

Chapter 4

Constraint Programming

Francesca Rossi, Peter van Beek, Toby Walsh

4.1 Introduction

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

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

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

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

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

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

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

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

4.2 Constraint Propagation

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

1 for finding solutions to CSPs that maintain a level of local consistency during the 1
2 search (e.g., [30, 54, 68]). In this section, we survey definitions of local consistency 2
3 and constraint propagation algorithms. Backtracking algorithms integrated with con- 3
4 straint propagation are the topic of a subsequent section. 4
5

6 4.2.1 Local Consistency 6

7 Currently, arc consistency [95, 96] is the most important local consistency in prac- 7
8 tice and has received the most attention. Given a constraint, a value for a variable in 8
9 the constraint is said to have a *support* if there exists values for the other variables 9
10 in the constraint such that the constraint is satisfied. A constraint is *arc consistent* or 10
11 if every value in the domains of the variables of the constraint has a support. A con- 11
12 straint can be made arc consistent by repeatedly removing unsupported values from the 12
13 domains of its variables. Removing unsupported values is often referred to as *prun-* 13
14 *ing* the domains. For constraints involving more than two variables, arc consistency 14
15 is often referred to as *hyper arc consistency* or *generalized arc consistency*. For ex- 15
16 ample, let the domains of variables x and y be $\{0, 1, 2\}$ and consider the constraint 16
17 $x + y = 1$. Enforcing arc consistency on this constraint would prune the domains of 17
18 both variables to just $\{0, 1\}$. The values pruned from the domains of the variables are 18
19 locally inconsistent—they do not belong to any set of assignments that satisfies the 19
20 constraint—and so cannot be part of any solution to the entire CSP. Enforcing arc 20
21 consistency on a CSP requires us to iterate over the domain value removal step until we 21
22 reach a fixed point. Algorithms for enforcing arc consistency have been extensively 22
23 studied and refined (see, e.g., [95, 11] and references therein). An optimal algorithm 23
24 for an arbitrary constraint has $O(rd^r)$ worst case time complexity, where r is the arity 24
25 of the constraint and d is the size of the domains of the variables [103]. 25
26

27 In general, there is a trade-off between the cost of the constraint propagation per- 27
28 formed at each node in the search tree, and the amount of pruning. One way to reduce 28
29 the cost of constraint propagation, is to consider more restricted local consistencies. 29
30 One important example is bounds consistency. Suppose that the domains of the vari- 30
31 ables are large and ordered and that the domains of the variables are represented by 31
32 intervals (the minimum and the maximum value in the domain). With bounds consis- 32
33 tency, instead of asking that each value in the domain has a support in the constraint, 33
34 we only ask that the minimum value and the maximum value each have a support 34
35 in the constraint. Although bounds consistency is weaker than arc consistency, it has 35
36 been shown to be useful for arithmetic constraints and global constraints as it can 36
37 sometimes be enforced more efficiently (see below). 37

38 For some types of problems, like temporal constraints, it may be worth enforcing 38
39 even stronger levels of local consistency than path consistency [95]. A problem involv- 39
40 ing binary constraints (that is, relations over just two variables) is *path consistent* if 40
41 every consistent pair of values for two variables can be extended to any third variables. 41
42 To make a problem path consistent, we may have to add additional binary constraints 42
43 to rule out consistent pairs of values which cannot be extended. 43

44 4.2.2 Global Constraints 44

45 Although global constraints are an important aspect of constraint programming, there 45
46 is no clear definition of what is and is not a global constraint. A global constraint is 46
47 47

1 a constraint over some sequence of variables. Global constraints also usually come 1
2 with a constraint propagation algorithm that does more pruning or performs prun- 2
3 ing cheaper than if we try to express the global constraint using smaller relations. 3
4 The canonical example of a global constraint is the `all-different` constraint. An 4
5 `all-different` constraint over a set of variables states that the variables must be 5
6 pairwise different. The `all-different` constraint is widely used in practice and 6
7 because of its importance is offered as a built-in constraint in most, if not all, ma- 7
8 jor commercial and research-based constraint programming systems. Starting with the 8
9 first global constraints in the CHIP constraint programming system [2], hundreds of 9
10 global constraints have been proposed and implemented (see, e.g., [7]). 10

11 The power of global constraints is two-fold. First, global constraints ease the task 11
12 of modeling an application as a CSP. Second, special purpose constraint propagation 12
13 algorithms can be designed which take advantage of the semantics of the constraint 13
14 and are therefore much more efficient. As an example, recall that enforcing arc con- 14
15 sistency on an arbitrary has $O(rd^r)$ worst case time complexity, where r is the arity 15
16 of the constraint and d is the size of the domains of the variables. In contrast, the 16
17 `all-different` constraint can be made arc consistent in $O(r^2d)$ time in the worst 17
18 case [116], and can be made bounds consistent in $O(r)$ time [100]. 18

19 Other examples of widely applicable global constraints are the global cardinality 19
20 constraint (`gcc`) [117] and the `cumulative` constraint [2]. A `gcc` over a set of 20
21 variables and values states that the number of variables instantiating to a value must 21
22 be between a given upper and lower bound, where the bounds can be different for 22
23 each value. A `cumulative` constraint over a set of variables representing the time 23
24 where different tasks are performed ensures that the tasks are ordered such that the 24
25 capacity of some resource used at any one time is not exceeded. Both of these types 25
26 of constraint commonly occur in rostering, timetabling, sequencing, and scheduling 26
27 applications. 27

28 4.3 Search 28

29 The main algorithmic technique for solving constraint satisfaction problems is search. 29
30 A search algorithm for solving a CSP can be either complete or incomplete. Complete, 30
31 or systematic algorithms, come with a guarantee that a solution will be found if one 31
32 exists, and can be used to show that a CSP does not have a solution and to find a provably 32
33 optimal solution. Incomplete, or non-systematic algorithms, cannot be used to show a 33
34 CSP does not have a solution or to find a provably optimal solution. However, such 34
35 algorithms are often effective at finding a solution if one exists and can be used to find 35
36 an approximation to an optimal solution. In this section, we survey backtracking and 36
37 local search algorithms for solving CSPs, as well as hybrid methods that draw upon 37
38 ideas from both artificial intelligence (AI) and operations research (OR). Backtracking 38
39 search algorithms are, in general, examples of systematic complete algorithms. Local 39
40 search algorithms are examples of incomplete algorithms. 40
41 41
42 42
43 43

44 4.3.1 Backtracking Search 44

45 A backtracking search for a solution to a CSP can be seen as performing a depth-first 45
46 traversal of a search tree. This search tree is generated as the search progresses. At a 46
47 47

