralized approach is that agents keep their constraints private.

On the contrary, in asynchronous distributed search, all agents are active at the same time, and they coordinate only to make sure that what they do on their variable is consistent with what other agents do on theirs. Asynchronous backtracking [148] is the main algorithm which follows this approach. Branch and bound can also be adapted to work in a distributed asynchronous setting.

Various improvements to these algorithms can be made. For example, variables can be instantiated with a dynamic rather than a fixed order, and agents can control constraints rather than variables. The Asynchronous Weak Commitment search algorithm [146] adopts a dynamic reordering. However, this is achieved via the use of much more space (to store the nogoods), otherwise completeness is lost.

Other search algorithms can be adapted to a distributed environment. For example, the DPOP algorithm [109] performs distributed dynamic programming. Also local search is very well suited for a distributed setting. In fact, local search works by making incremental modifications to a complete assignment, which are usually local to one or a small number of variables.

Open constraint problems are a different kind of distributed problems, where variable domains are incomplete and can be generated by several distributed agents. Domains are therefore incrementally discovered, and the aim is to solve the problem even if domains are not completely known. Both solutions and optimal solutions for such problems can be obtained in a distributed way without the need to know the entire domains. This approach can be used within several algorithms, such as the DPOP algorithm for distributed dynamic programming [110].

1.10 Application Areas

Constraint programming has proven useful in important applications from industry, business, manufacturing, and science. In this section, we survey three general application areas—vehicle routine, scheduling, and configuration—with an emphasis on why constraint programming has been successful and why constraint programming is now often the method of choice for solving problems in these domains.

Vehicle Routing is the task of constructing routes for vehicles to visit customers at minimum cost. A vehicle has a maximum capacity which cannot be exceeded and the customers may specify time windows in which deliveries are permitted. Much work on constraint programming approaches to vehicle routing has focused on alternative constraint models and additional implied constraints to increase the amount of pruning performed by constraint propagation. Constraint programming is well-suited for vehicle routing because of its ability to handle real-world (or side) constraints. Vehicle routing problems that arise in practice often have unique constraints that are particular to a business entity. In non-constraint programming approaches, such side constraints often have to be handled in an ad hoc manner. In constraint programming a wide variety of side constraints can be handled simply by adding them to the core model (see, e.g., [86, 108])

Scheduling is the task of assigning resources to a set of activities to minimize a cost function. Scheduling arises in diverse settings including in the allocation of gates to incoming planes at an airport, crews to an assembly line, and processes to a CPU. Constraint programming approaches to scheduling have aimed at generality, with the ability to seamlessly handle side constraints. As well, much effort has gone into improved implied constraints such as global constraints, edge-finding constraints and timetabling constraints, which lead to powerful constraint propagation. Additional advantages of a constraint propagation approach to scheduling include the ability to form hybrids of backtracking search and local search and the ease with which scheduling or domain specific heuristics can be incorporated within the search routines (see, e.g., [6, 18]).

Configuration is the task of assembling or configuring a customized system from a catalog of components. Configuration arises in diverse settings including in the assembly of home entertainment systems, cars and trucks, and travel packages. Constraint programming is well-suited to configuration because of (i) its flexibility in modeling and the declarativeness of the constraint model, (ii) the ability to explain a failure to find a customized system when the configuration task is over-constrained and to subsequently relax the user's constraints, (iii) the ability to perform interactive configuration where the user makes a sequence of choices and after each choice constraint propagation is used to restrict future possible choices, and (iv) the ability to incorporate reasoning about the user's preferences (see, e.g., [4, 85]).

