CHAPTER 2

Preference Modeling and Reasoning

Representing and reasoning about preferences is an area of increasing theoretical and practical interest in AI. Preferences and constraints occur in real-life problems in many forms. Intuitively, constraints are restrictions on the possible scenarios: for a scenario to be feasible, all constraints must be satisfied. For example, if we want to buy a PC, we may impose a lower limit on the size of its screen: only PCs that respect this limit will be considered. Preferences, on the other hand, express desires, satisfaction levels, rejection degrees, or costs. For example, we may prefer a tablet PC to a regular laptop, we may desire having a webcam, as well as spending as little as possible. In this case, all PCs will be considered, but some will be preferred to others.

In many real-life optimization problems, we may have both constraints and preferences. For example, in a product configuration problem [171], the producer may impose some constraints (e.g., component compatibility) as well as preferences in the forms of optimization criteria (e.g., minimize the supply time), or also subjective preferences over alternative products expressed in some language of preference statements (e.g., pairwise comparisons).

Preferences and constraints are closely related notions, since preferences can be seen as a form of "relaxed" constraints. For this reason, there are several constraint-based preference modeling frameworks. One of the most general of such frameworks defines a notion of *soft constraints* [134], which extends the classical constraint formalism to model preferences in a *quantitative* way, by expressing several degrees of satisfaction that can be either totally or partially ordered. The term *soft constraints* is used to distinguish this kind of constraints from the classical ones, that are usually called *hard constraint*. However, hard constraints can be seen as an instance of the concept of soft constraints where there are just two levels of satisfaction. In fact, a hard constraint can only be satisfied or violated, while a soft constraint can be satisfied at several levels. When there are both levels of satisfaction and levels of rejection, preferences are usually called *bipolar*, and they can be modeled by extending the soft constraint formalism [21].

Preferences can also be modeled in a *qualitative* (also called *ordinal*) way, that is, by pairwise comparisons. In this case, soft constraints (or their extensions) are not suitable. However, other AI preference formalisms are able to express preferences qualitatively, such as CP-nets [24]. More precisely, CP-nets provide an intuitive way to specify conditional preference statements that state the preferences over the instances of a certain feature, possibly depending on some other features. For example, we may say that we prefer a red car to a blue car if the car is a sports car. CP-nets

and soft constraints can be combined, providing a single environment where both qualitative and quantitative preferences can be modeled and handled.

Specific types of preferences come with their own reasoning methods. For example, *temporal preferences* are quantitative preferences that pertain to the position and duration of events in time. Soft constraints can be embedded naturally in a temporal constraint framework to handle this kind of preference [116, 144].

While soft constraints generalize the classical constraint formalism providing a way to model several kinds of preferences, this added expressive power comes at a cost, both in the modeling task as well as in the solution process. To mitigate these drawbacks, various AI techniques have been adopted. For example, *abstraction theory* [45] has been exploited to simplify the process of finding a most preferred solution of a soft constraint problem [18]. By abstracting a given preference problem, the idea is to obtain a new problem with a kind of preferences which is simpler to handle by the available solver, such that the solution of this simpler problem may give some useful information for the solution process of the original problem. Also, *inference* and *explanation computation* has been applied to preference-based systems to ease the understanding of the result of the solving process. In the context of soft constraints, explaining the result of a solving process means justifying why a certain solution is the best that can be obtained, or why no solution has been found, possibly suggesting minimal changes to the preferences that would provide the given problem with some solutions. For example, explanations have been used to guide users of preference-based configurators [75].

On the modeling side, it may be too tedious or demanding of a user to specify all the soft constraints. *Machine learning techniques* have therefore been used to learn the missing preferences [161, 185]. Alternatively, *preference elicitation* techniques [34], interleaved with search and propagation, have been exploited to minimize the user's effort in specifying the problem while still being able to find a most preferred solution [85].

2.1 CONSTRAINT REASONING

Constraint programming [48, 163] is a powerful paradigm for solving combinatorial search problems that draws on a wide range of techniques from artificial intelligence, computer science, databases, programming languages, and operations research. Constraint programming is currently applied with success to many domains, such as scheduling, planning, vehicle routing, configuration, networks, and bioinformatics. The basic idea in constraint programming is that the user states the constraints and a general purpose *constraint solver* is used to solve them.