1 node in the search tree, an uninstantiated variable is selected and the node is extended 1
2 where the branches out of the node represent alternative choices that may have to be 2
3 examined in order to find a solution. The method of extending a node in the search 3
4 tree is often called a branching strategy. Let x be the variable selected at a node. The 4
5 two most common branching strategies are to instantiate x in turn to each value in its 5
6 domain or to generate two branches, $x = a$ and $x \neq a$, for some value a in the domain 6
7 of x . The constraints are used to check whether a node may possibly lead to a solution 7
8 of the CSP and to prune subtrees containing no solutions. 8

9 Since the first uses of backtracking algorithms in computing [29, 65], many techni- 9
10 ques for improving the efficiency of a backtracking search algorithm have been 10
11 suggested and evaluated. Some of the most important techniques include constraint 11
12 propagation, nogood recording, backjumping, heuristics for variable and value order- 12
13 ing, and randomization and restart strategies. The best combinations of these tech- 13
14 niques result in robust backtracking algorithms that can now routinely solve large, and 14
15 combinatorially challenging instances that are of practical importance. 15

16 **Constraint propagation during search** 17

18 An important technique for improving efficiency is to maintain a level of local con- 18
19 sistency during the backtracking search by performing constraint propagation at each 19
20 node in the search tree. This has two important benefits. First, removing inconsis- 20
21 tencies during search can dramatically prune the search tree by removing many dead 21
22 ends and by simplifying the remaining subproblem. In some cases, a variable will 22
23 have an empty domain after constraint propagation; i.e., no value satisfies the unary 23
24 constraints over that variable. In this case, backtracking can be initiated as there is no 24
25 solution along this branch of the search tree. In other cases, the variables will have 25
26 their domains reduced. If a domain is reduced to a single value, the value of the vari- 26
27 able is forced and it does not need to be branched on in the future. Thus, it can be 27
28 much easier to find a solution to a CSP after constraint propagation or to show that the 28
29 CSP does not have a solution. Second, some of the most important variable ordering 29
30 heuristics make use of the information gathered by constraint propagation to make ef- 30
31 fective variable ordering decisions. As a result of these benefits, it is now standard for 31
32 a backtracking algorithm to incorporate some form of constraint propagation. 32

33 The idea of incorporating some form of constraint propagation into a backtrack- 33
34 ing algorithm arose from several directions. Davis and Putnam [30] propose unit 34
35 propagation, a form of constraint propagation specialized to SAT. McGregor [99] 35
36 and Haralick and Elliott proposed the *forward checking* backtracking algorithm [68] 36
37 which makes the constraints involving the most recently instantiated variable arc con- 37
38 sistent. Gaschnig [54] suggests *maintaining arc consistency* on all constraints during 38
39 backtracking search and gives the first explicit algorithm containing this idea. Mack- 39
40 worth [95] generalizes Gaschnig's proposal to backtracking algorithms that interleave 40
41 case-analysis with constraint propagation. 41

42 **Nogood recording** 43

44 One of the most effective techniques known for improving the performance of back- 44
45 tracking search on a CSP is to add implied constraints or nogoods. A constraint is 45
46 *implied* if the set of solutions to the CSP is the same with and without the constraint. 46
47 47

1 A *nogood* is a special type of implied constraint, a set of assignments for some subset 1
2 of variables which do not lead to a solution. Adding the “right” implied constraints to 2
3 a CSP can mean that many deadends are removed from the search tree and other dead- 3
4 ends are discovered after much less search effort. Three main techniques for adding 4
5 implied constraints have been investigated. One technique is to add implied constraints 5
6 by hand during the modeling phase. A second technique is to automatically add im- 6
7 plied constraints by applying a constraint propagation algorithm. Both of the above 7
8 techniques rule out local inconsistencies or deadends *before* they are encountered dur- 8
9 ing the search. A third technique is to automatically add implied constraints *after* a 9
10 local inconsistency or deadend is encountered in the search. The basis of this tech- 10
11 nique is the concept of a nogood—a set of assignments that is not consistent with any 11
12 solution. 12

13 Once a nogood for a deadend is discovered, it can be ruled out by adding a con- 13
14 straint. The technique, first informally described by Stallman and Sussman [130], is 14
15 often referred to as nogood or constraint recording. The hope is that the added con- 15
16 straints will prune the search space in the future. Dechter [31] provides the first formal 16
17 account of discovering and recording nogoods. Ginsberg’s [61] dynamic backtracking 17
18 algorithm performs nogood recording coupled with a strategy for deleting nogoods 18
19 in order to use only a polynomial amount of space. Schiex and Verfaillie [125] pro- 19
20 vide the first formal account of nogood recording within an algorithm that performs 20
21 constraint propagation. 21
22

23 **Backjumping** 23

24 Upon discovering a deadend in the search, a backtracking algorithm must uninstantiate 24
25 some previously instantiated variable. In the standard form of backtracking—called 25
26 chronological backtracking—the most recently instantiated variable becomes unin- 26
27 instantiated. However, backtracking chronologically may not address the reason for 27
28 the deadend. In backjumping, the algorithm backtracks to and retracts the decision 28
29 which bears some responsibility for the deadend. The idea is to (sometimes implic- 29
30 itly) record nogoods or explanations for failures in the search. The algorithms then 30
31 reason about these nogoods to determine the highest point in the search tree that can 31
32 safely be jumped to without missing any solutions. Stallman and Sussman [130] were 32
33 the first to informally propose a non-chronological backtracking algorithm—called 33
34 dependency-directed backtracking—that discovered and maintained nogoods in order 34
35 to backjump. The first explicit backjumping algorithm was given by Gaschnig [55]. 35
36 Subsequent generalizations of Gaschnig’s algorithm include Dechter’s [32] graph- 36
37 based backjumping algorithm and Prosser’s [113] conflict-directed backjumping al- 37
38 gorithm. 38
39

40 **Variable and value ordering heuristics** 40

41 When solving a CSP using backtracking search, a sequence of decisions must be made 41
42 as to which variable to branch on or instantiate next and which value to give to the 42
43 variable. These decisions are referred to as the variable and the value ordering. It 43
44 has been shown that for many problems, the choice of variable and value ordering 44
45 can be crucial to effectively solving the problem (e.g., [58, 62, 68]). When solving a 45
46 CSP using backtracking search interleaved with constraint propagation, the domains 46
47 47

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 $S = (t_1, t_2, t_3, \dots)$ is an infinite sequence where each t_i is either a positive integer or infinity. The idea is that a randomized backtracking algorithm is run for t_1 steps. If no solution is found within that cutoff, the algorithm is restarted and run for t_2 steps, and so on until a solution is found.