1.11 Conclusions

Constraint programming is now a relatively mature technology for solving a wide range of difficult combinatorial search problems. The basic ideas behind constraint programming are simple: a declarative representation of the problem constraints, combined with generic solving methods like chronological backtracking or local search. Constraint programming has a number of strengths including: rich modeling languages in which to represent complex and dynamic real-world problems; fast and general purpose inference methods, like enforcing arc consistency, for pruning parts of the search space; fast and special purpose inference methods associated with global constraints; hybrid methods that combine the strengths of constraint programming and operations research; local search methods that quickly find near-optimal solutions; a wide range of extensions like soft constraint solving and distributed constraint solving in which we can represent more closely problems met in practice. As a result, constraint programming is now used in a wide range of businesses and industries including manufacturing, transportation, health care, advertising, telecommunications, financial services, energy and utilities, as well as marketing and sales. Companies like American Express, BMW, Coors, Danone, eBay, France Telecom, General Electric, HP, JB Hunt, LL Bean, Mitsubishi Chemical, Nippon Steel, Orange, Porsche, QAD, Royal Bank of Scotland, Shell, Travelocity, US Postal Service, Visa, Wal-Mart, Xerox, Yves Rocher, and Zurich Insurance all use constraint programming to optimize their business processes. Despite this success, constraint programming is not (and may never be) a push-button technology that works "out of the box". It requires sophisticated users who master a constraint programming system, know how to model problems and how to customize search methods to these models. Future research needs to find ways to lower this barrier to using this powerful technology.

Bibliography

[1] *Ilog Solver 4.4. Reference Manual.* ILOG SA, Gentilly, France, 1998.

[2] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17:57–73, 1993.

[3] J. Allen. Maintaining knowledge about temporal intervals. *Journal of the Association for Computing Machinery*, 26(11):832–843, 1983.

[4] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs: Application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.

[5] K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[6] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems.* Kluwer, 2001.

[7] N. Beldiceanu. Global constraints as graph properties on structured network of elementary constraints of the same type. Technical Report T2000/01, SICS, 2000.

[8] F. Benhamou, D. McAllester, and P. Van Hentenryck. Clp(intervals). In M. Bruynooghe, editor, *Proceedings of International Symposium on Logic Programming*, pages 124–138. MIT Press, 1994.

[9] C. Bessiere, E. Hebrard, B. Hnich, and T. Walsh. Disjoint, partition and intersection constraints for set and multiset variables. In *10th International Conference on Principles and Practices of Constraint Programming (CP-2004).* Springer-Verlag, 2004.

[10] C. Bessière and J.-C. Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 61–75, Cambridge, Mass., 1996.

[11] C. Bessière, J.-C. Régin, R. H. C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165:165–185, 2005.

[12] S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and valued CSPs: Frameworks, properties and comparison. *Constraints*, 4:199–240, 1999.

[13] S. Bistarelli, U. Montanari, and F. Rossi. Constraint solving over semirings. In *Proc. IJCAI 1995*, 1995.

[14] S. Bistarelli, U. Montanari, and F. Rossi. Semiring based constraint solving and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[15] D. Brélaz. New methods to color the vertices of a graph. *Comm. ACM*, 22:251–256, 1979.

[16] A.A. Bulatov. A dichotomy theorem for constraints on a three-element set. In *Proceedings of 43rd IEEE Symposium on Foundations of Computer Science (FOCS'02)*, pages 649–658, 2002.

[17] M. Carlsson and N. Beldiceanu. Arc-consistency for a chain of lexicographic ordering constraints. Technical report T2002-18, Swedish Institute of Computer Science, 2002. ftp://ftp.sics.se/pub/SICS-reports/Reports/SICS-T–2002-18–SE.ps.Z.

[18] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 369–383, Santa Margherita Ligure, Italy, 1994.

[19] B.M.W. Cheng, K.M.F. Choi, J.H.M. Lee, and J.C.K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4:167–192, 1999.

[20] J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

[21] D.A. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, and B.M. Smith. Symmetry definitions for constraint satisfaction problems. In P. van Beek, editor, *Proceedings of Eleventh International Conference on Principles and Practice of Constraint Programming (CP2005)*, pages 17–31. Springer, 2005.

[22] A. Colmerauer. An introduction to Prolog-III. *Communication of the ACM*, 1990.

[23] A. Colmerauer. Prolog II reference manual and theoretical model. Technical report, Technical report, Groupe Intelligence Artificielle, Université Aix-Mareseille II, October 1982.