2.1.1 CONSTRAINTS

Constraints are just relations, and a *constraint satisfaction problem* (CSP) states which relations should hold among the given decision variables. For example, in scheduling activities in a company, the decision variables might be the starting times and the durations of the activities and the resources needed to perform them, and the constraints might be on the availability of the resources and on their use by a limited number of activities at a time. Another example is configuration, where

constraints are used to model compatibility requirements among components or user's requirements. For example, if we were to configure a laptop, some video boards may be incompatible with certain monitors. Also, the user may impose constraints on the weight and/or the screen size.

Formally, a constraint satisfaction problem (CSP) can be defined by a triple $\langle X, D, C \rangle$, where X is a set of variables, D is a collection of domains, as many as the variables, and C is a set of constraints. Each constraint involves some of the variables in X and a subset of the Cartesian product of their domains. This subset specifies the combination of values of the variables (of the constraint) that satisfy the constraint. A solution for a constraint satisfaction problem is an assignment of all the variables to values in their respective domains such that all constraints are satisfied.

As an example, consider the CSP with four variables $\{x_1, \ldots, x_4\}$, domain $\{1, 2, 3\}$ for all variables, and constraints $x_1 \neq x_2$, $x_2 \neq x_3$, $x_3 \neq x_4$, and $x_2 \neq x_4$. This is usually called a graph coloring problem, since the values in the variable domains can be interpreted as colors and the constraints say that some pairs of variables should have a different color. This CSP has several solutions. One of them is $\langle x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 1 \rangle$: if we assign the variables in this way, all constraints are satisfied. CSPs can be graphically represented by a *constraint graph* where nodes model variables and arcs (or hyperarcs) model constraints. The constraint graph of this CSP is shown in Figure 2.1.

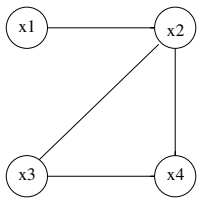

Figure 2.1: The constraint graph of a constraint satisfaction problem.

2.1.2 CONSTRAINT SOLVERS

Constraint solvers take a problem, represented in terms of decision variables and constraints, and find one or more assignments to all the variables that satisfy the constraints. In general, finding such an assignment is NP-hard and is computationally difficult. However, there are *islands of tractability*: for example, if the constraint graph does not have cycles, then finding a solution requires just polynomial time in the size of the CSP [49, 74].

Constraint solvers search the solution space either systematically, as with *backtracking* or *branch-and-bound* algorithms, or they use forms of *local search*, which may be incomplete. The state space of the problem can be represented by a tree, called the *search tree*. Each node in such a tree represents an assignment to some of the variables. The root represents the empty variable assignment, while each leaf represents an assignment to all the variables. Given a node, its children are generated by selecting a variable not yet assigned and choosing as many assignments for this variable as the

number of values in its variable domain. Each arc connecting a node with its children is labelled with one of those values, meaning that this value is assigned to the selected variable. Each path in the search tree (from the root to a node) represents a (possibly incomplete) variable assignment. Since each node has a unique path from the root to it, there is a one-to-one correspondence between tree nodes and variable assignments.

Consider again the graph coloring example above. Its search tree has the root node, plus 3 nodes at the first level (for the 3 possible ways to instantiate the first variables), 9 nodes at the second level, 27 nodes at the third level, and 81 leaves.

Backtracking search. Backtracking search performs a partial depth-first traversal of the search tree. At each step, a new variable is instantiated with a value in its domain, unless there are no values which are compatible with the previous variable assignments. Compatibility here means that the new variable assignment, together with the previous ones, satisfies all the constraints among such variables. We stop when all variables are instantiated, or report failure when all possible instantiations for the variables have been considered and some variables remain uninstantiated. The time complexity of backtracking search is exponential in the number of variables, while its space complexity is linear.