4.3.2 Local Search

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

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

1 a variable x is chosen that appears in a constraint that is not satisfied. A new value is 1
2 then chosen for x that minimizes the number of constraints that are not satisfied. In 2
3 the successful GSAT algorithm for SAT problems [127], a best-improvement move is 3
4 performed. A variable x is chosen and its value is flipped (true to false or vice versa) 4
5 that leads to the largest reduction in the cost function—the number of clauses that are 5
6 not satisfied. Fourth is the choice of stopping criteria for the algorithm. The stopping 6
7 criteria is usually some combination of an upper bound on the maximum number of 7
8 moves or iterations, a test whether a solution of low enough cost has been found, and 8
9 a test whether the number of iterations since the last (big enough) improvement is too 9
10 large. 10

11 The simplest local search algorithms continually make moves in the graph until all 11
12 moves to neighbors would result in an increase in the cost function. The final node 12
13 then represents the solution to the CSP. However, note that the solution may only 13
14 be a local minima (relative to its neighbors) but not globally optimal. As well, if we 14
15 are solving a satisfaction problem, the final node may not actually satisfy all of the 15
16 constraints. Several techniques have been developed for improving the efficiency and 16
17 the quality of the solutions found by local search. The most important of these include: 17
18 multi-starts where the algorithm is restarted with different starting solutions and the 18
19 best solution found from all runs is reported and threshold accepting algorithms that 19
20 sometimes move to worse cost neighbors to escape local minima such as simulated 20
21 annealing [83] and tabu search [63]. In simulated annealing, worse cost neighbors are 21
22 moved to with a probability that is gradually decreased over time. In tabu search, a 22
23 move is made to a neighbor with the best cost, even if it is worse than the cost of 23
24 the current node. However, to prevent cycling, a history of the recently visited nodes 24
25 called a tabu list is kept and a move to a node is blocked if it appears on the tabu list. 25
26

27 4.3.3 Hybrid Methods 27

28 Hybrid methods combine together two or more solution techniques. Whilst there exist 28
29 interesting hybrids of systematic and local search methods, some of the most promis- 29
30 ing hybrid methods combine together AI and OR techniques like backtracking and 30
31 linear programming. Linear programming (LP) is one of the most powerful techniques 31
32 to have emerged out of OR. In fact, if a problem can be modeled by linear inequali- 32
33 ties over continuous variables, then LP is almost certainly a better method to solve it 33
34 than CP. 34
35

36 One of the most popular approaches to bring linear programming into CP is to 36
37 create a *relaxation* of (some parts of) the CP problem that is linear. Relaxation may 37
38 be both dropping the integrality requirement on some of the decision variables or on 38
39 the tightness of the constraints. Linear relaxations have been proposed for a number 39
40 of global constraints including the `all different`, `circuit` and `cumulative` 40
41 constraints [72]. Such relaxations can then be given to a LP solver. The LP solution can 41
42 be used in a number of ways to prune domains and guide search. For example, it can 42
43 tighten bounds on a variable (e.g., the variable representing the optimization cost). We 43
44 may also be able to prune domains by using reduced costs or Lagrange multipliers. In 44
45 addition, the continuous LP solution may by chance be integral (and thus be a solution 45
46 to the original CP model). Even if the LP solution is not integral, we can use it to guide 46
47 search (e.g., branching on the most non-integral variable). One of the advantages of 47

1 using a linear relaxation is that the LP solver takes a more global view than a CP solver 1
 2 which just makes “local” inferences. 2

3 Two other well-known OR techniques that have been combined with CP are branch 3
 4 and price and Bender’s decomposition. With branch and price, CP can be used to per- 4
 5 form the column generation, identifying variables to add dynamically to the search. 5
 6 With Bender’s decomposition, CP can be used to perform the row generation, gener- 6
 7 ating new constraints (nogoods) to add to the model. Hybrid methods like these have 7
 8 permitted us to solve problems beyond the reach of either CP or OR alone. For ex- 8
 9 ample, a CP based branch and price hybrid was the first to solve the 8-team traveling 9
 10 tournament problem to optimality [43]. 10

11 4.4 Tractability 11

12 Constraint satisfaction is NP-complete and thus intractable in general. It is easy to 12
 13 see how to reduce a problem like graph 3-coloring or propositional satisfiability to a 13
 14 CSP. Considerable effort has therefore been invested in identifying restricted classes of 14
 15 constraint satisfaction problems which are tractable. For Boolean problems where the 15
 16 decision variables have just one of two possible values, Schaefer’s dichotomy theo- 16
 17 rem gives an elegant characterization of the six tractable classes of relations [121]: 17
 18 those that are satisfied by only true assignments; those that are satisfied by only 18
 19 false assignments; Horn clauses; co-Horn clauses (i.e., at most one negated variable); 19
 20 2-CNF clauses; and affine relations. It appears considerably more difficult to charac- 20
 21 terize tractable classes for non-Booleans domains. Research has typically broken the 21
 22 problem into two parts: tractable languages (where the relations are fixed but they can 22
 23 be combined in any way), and tractable constraint graphs (where the constraint graph 23
 24 is restricted but any sort of relation can be used). 24
 25 25
 26 26
 27 27

28 4.4.1 Tractable Constraint Languages 28

29 We first restrict ourselves to instances of constraint satisfaction problems which can 29
 30 be built using some limited language of constraint relations. For example, we might 30
 31 consider the class of constraint satisfaction problems built from just the not-equals 31
 32 relation. For k -valued variables, this gives k -coloring problems. Hence, the problem 32
 33 class is tractable iff $k \leq 2$. 33
 34 34
 35 35

36 Some examples 36

37 We consider some examples of tractable constraint languages. Zero/one/all (or ZOA) 37
 38 constraints are binary constraints in which each value is supported by zero, one or all 38
 39 values [25]. Such constraints are useful in scene labeling and other problems. ZOA 39
 40 constraints are tractable [25] and can, in fact, be solved in $O(e(d + n))$ where e is the 40
 41 number of constraints, d is the domain size and n is the number of variables [149]. 41
 42 This result generalizes the result that 2-SAT is linear since every binary relation on a 42
 43 Boolean domain is a ZOA constraint. Similarly, this result generalizes the result that 43
 44 functional binary constraints are tractable. The ZOA constraint language is maximal in 44
 45 the sense that, if we add any relation to the language which is not ZOA, the language 45
 46 becomes NP-complete [25]. 46
 47 47

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 clones [79–81]. For example, they show how to construct a so-called “indicator” problem that determines whether a constraint language over finite domains is NP-complete or tractable. They are also able to show that the search problem (where we want to find a solution) is no harder than the corresponding decision problem (where we want to just determine if a solution exists or not).