[24] M. Cooper. High-order consistency in valued constraint satisfaction. *Constraints*, 10:283–305, 2005.

[25] M. Cooper, D. Cohen, and P. Jeavons. Characterizing tractable constraints. *Artificial Intelligence*, 65:347–361, 1994.

[26] M. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154(1-2):199–227, April 2004. see arXiv.org/abs/cs.AI/0111038.

[27] J. Crawford, G. Luks, M. Ginsberg, and A. Roy. Symmetry breaking predicates for search problems. In *Proceedings of the 5th International Conference on Knowledge Representation and Reasoning, (KR '96)*, pages 148–159, 1996.

[28] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[29] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. ACM*, 5:394–397, 1962.

[30] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, 1960.

[31] R. Dechter. Learning while searching in constraint satisfaction problems. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 178–183, Philadelphia, 1986.

[32] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[33] R. Dechter. From local to global consistency. *Artificial Intelligence*, 55:87–107, 1992.

[34] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1–2):41–85, 1999.

[35] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[36] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1–3):61–95, 1991.

[37] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[38] D. Diaz. *The GNU Prolog web site. URL: http://pauillac.inria.fr/~diaz/gnu-prolog/*.

[39] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proc. International Conference on Fifth Generation Computer Systems*. Tokyo, Japan, 1988.

[40] G. Dooms, Y. Deville, and P. Dupont. Cp(graph): Introducing a graph computation domain in constraint programming. In *10th International Conference on Principles and Practices of Constraint Programming (CP-2004)*. Springer-Verlag, 2004.

[41] D. Dubois, H. Fargier, and H. Prade. Using fuzzy constraints in job-shop scheduling. In *Proc. of IJCAI-93/SIGMAN Workshop on Knowledge-based Production Planning, Scheduling and Control*, Chambery, France, August 1993.

[42] D. Dubois and H. Prade. *Fuzzy sets and systems: theory and applications*. Academic Press, 1980.

[43] K. Easton, G. Nemhauser, and M. Trick. Solving the traveling tournament problem: a combined integer programming and constraint programming approach. In *Proceedings of the International Conference on the Practice and Theory of Automated Timetabling (PATAT 2002)*, 2002.

[44] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In T. Walsh, editor, *Proceedings of 7th International Conference on Principles and Practice of Constraint Programming (CP2001)*, pages 93–107. Springer, 2001.

[45] H. Fargier, J. Lang, and T. Schiex. Selecting preferred solutions in fuzzy constraint satisfaction problems. In *Proc. of the 1st European Congress on Fuzzy and Intelligent Technologies*, 1993.

[46] A.J. Fernandez and P.M. Hill. An interval constraint system for lattice domains. *ACM Transactions on Programming Languages and Systems (TOPLAS-2004)*, 26(1), 2004.

[47] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetry in matrix models. In *8th International Conference on Principles and Practices of Constraint Programming (CP-2002)*. Springer, 2002.

[48] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Matrix modelling. Technical Report APES-36-2001, APES group, 2001. Available from http://www.dcs.st-and.ac.uk/ apes/reports/apes-36-2001.ps.gz. Presented at Formul'01 Workshop on Modelling and Problem Formulation, CP2001 post-conference workshop.

[49] E. C. Freuder. Synthesizing constraint expressions. *Comm. ACM*, 21:958–966, 1978.

[50] E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[51] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *8th International Conference on Principles and Practices of Constraint Programming (CP-2002)*. Springer, 2002.

[52] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic*

programming, 37:95–138, 1998.

[53] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.

[54] J. Gaschnig. A constraint satisfaction method for inference making. In *Proceedings Twelfth Annual Allerton Conference on Circuit and System Theory*, pages 866–874, Monticello, Illinois, 1974.

[55] J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Canadian Conference on Artificial Intelligence*, pages 268–277, Toronto, 1978.

[56] P. A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 31–35, Vienna, 1992.

[57] P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of the 10th ECAI*, pages 31–35. European Conference on Artificial Intelligence, 1992.