For the graph coloring problem above, if we choose to assign variables in increasing order of their index, and to try domain values from the smallest to the largest, backtracking search would visit the part of the search tree shown in Figure 2.2 a) before finding the first solution of the CSP. The arrows denote the sequence of steps made by the algorithm to visit some nodes of the search tree, assuming it always follows the leftmost alternative. In this case, no backtracking is necessary to find a solution. If instead we have just two colors (thus two values in each variable domain, say 1 and 2), then backtracking search will visit the part of the search tree shown in Figure 2.2 b) before reporting a failure, meaning that the problem has no solution.

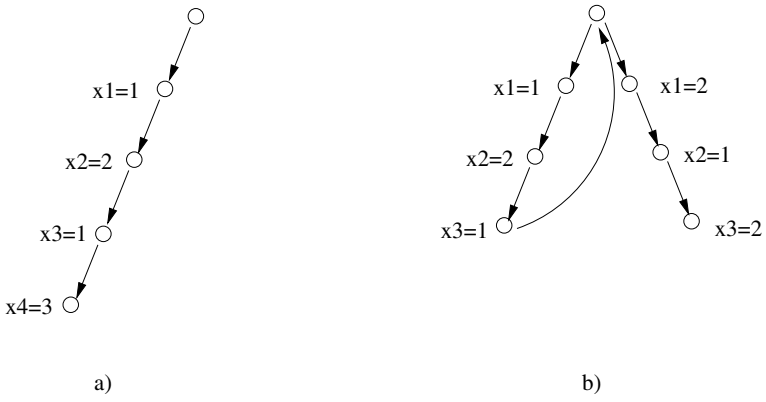

Figure 2.2: Backtracking search for the CSP of Fig. 2.1. Part a) considers three colors per variable, while part b) considers just two colors.

Local search. Local search [104] is an alternative approach for solving computationally hard combinatorial problems. Given a problem instance, the basic idea underlying local search is to start from an initial search position in the space of all possible assignments (typically, a randomly or heuristically generated assignment, which may be infeasible, sub-optimal or incomplete), and to improve iteratively this assignment by means of minor modifications. At each search step, we move to a new assignment selected from a local neighborhood via an heuristic evaluation function. For example, the neighborhood could contain assignments that differ in the value assigned to one variable, and the evaluation function could be the number of satisfied constraints. This process is repeated until a solution is found or a predetermined number of steps is reached. To ensure that the search process does not stagnate, most local search methods make use of random moves: for example, at every step, with a certain probability a random move might be performed rather than the usual move to the best neighbor.

Branch-and-bound search. Rather than trying to satisfy a set of constraints, finding any variable assignment that does not violate any of them, we may want to distinguish among such feasible assignments, and select one that is optimal according to some optimization criterion. Such criteria are usually modeled via an objective function that measures the quality of each solution. The aim is then to find a solution with optimal (say, maximal) quality, where the quality of a solution can be expressed in terms of preferences. For such problems, branch-and-bound is often used to find an optimal solution.

Branch-and-bound search traverses the search tree like backtracking search, except that it does not stop when one solution is found, but it continues looking for possibly better solutions. To do this, it keeps two values during the tree visit: the quality of the best solution found so far (initialized with a dummy solution with the worst possible value) and, given the current node t, an over-estimation of the quality of all complete assignments below t (usually called an *heuristic function*). When such an over-estimation is worse than the quality of the best solution found so far, we can avoid visiting the subtree rooted at t since it cannot contain any solution which is better than the best one found so far. After exhaustively visiting the nodes in this search tree, the last solution stored is optimal according to the objective function.

Consider again the CSP in Fig. 2.1, with three colors for each variable. Suppose the objective function is to maximize $x_1 + x_3$. Let us assume we use the heuristic function $max(D_1) + max(D_3)$, where D_1 and D_3 are the domains of x_1 and x_3, respectively. Then, assuming we try variables and values in the same order as above, branch-and-bound search would visit the part of the search tree shown in Fig. 2.3 before finding an optimal solution, with a value of the objective function of 6. The arrows denote the sequence of steps made by the algorithm to visit the nodes of the search tree, assuming it always follows the leftmost alternative. For example, the algorithm first instantiates all variables down the leftmost branch (thus obtaining the solution $\langle x_1 = 1, x_2 = 2, x_3 = 1, x_4 = 3\rangle$), then it jumps back to a new instantiation for x_3, and finds a second solution $\langle x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 1\rangle$), then it jumps back to the root, and so on. Each node, except the leaves, has a certain

value of the heuristic functions h, while the leaves, which represent complete variable assignments, are associated with a certain value of the objective function f.