1 Dichotomy results 1

2 As we explained, for Boolean domains, Schaefer’s result completely characterizes the 2
3 tractable constraint languages. For three valued variables, Bulatov has provided a more 3
4 complex but nevertheless complete dichotomy result [16]. Bulatov also has given a 4
5 cubic time algorithm for identifying these tractable cases. It remains an open question 5
6 if a similar dichotomy result holds for constraint languages over any finite domain. 6
7

8 Infinite domains 8

9 Many (but not all) of the tractability results continue to hold if variable domains are 9
10 infinite. For example, Allen’s interval algebra introduces binary relations and composi- 10
11 tions of such relations over time intervals [3]. This can be viewed as a binary constraint 11
12 problem over intervals on the real line. Linear Horn is an important tractable class for 12
13 temporal reasoning. It properly includes the point algebra, and ORD-Horn constraints. 13
14 A constraint over an infinite ordered set is *linear Horn* when it is equivalent to a finite 14
15 disjunction of linear disequalities and at most one weak linear inequality. For example, 15
16 $(X - Y \leq Z) \vee (X + Y + Z \neq 0)$ is linear Horn [88, 84]. 16
17

18 4.4.2 Tractable Constraint Graphs 18

19 We now consider tractability results where we permit any sort of relation but restrict 19
20 the constraint graph in some way. Most of these results concern tree or tree-like struc- 20
21 tures. We need to distinguish between three types of constraint graph: the *primal* 21
22 constraint graph has a node for each variable and edges between variables in the 22
23 same constraint, the *dual* constraint graph has a node for each constraint and edges 23
24 between constraints sharing variables, and the constraint *hypergraph* has a node for 24
25 each variable and a hyperedge between all the variables in each constraint. 25
26

27 Mackworth gave one of the first tractability results for constraint satisfaction prob- 27
28 lems: a binary constraint networks whose primal graph is a tree can be solved in linear 28
29 time [97]. More generally, a constraint problem can be solved in a time that is expo- 29
30 nential in the induced width of the primal graph for a given variable ordering using 30
31 a join-tree clustering or (for space efficiency) a variable elimination algorithm. The 31
32 induced width is the maximum number of parents to any node in the induced graph 32
33 (in which we add edges between any two parents that are both connected to the same 33
34 child). For non-binary constraints, we tend to obtain tighter results by considering the 34
35 constraint hypergraph [67]. For example, an acyclic non-binary constraint problem 35
36 will have high tree-width, even though it can be solved in quadratic time. Indeed, res- 36
37 ults based on hypertree width have been proven to strongly dominate those based on 37
38 cycle cutset width, biconnected width, and hinge width [67]. 38
39

40 4.5 Modeling 40

41 Constraint programming is, in some respects, one of the purest realizations of the 41
42 dream of declarative programming: you state the constraints and the computer solves 42
43 them using one of a handful of standard methods like the maintaining arc consistency 43
44 backtracking search procedure. In reality, constraint programming falls short of this 44
45 dream. There are usually many logically equivalent ways to model a problem. The 45
46
47

1 model we use is often critical as to whether or not the problem can be solved. Whilst 1
 2 modeling a problem so it can be successfully solved using constraint programming is 2
 3 an art, a number of key lessons have started to be identified. 3
 4

5 4.5.1 CP \vee \neg CP 5

6 We must first decide if constraint programming is a suitable technology in which to 6
 7 model our problem, or whether we should consider some other approach like math- 7
 8 ematical programming or simulation. It is often hard to answer this question as the 8
 9 problem we are trying to solve is often not well defined. The constraints of the prob- 9
 10 lem may not have been explicitly identified. We may therefore have to extract the 10
 11 problem constraints from the user in order to build a model. To compound matters, for 11
 12 economic and other reasons, problems are nearly always over-constrained. We must 12
 13 therefore also identify the often conflicting objectives (price, speed, weight, . . .) that 13
 14 need to be considered. We must then decide which constraints to consider as hard, 14
 15 which constraints to compile into the search strategy and heuristics, and which con- 15
 16 straints to ignore. 16
 17

18 Real world combinatorial search problems are typically much too large to solve 18
 19 exactly. Problem decomposition is therefore a vital aspect of modeling. We have to 19
 20 decide how to divide the problem up and where to make simplifying approximations. 20
 21 For example, in a production planning problem, we might ignore how the availability 21
 22 of operators but focus first on scheduling the machines. Having decided on a produc- 22
 23 tion schedule for the machines, we can then attempt to minimize the labor costs. 23

24 Another key concern in modeling a problem is stability. How much variability is 24
 25 there between instances of the problem? How stable is the solution method to small 25
 26 changes? Is the problem very dynamic? What happens if (a small amount of) the data 26
 27 changes? Do solutions need to be robust to small changes? Many such questions need 27
 28 to be answered before we can be sure that constraint programming is indeed a suitable 28
 29 technology. 29
 30

31 4.5.2 Viewpoints 31

32 Having decided to use constraint programming, we then need to decide the variables, 32
 33 their possible domains and the constraints that apply to these variables. The concept 33
 34 of viewpoint [57, 19] is often useful at this point. There are typically several differ- 34
 35 ent viewpoints that we can have of a problem. For example, if we are scheduling the 35
 36 next World Cup, are we assigning games to time slots, or time slots to games? Dif- 36
 37 ferent models can be built corresponding to each of these viewpoints. We might have 37
 38 variables representing the games with their values being time slots, or we might have 38
 39 variables representing the time slots with their values being games. 39
 40

41 A good rule of thumb is to choose the viewpoint which permits the constraints to 41
 42 be expressed easily. The hope is that the constraint solver will then be able to reason 42
 43 with the constraints effectively. In some cases, it is best to use multiple viewpoints and 43
 44 to maintain consistency between them with *channeling* constraints [19]. One common 44
 45 viewpoint is a *matrix model* in which the decision variables form a matrix or array [48, 45
 46 47]. For example, we might need to decide which factory processes which order. This 46
 47 can be modeled with an 0/1 matrix O_{ij} which is 1 iff order i is processed in factory j . 47

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

3 To help specify the constraints, we might introduce auxiliary variables. For exam-
4 ple, in the Golomb ruler problem (prob006 in CSPLib.org), we wish to mark ticks on
5 an integer ruler so that all the distances between ticks are unique. The problem has
6 applications in radio-astronomy and elsewhere. One viewpoint is to have a variable
7 for each tick, whose value is the position on the ruler. To specify the constraint that
8 all the distances between ticks are unique, we can introduce auxiliary variable D_{ij}
9 for the distance between the i th and j th tick [128]. We can then post a global `all-`
10 `different` constraint on these auxiliary variables. It may be helpful to permit the
11 constraint solver to branch on the auxiliary variables. It can also be useful to add im-
12 plied (or redundant) constraints to help the constraint solver prune the search space.
13 For example, in the Golomb ruler problem, we can add the implied constraint that
14 $D_{ij} < D_{ik}$ for $j < k$ [128]. This will help reduce search.