[58] I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 179–193, Cambridge, Mass., 1996.

[59] I.P. Gent and B.M. Smith. Symmetry breaking in constraint programming. In W. Horn, editor, *Proceedings of ECAI-2000*, pages 599–603. IOS Press, 2000.

[60] C. Gervet. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(3):191–244, 1997.

[61] M. L. Ginsberg. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1:25–46, 1993.

[62] M. L. Ginsberg, M. Frank, M. P. Halpin, and M. C. Torrance. Search lessons learned from crossword puzzles. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 210–215, Boston, Mass., 1990.

[63] F. Glover and M. Laguna. *Tabu Search*. Kluwer, 1997.

[64] K. Golden and W. Pang. Constraint reasoning over strings. In F. Rossi, editor, *Proceedings of Ninth International Conference on Principles and Practice of Constraint Programming (CP2003)*, pages 377–391. Springer, 2003.

[65] S. Golomb and L. Baumert. Backtrack programming. *J. ACM*, 12:516–524, 1965.

[66] C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. of Automated Reasoning*, 24:67–100, 2000.

[67] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.

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

[69] W. D. Harvey. *Nonsystematic backtracking search*. PhD thesis, Stanford University, 1995.

[70] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[71] M. Henz, G. Smolka, and J. Wurtz. Oz – a programming language for multi-agent systems. In *Proc. 13th IJCAI*, 1995.

[72] J. Hooker. *Logic-Based Methods for Optimization: Combining Optimization and*

Constraint Satisfaction. Wiley, New York, 2000.

[73] H.H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, 2004.

[74] E. Hyvönen. Constraint reasoning based on interval arithmetic. *Artificial Intelligence*, 58:71–112, 1992.

[75] J. Jaffar and al. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 1992.

[76] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proc. POPL*. ACM, 1987.

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

[78] S. Janson. *AKL - A Multiparadigm Programming Language*. PhD thesis, Uppsala Theses in Computer Science 19, ISSN 0283-359X, ISBN 91-506-1046-5, Uppsala University, and SICS Dissertation Series 14, ISSN 1101-1335, ISRN SICS/D-14–SE, 1994.

[79] P. Jeavons, D.A. Cohen, and M. Cooper. Constraints, consistency and closure. *Artificial Intelligence*, 101(1-2):251–265, 1998.

[80] P. Jeavons, D.A. Cohen, and M. Gyssens. Closure properties of constraints. *Journal of the Association for Computing Machinery*, 44:527–548, 1997.

[81] P. Jeavons, D.A. Cohen, and M. Gyssens. How to determine the expressive power of constraints. *Constraints*, 4:113–131, 1999.

[82] P. Jeavons and M. Cooper. Tractable constraints on ordered domains. *Artificial Intelligence*, 79:327–339, 1995.

[83] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation: Part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, 1991.

[84] P. Jonsson and C. Backstrom. A unifying approach to temporal constraint reasoning. *Artificial Intelligence*, 102:143–155, 1998.

[85] U. Junker and D. Mailharro. Preference programming: Advanced problem solving for configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1):13–29, 2003.

[86] P. Kilby, P. Prosser, and P. Shaw. A comparison of traditional and constraint-based heuristic methods on vehicle routing problems with side constraints. *Constraints*, 5(4):389–414, 2000.

[87] Z. Kiziltan and T. Walsh. Constraint programming with multisets. In *Proceedings of the 2nd International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-02), held alongside CP-02*, 2002.

[88] M. Koubarakis. Tractable disjunctions of linear constraints. In E.C. Freuder, editor, *Proceedings of Second International Conference on Principles and Practice of Constraint Programming (CP96)*, pages 297–307. Springer, 1996.

[89] J. Larrosa and P. Meseguer. Exploiting the use of DAC in Max-CSP. In *Proc. of CP'96*, pages 308–322, Boston (MA), 1996.

[90] J. Larrosa and P. Meseguer. Partition-based lower bound for Max-CSP. In *Proc. of the 5^th International Conference on Principles and Practice of Constraint Programming (CP-99)*, pages 303–315, 1999.