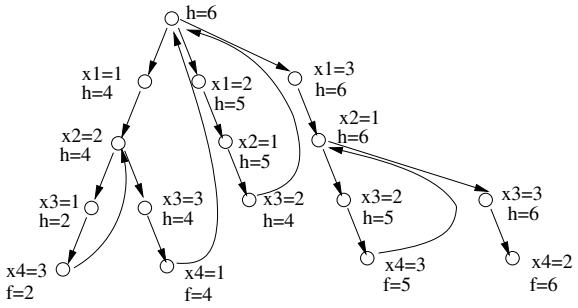

Figure 2.3: Branch-and-bound search for the CSP of Fig. 2.1.

Constraint propagation. Constraint propagation [16] involves modifying the constraints or the variable domains, for example by deleting values from the domains, without changing the solution set. This can be seen as a form of inference. For example, constraint propagation can be used to infer that some values can be eliminated from the domain of a particular variable, based on the other domains and on the constraints, since these values cannot be part of any solution.

The most widely used form of constraint propagation is *arc consistency*. A constraint between two variables x and y is arc-consistent if, for every value a in the domain of x there exists a value b in the domain of y such that the assignment $(x = a, y = b)$ satisfies the constraint, and vice versa.

Given any constraint satisfaction problem, there are many polynomial-time algorithms to achieve arc consistency over all the constraints in the problem. At the end, we have an new problem with the same variables and constraints, but possibly smaller variable domains. Moreover, the new problem has the same set of solutions as the original one.

Consider a constraint between variables x and y, which states that x should be smaller than y, and assume the domain of both variables is $\{1, 2, 3\}$. This constraint is not arc consistent, since when $x = 3$, there is no value in the domain of y which, together with $x = 3$, would satisfy the constraint. Also, if $y = 1$, there is no value for x which, together with $y = 1$, satisfies the constraint. We can make the constraint arc consistent by eliminating 3 from the domain of x and 1 from the domain of y.

Since a backtracking or a branch-and-bound algorithm searching for a solution need to consider the different possible instantiations for the variables from their domains, the search space is smaller if the domains are smaller. Thus, achieving arc consistency (or, more generally, performing some form of constraint propagation) may greatly help the behaviour of the search algorithms by reducing the parts of the search space that need to be visited. Systematic solving methods therefore often interleave search and constraint propagation.

2.2 SOFT CONSTRAINTS

While constraint satisfaction methods have been successfully applied to many real-life combinatorial problems, in some cases the hard constraint framework is not expressive enough. For example, it is possible that after having listed the desired constraints on the decision variables, there is no way to satisfy them all. In this case, the problem is said to be *over-constrained*, and the model needs to be refined to relax some of the constraints. This relaxing process, even when it is feasible, is rarely formalized and is normally difficult and time consuming. Even when all the constraints can be satisfied, we may want to discriminate between the (equally good) solutions. These scenarios often occur when constraints are used to formalize desired properties rather than requirements that cannot be violated. Such desired properties are not faithfully represented by constraints, but they should rather be considered as *preferences* whose violation should be avoided as much as possible.

As an example, consider a typical university timetabling problem which aims at assigning courses and teachers to classrooms and time slots. There are usually many constraints, such as the size of the classrooms, the opening hours of the building, or the fact that the same teacher cannot teach two different classes at the same time. However, there are usually also many preferences, which state, for example, the desires of the teachers (like that he prefers not to teach on Fridays), or university policies (like that it is preferable to use smaller classrooms). If all these preferences are modeled by constraints, it is easy to find scenarios where there is no way to satisfy all of them. In this case, what one would like is to satisfy all hard requirements while violating the preferences as little as possible. Thus, preferences must be represented as different from constraints. Modeling preferences correctly also allows us to discriminate among all the solutions which satisfy the hard constraints. In fact, there could be two timetables which both satisfy the hard requirements, but where one of them better satisfies the desires, and this should be the chosen one. Similar scenarios can be found in most of the typical application areas for constraints, such as scheduling, resource allocation, rostering, vehicle routing, etc.