16 4.5.3 Symmetry

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

26 Two common types of symmetry are variable symmetries (which act just on vari-
27 ables), and value symmetries (which act just on values) [21]. With variable symme-
28 tries, there are a number of well understood symmetry breaking methods. For example,
29 many problems can be naturally modeled using a matrix model in which the rows
30 and columns of the matrix are symmetric and can be freely permuted. We can break
31 such symmetry by lexicographically ordering the rows and columns [47]. Efficient
32 constraint propagation algorithms have therefore been developed for such ordering
33 constraints [51, 17]. Similarly, with value symmetries, there are a number of well un-
34 derstood symmetry breaking methods. For example, if all values are interchangeable,
35 we can break symmetry by posting some simple precedence constraints [92]. Alterna-
36 tively, we can turn value symmetry into variable symmetry [47, 93, 114] and then use
37 one of the standard methods for breaking variable symmetry.

40 4.6 Soft Constraints and Optimization

41 It is often the case that, after having listed the desired constraints among the decision
42 variables, there is no way to satisfy them all. That is, the problem is *over-constrained*.
43 Even when all the constraints can be satisfied, and there are several solutions, such
44 solutions appear equally good, and there is no way to discriminate among them. These
45 scenarios often occur when constraints are used to formalize desired properties rather
46 than requirements that cannot be violated. Such desired properties should rather be
47

1 considered as *preferences*, whose violation should be avoided as far as possible. *Soft* 1
 2 *constraints* provide one way to model such preferences. 2

3 4.6.1 Modeling Soft Constraints 3

4 There are many classes of soft constraints. The first one that was introduced concerns 4
 5 the so-called *fuzzy constraints* and it is based on fuzzy set theory [42, 41]. A fuzzy 5
 6 constraint is not a set (of allowed tuples of values to variables), but rather a *fuzzy* 6
 7 *set* [42], where each element has a graded degree of membership. For each assign- 7
 8 ment of values to its variables, we do not have to say whether it belongs to the set or 8
 9 not, but how much it does so. This allows us to represent the fact that a combination 9
 10 of values for the variables of the constraint is partially permitted. We can also say that 10
 11 the membership degree of an assignment gives us the *preference* for that assignment. In 11
 12 fuzzy constraints, preferences are between 0 and 1, with 1 being complete acceptance 12
 13 and 0 being total rejection. The preference of a solution is then computed by taking the 13
 14 minimal preference over the constraints. This may seem awkward in some scenarios, 14
 15 but it is instead very natural, for example, when we are reasoning about critical ap- 15
 16 plications, such as space or medical applications, where we want to be as cautious as 16
 17 possible. *Possibilistic constraints* [122] are very similar to fuzzy constraints and they 17
 18 have the same expressive power: priorities are associated to constraints and the aim 18
 19 is to find an assignment which minimizes the priority of the most important violated 19
 20 constraint. 20
 21 21
 22 22

23 Lack of discrimination among solutions with the same minimal preferences is one 23
 24 of the main drawbacks of fuzzy constraints (the so-called *drowning effect*). To avoid 24
 25 this, one can use *fuzzy lexicographic constraints* [45]. The idea is to consider not just 25
 26 the least preference value, but all the preference values when evaluating a complete as- 26
 27 signment, and to sort such values in increasing order. When two complete assignments 27
 28 are compared, the two sorted preference lists are then compared lexicographically. 28

29 There are situations where we are more interested in the damages we get by not 29
 30 satisfying a constraint rather than in the advantages we obtain when we satisfy it. 30
 31 A natural way to extend the classical constraint formalism to deal with these situations 31
 32 consists of associating a certain penalty or cost to each constraint, to be paid when the 32
 33 constraint is violated. A *weighted constraint* is thus just a classical constraint plus 33
 34 a weight. The cost of an assignment is the sum of all weights of those constraints 34
 35 which are violated. An optimal solution is a complete assignment with minimal cost. 35
 36 In the particular case when all penalties are equal to 1, this is called the MAX-CSP 36
 37 problem [50]. In fact, in this case the task consists of finding an assignment where the 37
 38 number of violated constraints is minimal, which is equivalent to say that the number 38
 39 of satisfied constraints is maximal. 39

40 Weighted constraints are among the most expressive soft constraint frameworks, 40
 41 in the sense that the task of finding an optimal solution for fuzzy, possibilistic, or 41
 42 lexicographic constraint problems can be efficiently reduced to the task of finding an 42
 43 optimal solution for a weighted constraint problem [124]. 43

44 The literature contains also at least two general formalisms to model soft con- 44
 45 straints, of which all the classes above are instances: *semiring-based constraints* [13, 45
 46 14] and *valued constraints* [124]. Semiring-based constraints rely on a simple alge- 46
 47 braic structure which is very similar to a semiring, and it is used to formalize the 47

1 notion of preferences (or satisfaction degrees), the way preferences are ordered, and 1
 2 how to combine them. The minimum preference is used to capture the notion of ab- 2
 3 solute non-satisfaction, which is typical of hard constraints. Similarly for the maximal 3
 4 preference, which can model complete satisfaction. Valued constraints depend on a 4
 5 different algebraic structure, a positive totally ordered commutative monoid, and use 5
 6 a different syntax than semiring-based constraints. However, they have the same ex- 6
 7 pressive power, if we assume preferences to be totally ordered [12]. Partially ordered 7
 8 preferences can be useful, for example, when we need to reason with more than one 8
 9 optimization criterion, since in this case there could be situations which are naturally 9
 10 not comparable. 10

11 Soft constraint problems are as expressive, and as difficult to solve, as constraint 11
 12 optimization problems, which are just constraint problems plus an objective function. 12
 13 In fact, given any soft constraint problem, we can always build a constraint optimiza- 13
 14 tion problem with the same solution ordering, and vice versa. 14
 15

16 4.6.2 Searching for the Best Solution 16

17 The most natural way to solve a soft constraint problem, or a constraint optimization 17
 18 problem, is to use Branch and Bound. *Depth First Branch and bound* (DFBB) per- 18
 19 forms a depth-first traversal of the search tree. At each node, it keeps a lower bound 19
 20 lb and an upper bound ub . The *lower bound* is an underestimation of the violation 20
 21 degree of any complete assignment below the current node. The *upper bound* ub is 21
 22 the maximum violation degree that we are willing to accept. When $ub \leq lb(t)$, the sub- 22
 23 tree can be pruned because it contains no solution with violation degree lower than ub . 23
 24 The time complexity of DFBB is exponential, while its space complexity is linear. The 24
 25 efficiency of DFBB depends largely on its pruning capacity, that relies on the quality 25
 26 of its bounds: the higher lb and the lower ub , the better DFBB performs, since it does 26
 27 more pruning, exploring a smaller part of the search tree. Thus many efforts have been 27
 28 made to improve (that is, to increase) the lower bound. 28

29 While the simplest lower bound computation takes into account just the past 29
 30 variables (that is, those already assigned), more sophisticated lower bounds include 30
 31 contributions of other constraints or variables. For example, a lower bound which con- 31
 32 siders constraints among past and future variables has been implemented in the *Partial*
 33 *Forward Checking* (PFC) algorithm [50]. Another lower bound, which includes con- 33
 34 tributions from constraints among future variables, was first implemented in [143] and 34
 35 then used also in [89, 90], where the algorithm PFC-MRDAC has been shown to give 35
 36 a substantial improvement in performance with respect to previous approaches. An 36
 37 alternative lower bound is presented within the *Russian doll search* algorithm [140]
 38 and in the *specialized* RDS approach [101]. 38
 39