[91] J. Larrosa, E. Morancho, and D. Niso. On the practical applicability of bucket elimination: Still-life as a case study. *Journal of Artificial Intelligence Research*,

23:421–440, 2005.

[92] Y.C. Law and J.H.M. Lee. Global constraints for integer and set value precedence. In *Proceedings of 10th International Conference on Principles and Practice of Constraint Programming (CP2004)*, pages 362–376. Springer, 2004.

[93] Y.C. Law and J.H.M. Lee. Breaking value symmetries in matrix models using channeling constraints. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC-2005)*, pages 375–380, 2005.

[94] J. W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1993.

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

[96] A. K. Mackworth. On reading sketch maps. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 598–606, Cambridge, Mass., 1977.

[97] A.K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

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

[99] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inform. Sci.*, 19:229–250, 1979.

[100] K. Mehlhorn and S. Thiel. Faster algorithms for bound-consistency of the sortedness and alldifferent constraint. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 306–319, Singapore, 2000.

[101] P. Meseguer and M. Sanchez. Specializing russian doll search. In *Principles and Practice of Constraint Programming - CP 2001*, volume 2239 of *LNCS*, pages 464–478, Paphos, Cyprus, November 2001. Springer Verlag.

[102] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–206, 1992.

[103] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 651–656, Munchen, Germany, 1988.

[104] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inform. Sci.*, 7:95–132, 1974.

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

[106] B. Nebel and H.-J. Burckert. Reasoning about temporal relations: A maximal tractable subclass of Allen's interval algebra. *Journal of the Association for Computing Machinery*, 42(1):43–66, 1995.

[107] W.J. Older and A. Vellino. Extending Prolog with constraint arithmetic on real intervals. In *Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering*. IEEE Computer Society Press, 1990.

[108] G. Pesant, M. Gendreau, J. Potvin, and J. Rousseau. An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32(1):12–29, 1998.

[109] A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of the 19th IJCAI*, pages 266–271, 2005.

[110] A. Petcu and B. Faltings. Odpop: An algorithm for open distributed constraint optimization. In *AAMAS 06 Workshop on Distributed Constraint Reasoning*, 2006.

[111] T. Petit, J-C. Régin, and C. Bessière. Meta-constraints on violations for over constrained problems. In *IEEE-ICTAI'2000 international conference*, pages 358–365, Vancouver, Canada, November 2000.

[112] T. Petit, J-C. Régin, and C. Bessière. Specific filtering algorithms for over-constrained problems. In *Principles and Practice of Constraint Programming - CP 2001*, volume 2239 of *LNCS*, pages 451–463, Paphos, Cyprus, November 2001. Springer Verlag.

[113] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[114] J-F. Puget. Breaking all value symmetries in surjection problems. In P. van Beek, editor, *Proceedings of Eleventh International Conference on Principles and Practice of Constraint Programming (CP2005)*. Springer, 2005.

[115] C-G. Quimper and T. Walsh. Beyond finite domains: the all different and global cardinality constraints. In *11th International Conference on Principles and Practices of Constraint Programming (CP-2005)*. Springer-Verlag, 2005.

[116] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 362–367, Seattle, 1994.

[117] J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 209–215, Portland, Oregon, 1996.

[118] C. Roney-Dougal, I. Gent, T. Kelsey, and S. Linton. Tractable symmetry breaking using restricted search trees. In *Proceedings of ECAI-2004*. IOS Press, 2004.

[119] A. Sadler and C. Gervet. Global reasoning on sets. In *Proceedings of Workshop on Modelling and Problem Formulation (FORMUL'01)*, 2001. held alongside CP-01.

[120] V. Saraswat. *Concurrent constraint programming*. MIT Press, 1993.

[121] T. Schaefer. The complexity of satisfiability problems. In *Proceedings of 10th ACM Symposium on Theory of Computation*, pages 216–226, 1978.