In general, preferences can be *quantitative* or *qualitative* (e.g., "I like this at level 10" versus "I like this more than that"). Preferences can also be conditional (e.g., "If the main dish is fish, I prefer white wine to red wine"). Preferences and constraints may also co-exist. For example, in a product configuration problem, there may be production constraints (for example, a limited number of convertible cars can be built each month), marketing preferences (for example, that it would be better to sell the standard paint types), whilst the user may have preferences of various kind (for example, that if it is a sport car, she prefers red).

To cope with some of these scenarios, hard constraints have been generalized in various ways in the past decades. The underlying observation of such generalizations is that hard constraints are relations, and thus they can either be satisfied or violated. Preferences need instead a notion that has several levels of satisfiability. In the early '90s, several *soft constraint* formalisms were proposed that generalize the notion of constraint to allow for more than two levels of satisfiability.

2.2.1 SPECIFIC SOFT CONSTRAINT FORMALISMS

Here we list some specific soft constraint formalisms, among which fuzzy constraints, weighted constraints, and probabilistic constraints.

Fuzzy constraints. Fuzzy constraints (see, for instance, [59, 170]) use (discretized) preference values between 0 and 1. The quality of a solution is the minimum preference associated with constraints in that solution. The aim is then to find a solution whose quality is highest. Since only the minimum preference is considered when evaluating a variable assignment, fuzzy constraints suffer from the so-called "drowning effect" (that is, the worst level of satisfiability "drowns" all the others). This is typical of a pessimistic approach, that can be useful or even necessary in critical applications, such as in the medical or aerospace domain. However, this approach is too pessimistic in other domains. For this reason, *lexicographic constraints* were introduced [70] to obtain a more discriminating ordering of the solutions: to order two solutions, we compare lexicographically the ordered sequence of all the preference values given by the constraints to those two solutions. In this way, solutions with different minimum preference values are ordered as in the classical fuzzy constraint setting, but also solutions with the same minimum preference (that would be equally preferred in fuzzy constraints) can be discriminated.

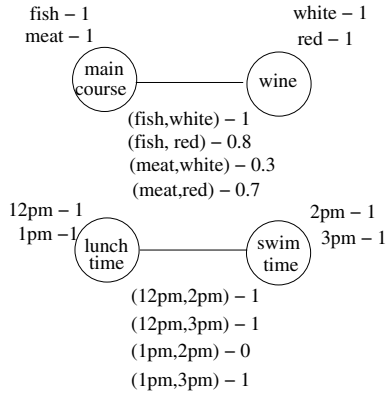

Figure 2.4: A fuzzy constraint problem.

An example of a fuzzy constraint problem can be seen in Figure 2.4. We are deciding on what to have for lunch and when to go swimming. The fuzzy CSP has four variables (represented by the circles), each with two values. For example, wine can be either red or white. There are three constraints, represented by solid arrows, and giving a preference value between 0 and 1 to each assignment of the variables of the constraint. If we consider the variable assignment ⟨main course = meat, wine = white, lunch time = 1pm, swim time = 2pm⟩, then its preference value is 0, since we have to take the minimum contribution of all constraints, and the constraint between lunch time and swim time gives preference value 0. If instead we consider the variable assignment ⟨main course

= fish, wine = white, lunch time = 12pm, swim time = 2pm⟩, then its preference value is 1, since all constraints are satisfied at level 1. Thus, this second assignment is preferred to the first one. Since the scale of preference values is between 0 and 1, with 1 modeling the highest preference, this second assignment is also optimal.

Probabilistic constraints. Another extension of hard constraints are *probabilistic constraints* [69], where, in the context of an uncertain model of the real world, each constraint is associated with the probability of being present in the real problem. Solutions are then associated with their conjoint probability (assuming independence of the constraints), and the aim is to find a solution with the highest probability.

Weighted constraints. In many real-life problems, it is often natural to talk about costs or penalties, rather than preferences. In fact, there are situations where we are more interested in the damage caused by not satisfying a constraint rather than in the advantage we obtain when we satisfy it. For example, when we want to use tolled highways to get to a different city, we may want to find the itinerary with the minimum overall cost. Also, when we want to put together a PC by buying all of its parts, we want to find the combination of parts which has the minimum price. A natural way to extend the hard constraint formalism to deal with these situations consists of associating a certain penalty or cost to each constraint, to be paid when the constraint is violated.