40 4.6.3 Inference in Soft Constraints 40

41 Inference in classical constraint problems consists of computing and adding implied 41
 42 constraints, producing a problem which is more explicit and hopefully easier to solve. 42
 43 If this process is always capable of solving the problem, then inference is said to 43
 44 be complete. Otherwise, inference is incomplete and it has to be complemented with 44
 45 search. For classical constraints, adaptive consistency enforcing is complete while lo- 45
 46 cal consistency (such as arc or path consistency) enforcing is in general incomplete. 46
 47

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

1 weighted or lexicographic constraints, which are not idempotent. For these classes of 1
2 soft constraints, equivalence can still be maintained, compensating the addition of new 2
3 constraints by the “subtraction” of others. This can be done in all *fair* classes of soft 3
4 constraints [24], where it is possible to define the notion of “subtraction”. In this way, 4
5 arc consistency has been extended to fair valued structures in [123, 26]. While equiv- 5
6 alence and termination in polynomial time can be achieved, constraint propagation 6
7 on non-idempotent classes of soft constraints does not assure the uniqueness of the 7
8 resulting problem. 8

9 Several global constraints and their associated algorithms have been extended 9
10 to handle soft constraints. All these proposals have been made using the approach 10
11 of [111] where a soft constraint is represented as a hard constraint with an extra vari- 11
12 able representing the cost of the assignment of the other variables. Examples of global 12
13 constraints that have been defined for soft constraints are the soft `all-different` 13
14 and soft `gcc` [112, 138, 139]. 14

16 4.7 Constraint Logic Programming 16

17 Constraints can, and have been, embedded in many programming environments, but 17
18 some are more suitable than others. The fact that constraints can be seen as relations 18
19 or predicates, that their conjunction can be seen as a *logical and*, and that backtracking 19
20 search is a basic methodology to solve them, makes them very compatible with logic 20
21 programming [94], which is based on predicates, logical conjunctions, and depth-first 21
22 search. The addition of constraints to logic programming has given the *constraint logic* 22
23 *programming* paradigm [77, 98]. 23
24 24
25 25

26 4.7.1 Logic Programs 26

27 Logic programming (LP) [94] is based on a unique declarative programming idea 27
28 where programs are not made of statements (like in imperative programming) nor of 28
29 functions (as in functional programming), but of logical implications between collec- 29
30 tions of predicates. A logic program is thus seen as a logical theory and has the form 30
31 of a set of rules (called *clauses*) which relate the truth value of a literal (the *head* of 31
32 the clause) to that of a collection of other literals (the *body* of the clause). 32
33 33

34 Executing a logic program means asking for the truth value of a certain statement, 34
35 called the *goal*. Operationally, this is done by repeatedly transforming the goal via a 35
36 sequence of *resolution steps*, until we either end up with the empty goal (in this case 36
37 the proof is successful), or we cannot continue and we do not have the empty goal 37
38 (and in this case we have a failure), or we continue forever (and in this case we have 38
39 an infinite computation). Each resolution step involves the unification between a literal 39
40 which is part of a goal and the head of a clause. 40

41 Finite domain CSPs can always be modeled in LP by using one clause for the 41
42 definition of the problem graph and many facts to define the constraints. However, this 42
43 modeling is not convenient, since LP’s execution engine corresponds to depth-first 43
44 search with chronological backtracking and this may not be the most efficient way to 44
45 solve the CSP. Also, it ignores the power of constraint propagation in solving a CSP. 45

46 Constraint logic programming languages extend LP by providing many tools to im- 46
47 prove the solution efficiency using constraint processing techniques. They also extend 47

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).

4.7.2 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*).

4.7.3 LP and CLP Languages

The concept of logic programming [94, 132] was first developed in the 1970s, while the first constraint logic programming language was Prolog II [23], which was designed by Colmerauer in the early 1980s. 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

1 of constraints, and developed the concept of a constraint logic programming scheme 1
2 in 1987 [76]. From then on, several instances of the CLP scheme were developed: Pro- 2
3 log III [22], with constraints over terms, strings, booleans, and real linear arithmetic; 3
4 CLP(R) [75], with constraints over terms and real arithmetics; and CHIP [39], with 4
5 constraints over terms, finite domains, and finite ranges of integers. 5

6 Constraint logic programming over finite domains was first implemented in the 6
7 late 1980s by Pascal Van Hentenryck [70] within the language CHIP [39]. Since then, 7
8 newer constraint propagation algorithms have been developed and added to more re- 8
9 cent CLP(FD) languages, like GNU Prolog [38] and ECLiPSe [142]. 9

10 4.7.4 Other Programming Paradigms 10

11 Whilst constraints have been provided in declarative languages like CLP, constraint- 11
12 based tools have also been provided for imperative languages in the form of libraries. 12
13 The typical programming languages used to develop such solvers are C++ and Java. 13
14 ILOG [1] is one the most successful companies to produce such constraint-based li- 14
15 braries and tools. ILOG has C++ and Java based constraint libraries, which uses many 15
16 of the techniques described in this chapter, as well as a constraint-based configurator, 16
17 scheduler and vehicle routing libraries. 17

18 Constraints have also been successfully embedded within concurrent constraint 18
19 programming [120], where concurrent agents interact by posting and reading con- 19
20 straints in a shared store. Languages which follow this approach to programming are 20
21 AKL [78] and Oz [71]. Finally, high level modeling languages exist for modeling 21
22 constraint problems and specifying search strategies. For example, OPL [135] is a 22
23 modeling language similar to AMPL in which constraint problems can be naturally 23
24 modeled and the desired search strategy easily specified, while COMET is an OO 24
25 programming language for constraint-based local search [136]. CHR (Constraint 25
26 Handling Rules) is instead a rule-based language related to CLP where constraint 26
27 solvers can be easily modeled [52]. 27
28 28
29 29
30 30

31 4.8 Beyond Finite Domains 31

32 Real-world problems often take us beyond finite domain variables. For example, to 32
33 reason about power consumption, we might want a decision variable to range over the 33
34 reals. Constraint programming has therefore been extended to deal with more than just 34
35 finite (or enumerated) domains of values. In this section, we consider three of the most 35
36 important extensions. 36
37 37
38 38

39 4.8.1 Intervals 39

40 The constraint programming approach to deal with continuous decision variables 40
41 is typically via intervals [20, 28, 74, 107]. We represent the domain of a continu- 41
42 ous variable by a set of disjoint intervals. In practice, the bounds on an interval are 42
43 represented by machine representable numbers such as floats. We usually solve a 43
44 continuous problem by finding a covering of the solution space by means of a finite set of 44
45 multi-dimensional interval boxes with some required precision. Such a covering can 45
46 be found by means of a branch-and-reduce algorithm which branches (by splitting an 46
47 47

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].