[122] T. Schiex. Possibilistic constraint satisfaction problems or "How to handle soft constraints ?". In *Proc. of the 8^{th} Int. Conf. on Uncertainty in Artificial Intelligence*, Stanford, CA, July 1992.

[123] T. Schiex. Arc consistency for soft constraints. In *Principles and Practice of Constraint Programming - CP 2000*, volume 1894 of *LNCS*, pages 411–424, Singapore, September 2000.

[124] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: hard and easy problems. In *Proc. IJCAI 1995*, pages 631–637, 1995.

[125] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 3:1–15, 1994.

[126] M. Sellmann and P. Van Hentenryck. Structural symmetry breaking. In *Proceedings of the 19th IJCAI*. International Joint Conference on Artificial Intelligence, 2005.

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

[128] B. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied con-

straints to model non-binary problems. In *Proceedings of the 16th National Conference on AI*, pages 182–187. American Association for Artificial Intelligence, 2000.

[129] P. Snow and E.C. Freuder. Improved relaxation and search methods for approximate constraint satisfaction with a maximin criterion. In *Proc. of the 8^{th} biennal conf. of the Canadian society for comput. studies of intelligence*, pages 227–230, May 1990.

[130] R. 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, 1977.

[131] K. Stergiou and M. Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence*, 120(1):81–117, 2000.

[132] L. Sterling and E. Shapiro. *The art of Prolog*. MIT Press, 1994.

[133] P. van Beek. On the minimality and decomposability of constraint networks. In *Proceedings of 10th National Conference on Artificial Intelligence*, pages 447–452. AAAI Press/The MIT Press, 1992.

[134] P. van Beek and R. Dechter. Constraint tightness and looseness versus local and global consistency. *Journal of the Association for Computing Machinery*, 44:549–566, 1997.

[135] P. van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, 1999.

[136] P. van Hentenryck and L. Michel. *Constraint-Based Local Search*. MIT Press, 2005.

[137] P. van Hentenryck, L. Michel, and Y. Deville. *Numerica: A Modeling Language for Global Optimization*. MIT Press, 1997.

[138] W. J. van Hoeve. A hyper-arc consistency algorithm for the soft alldifferent constraint. In *Proc. of the tenth international conference on Principles and Practice of Constraint Programming (CP 2004)*, number 3258 in LNCS, 2004.

[139] W. J. van Hoeve, G. Pesant, and L-M. Rousseau. On global warming (softening global constraints). In *Proc. of the 6th International Workshop on Preferences and Soft Constraints*, Toronto, Canada, 2004.

[140] G. Verfaillie, M. Lemaître, and T. Schiex. Russian doll search. In *Proc. AAAI 1996*, pages 181–187, Portland, Oregon, 1996.

[141] M. Vilain and H. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of 5th National Conference on Artificial Intelligence*, pages 377–382. Morgan Kaufmann, 1986.

[142] M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997. Available via http://eclipse.crosscoreop.com/eclipse/reports/index.html.

[143] R. J. Wallace. Directed arc consistency preprocessing. In M. Meyer, editor, *Selected papers from the ECAI-94 Workshop on Constraint Processing*, number 923 in LNCS, pages 121–137. Springer, Berlin, 1995.

[144] T. Walsh. Search in a small world. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1172–1177, Stockholm, 1999.

[145] T. Walsh. Consistency and propagation with multiset constraints: A formal viewpoint. In F. Rossi, editor, *9th International Conference on Principles and Practices of Constraint Programming (CP-2003)*. Springer, 2003.

[146] M. Yokoo. Weak-commitment search for solving constraint satisfaction problems. In *Proceedings of the 12th AAAI*, pages 313–318, 1994.

[147] M. Yokoo. *Distributed Constraint Satisfaction: Foundation of Cooperation in*

Multi-agent Systems. Springer, 2001.

[148] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the 12th ICDCS*, pages 614–621, 1992.

[149] Y. Zhang, R. Yap, and J. Jaffar. Functional eliminations and 0/1/all constraints. In *Proceedings of 16th National Conference on Artificial Intelligence*, pages 281–290. AAAI Press/The MIT Press, 1999.