4.8.2 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.

4.8.3 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].

4.9 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.

1 Domains are therefore incrementally discovered, and the aim is to solve the problem 1
2 even if domains are not completely known. Both solutions and optimal solutions for 2
3 such problems can be obtained in a distributed way without the need to know the en- 3
4 tire domains. This approach can be used within several algorithms, such as the DPOP 4
5 algorithm for distributed dynamic programming [110]. 5
6

7 4.10 Application Areas 7

8 9
10 Constraint programming has proven useful in important applications from industry, 10
11 business, manufacturing, and science. In this section, we survey three general applica- 11
12 tion areas—vehicle routine, scheduling, and configuration—with an emphasis on why 12
13 constraint programming has been successful and why constraint programming is now 13
14 often the method of choice for solving problems in these domains. 14

15 Vehicle Routing is the task of constructing routes for vehicles to visit customers at 15
16 minimum cost. A vehicle has a maximum capacity which cannot be exceeded and the 16
17 customers may specify time windows in which deliveries are permitted. Much work 17
18 on constraint programming approaches to vehicle routing has focused on alternative 18
19 constraint models and additional implied constraints to increase the amount of prun- 19
20 ing performed by constraint propagation. Constraint programming is well-suited for 20
21 vehicle routing because of its ability to handle real-world (or side) constraints. Vehicle 21
22 routing problems that arise in practice often have unique constraints that are particular 22
23 to a business entity. In non-constraint programming approaches, such side constraints 23
24 often have to be handled in an ad hoc manner. In constraint programming a wide vari- 24
25 ety of side constraints can be handled simply by adding them to the core model (see, 25
26 e.g., [86, 108]). 26

27 Scheduling is the task of assigning resources to a set of activities to minimize a 27
28 cost function. Scheduling arises in diverse settings including in the allocation of gates 28
29 to incoming planes at an airport, crews to an assembly line, and processes to a CPU. 29
30 Constraint programming approaches to scheduling have aimed at generality, with the 30
31 ability to seamlessly handle side constraints. As well, much effort has gone into im- 31
32 proved implied constraints such as global constraints, edge-finding constraints and 32
33 timetabling constraints, which lead to powerful constraint propagation. Additional ad- 33
34 vantages of a constraint propagation approach to scheduling include the ability to form 34
35 hybrids of backtracking search and local search and the ease with which scheduling 35
36 or domain specific heuristics can be incorporated within the search routines (see, e.g., 36
37 [6, 18]). 37

38 Configuration is the task of assembling or configuring a customized system from a 38
39 catalog of components. Configuration arises in diverse settings including in the assem- 39
40 bly of home entertainment systems, cars and trucks, and travel packages. Constraint 40
41 programming is well-suited to configuration because of (i) its flexibility in modeling 41
42 and the declarativeness of the constraint model, (ii) the ability to explain a failure to 42
43 find a customized system when the configuration task is over-constrained and to subse- 43
44 quently relax the user's constraints, (iii) the ability to perform interactive configuration 44
45 where the user makes a sequence of choices and after each choice constraint propa- 45
46 gation is used to restrict future possible choices, and (iv) the ability to incorporate 46
47 reasoning about the user's preferences (see, e.g., [4, 85]). 47

4.11 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. *Math. Comput. Modelling*, 17:57–73, 1993.
- [3] J. Allen. Maintaining knowledge about temporal intervals. *Journal ACM*, 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. Bessière, 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, MA, 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. <ftp://ftp.sics.se/pub/SICS-reports/Reports/SICS-T-2002-18-SE.ps.Z>, 2002.
- [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. *Comm. ACM*, 1990.
- [23] **A. Colmerauer.** Prolog II reference manual and theoretical model. Technical report, Groupe Intelligence Artificielle, Université Aix-Marseille 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. <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. Doms, 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. Available from <http://www.dcs.st-and.ac.uk/~apes/reports/apes-36-2001.ps.gz>, 2001. 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, IL, 1974.
- [55] **J. Gaschnig**. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisfying 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, MA, 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. Artificial Intelligence Res.*, 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, MA, 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. 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, et 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*, 19–20, 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. *J. ACM*, 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)*, 2002. Held alongside CP-02.
- [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 5th International Conference on Principles and Practice of Constraint Programming (CP-99)*, pages 303–315, 1999.
- [91] **J. Larrosa**, **E. Morrancho**, 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, MA, 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 sort-edges and alldifferent constraint. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 306–319, Singapore, 2000.

- 1 [101] **P. Meseguer** and **M. Sanchez**. Specializing Russian doll search. In *Principles and Practice of Constraint Programming — CP 2001, LNCS*, vol. 2239, Paphos, Cyprus, November 2001, pages 464–478. Springer-Verlag, 2001.
- 2
3
- 4 [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.
- 5
6
- 7 [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.
- 8
9
- 10 [104] **U. Montanari**. Networks of constraints: Fundamental properties and applications to picture processing. *Inform. Sci.*, 7:95–132, 1974.
- 11
- 12 [105] **U. Montanari**. Networks of constraints: Fundamental properties and applications to picture processing. *Inform. Sci.*, 7:95–132, 1974.
- 13
- 14 [106] **B. Nebel** and **H.-J. Burckert**. Reasoning about temporal relations: A maximal tractable subclass of Allen’s interval algebra. *J. ACM*, 42(1):43–66, 1995.
- 15
- 16 [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.
- 17
18
- 19 [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.
- 20
21
- 22 [109] **A. Petcu** and **B. Faltings**. A scalable method for multiagent constraint optimization. In *Proceedings of the 19th IJCAI*, pages 266–271, 2005.
- 23
- 24 [110] **A. Petcu** and **B. Faltings**. ODPOP: An algorithm for open distributed constraint optimization. In *AAMAS 06 Workshop on Distributed Constraint Reasoning*, 2006.
- 25
26
- 27 [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.
- 28
29
- 30 [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, LNCS*, vol. 2239, Paphos, Cyprus, November 2001, pages 451–463. Springer-Verlag, 2001.
- 31
32
33
- 34 [113] **P. Prosser**. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
- 35
- 36 [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.
- 37
38
- 39 [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.
- 40
41
- 42 [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.
- 43
44
- 45 [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, OR, 1996.
- 46
47

- [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 8th 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, LNCS*, vol. 1894, Singapore, September 2000, pages 411–424. Springer, 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, CA, 1992.
- [128] **B. Smith, K. Stergiou, and T. Walsh.** Using auxiliary variables and implied constraints 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 8th Biennial 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. *J. ACM*, 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.

- 1 [137] **P. van Hentenryck**, **L. Michel**, and **Y. Deville**. *Numerica: A Modeling Language* 1
2 *for Global Optimization*. MIT Press, 1997. 2
- 3 [138] **W.J. van Hoeve**. A hyper-arc consistency algorithm for the soft alldifferent 3
4 constraint. In *Proc. of the Tenth International Conference on Principles and* 4
5 *Practice of Constraint Programming (CP 2004)*, LNCS, vol. 3258. Springer, 5
6 2004. 6
- 7 [139] **W.J. van Hoeve**, **G. Pesant**, and **L.-M. Rousseau**. On global warming (softening 7
8 global constraints). In *Proc. of the 6th International Workshop on Preferences* 8
9 *and Soft Constraints*, Toronto, Canada, 2004. 9
- 10 [140] **G. Verfaillie**, **M. Lemaître**, and **T. Schiex**. Russian doll search. In *Proc. AAAI* 10
11 *1996*, pages 181–187, Portland, OR, 1996. 11
- 12 [141] **M. Vilain** and **H. Kautz**. Constraint propagation algorithms for temporal rea- 12
13 soning. In *Proceedings of 5th National Conference on Artificial Intelligence*, 13
14 pages 377–382. Morgan Kaufmann, 1986. 14
- 15 [142] **M. Wallace**, **S. Novello**, and **J. Schimpf**. ECLiPSe: A platform for constraint 15
16 logic programming. *ICL Systems Journal*, 12(1):159–200, 1997. Available via 16
17 <http://eclipse.crosscoreop.com/eclipse/reports/index.html>. 17
- 18 [143] **R.J. Wallace**. Directed arc consistency preprocessing. In **M. Meyer**, editor. *Se-* 18
19 *lected Papers from the ECAI-94 Workshop on Constraint Processing*, LNCS, 19
20 vol. 923, pages 121–137. Springer, Berlin, 1995. 20
- 21 [144] **T. Walsh**. Search in a small world. In *Proceedings of the Sixteenth International* 21
22 *Joint Conference on Artificial Intelligence*, pages 1172–1177, Stockholm, 1999. 22
- 23 [145] **T. Walsh**. Consistency and propagation with multiset constraints: A formal 23
24 viewpoint. In **F. Rossi**, editor. *9th International Conference on Principles and* 24
25 *Practices of Constraint Programming (CP-2003)*. Springer, 2003. 25
- 26 [146] **M. Yokoo**. Weak-commitment search for solving constraint satisfaction prob- 26
27 lems. In *Proceedings of the 12th AAAI*, pages 313–318, 1994. 27
- 28 [147] **M. Yokoo**. *Distributed Constraint Satisfaction: Foundation of Cooperation in* 28
29 *Multi-Agent Systems*. Springer, 2001. 29
- 30 [148] **M. Yokoo**, **E.H. Durfee**, **T. Ishida**, and **K. Kuwabara**. Distributed constraint sat- 30
31 isfaction for formalizing distributed problem solving. In *Proceedings of the 12th* 31
32 *ICDCS*, pages 614–621, 1992. 32
- 33 [149] **Y. Zhang**, **R. Yap**, and **J. Jaffar**. Functional eliminations and 0/1/all con- 33
34 straints. In *Proceedings of 16th National Conference on Artificial Intelligence*, 34
35 pages 281–290. AAAI Press/The MIT Press, 1999. 35
36 36
37 37
38 38
39 39
40 40
41 41
42 42
43 43
44 44
45 45
46 46
47 47

Proof of Raw Subject Index

1		1
2		2
3		3
4		4
5		5
6		6
7	<hr/>	7
8	Page: 1	8
9	constraint programming	9
10	constraint satisfaction problem	10
11	<hr/>	11
12	Page: 2	12
13	consistency!local	13
14	constraint propagation	14
15	<hr/>	15
16	Page: 3	16
17	consistency!arc consistency	17
18	consistency!hyper arc consistency	18
19	consistency!generalized arc consistency	19
20	consistency!bounds consistency	20
21	constraints!temporal	21
22	consistency!path consistency	22
23	constraint!global	23
24	<hr/>	24
25	Page: 4	25
26	constraint!all-different	26
27	constraint!cardinality	27
28	constraint!cumulative	28
29	search!depth-first	29
30	<hr/>	30
31	Page: 5	31
32	search!branching strategy	32
33	constraint!propagation	33
34	unit propagation	34
35	forward checking	35
36	search!forward checking	36
37	maintaining arc consistency	37
38	search!maintaining arc consistency	38
39	constraint!implied	39
40	<hr/>	40
41	Page: 6	41
42	nogood	42
43	constraint!nogood	43
44	dynamic backtracking	44
45	search!dynamic backtracking	45
46	dependency-directed backtracking	46
47	search!dependency-directed backtracking	47
	backjumping	
	search!backjumping	
	conflict-directed backjumping	
	search!conflict-directed backjumping	
	heuristic!variable ordering	
	heuristic!value ordering	
	<hr/>	
	Page: 7	
	restarts	
	randomization	
	search!randomization and restarts	
	local search	
	search!local	
	<hr/>	
	Page: 8	
	linear programming	
	relaxation	
	constraints!global	
	<hr/>	
	Page: 9	
	Schaefer's dichotomy theorem	
	Horn clauses	
	constraints!ZOA	
	<hr/>	
	Page: 10	
	constraints!row-convex	
	consistency!path consistency	
	<hr/>	
	Page: 11	
	interval algebra	
	constraints!linear Horn	
	linear Horn	
	point algebra	
	constraint graph	
	constraint!graph	
	<hr/>	
	Page: 12	
	constraint!channeling	
	<hr/>	
	Page: 13	
	symmetry	
	symmetry breaking	
	<hr/>	
	Page: 14	
	preferences	
	constraints!soft	
	constraints!fuzzy	
	constraints!possibilistic	
	constraints!weighted	
	constraints!semiring	

Proof of Raw Subject Index

1		constraint logic programming	1
2	Page: 15	logic programming	2
3	constraints!valued		3
4	search!branch and bound	Page: 19	4
5	inference	constraint logic programming	5
6			6
7	Page: 16	Page: 20	7
8	bucket elimination	constraints!temporal	8
9	consistency!adaptive	point algebra	9
10	soft constraint propagation	interval algebra	10
11		constraints!set	11
12	Page: 17		12
13	constraint logic programming	Page: 21	13
14	logic programming!constraint logic	constraints!distributed	14
15	programming		15
16	CLP	Page: 22	16
17	logic programming	constraint programming!applications	17
18	resolution	vehicle routing	18
19	unification	scheduling	19
20		configuration	20
21	Page: 18		21
22			22
23			23
24			24
25			25
26			26
27			27
28			28
29			29
30			30
31			31
32			32
33			33
34			34
35			35
36			36
37			37
38			38
39			39
40			40
41			41
42			42
43			43
44			44
45			45
46			46
47			47