In *weighted constraints*, each constraint is associated with a weight, and the aim is to find a solution for which the sum of the weights of the satisfied constraints is maximal. A very useful instance of weighted constraints are MaxCSPs, where weights are just 0 or 1 (0 if the constraint is violated and 1 if it is satisfied). In this case, we want to satisfy as many constraints as possible.

It is easy to transform probabilities into additive costs by taking their logarithm, and this allows us to reduce any probabilistic constraint instance to a weighted constraint instance [174]. Notice, however, that probabilistic constraints are similar to fuzzy constraints, since in both cases the values associated to the constraints are between 0 and 1, and better solutions have higher values. The main difference is that, while in fuzzy constraints the evaluation of a solution is the minimum value (over all the constraints), in probabilistic constraints it is the product of all the values.

Weighted constraints are among the most expressive soft constraint frameworks, in the sense that the task of finding an optimal solution for possibilistic, lexicographic or probabilistic frameworks can be efficiently (that is, in polynomial time) reduced to the task of finding an optimal solution for a weighted constraint instance [174].

While fuzzy, lexicographic, and probabilistic constraints were defined to model real-life situations that could not be faithfully modeled via hard constraints, weighted constraints and MaxCSPs have mainly been defined to address over-constrained problems, where there are so many constraints that the problem has no solution. In this case, the goal is to satisfy as many constraints as possible, possibly giving them some importance levels.

2.2.2 GENERAL SOFT CONSTRAINT FORMALISMS

Partial constraint satisfaction. Over-constrained problems have been the motivation also for the definitions of the first general framework to extend hard constraints in order to model preferences, called *partial constraint satisfaction* [78]. In order to find a solution for an over-constrained hard constraint problem, partial constraint satisfaction tries to identify another hard constraint problem which is both consistent and as *close* as possible to the original. The space of problems considered to find this consistent network is defined using constraint relaxations (by, for example, forgetting constraints or adding extra authorized combinations to the existing constraints) together with a specific metric, which is needed to identify the nearest problem. MaxCSPs are then just an instance of partial CSPs where the metric is simply the number of satisfied constraints.

Semiring-based soft constraints. Another general constraint-based formalism for modeling preferences is the *semiring-based formalism* [19, 134], which encompasses most of the previous extensions. Its aim is to provide a single environment where properties of specific preference formalisms (such as fuzzy or weighted constraints) can be proven once and for all, and then inherited by all the instances. At the technical level, this is done by introducing a structure representing the levels of satisfiability of the constraints. Such a structure is just a set with two operations: one (written +) is used to generate an ordering over the levels, while the other one ($\times$) is used to define how two levels can be combined and which level is the result of such combination. Moreover, two elements of the set are the *best* and *worst* element among all preference levels. Because of the properties required on such operations, this structure can be defined as a variant of the notion of semiring. More precisely: a *c-semiring* is a 5-tuple $\langle E, +_s, \times_s, \mathbf{0}, \mathbf{1}\rangle$ such that:

- E is a set containing at least $\mathbf{0}$ and $\mathbf{1}$;
- $+_s$ is a binary operator closed in E, associative, commutative and idempotent, for which $\mathbf{0}$ is a neutral element and $\mathbf{1}$ an annihilator;
- $\times_s$ is a binary operator closed in E, associative and commutative, for which $\mathbf{0}$ is an annihilator and $\mathbf{1}$ a neutral element;
- $\times_s$ distributes over $+_s$.

In words, the minimum level $\mathbf{0}$ is used to capture the notion of absolute non-satisfaction, which is typical of hard constraints. Since a single complete unsatisfaction is unacceptable, $\mathbf{0}$ must be an annihilator for $\times_s$. This means that, when combining a completely violated constraint with a constraint which is satisfied at some level, we get a complete violation. Conversely, a complete satisfaction should not hamper the overall satisfaction degree, which explains why $\mathbf{1}$ is a neutral element for $\times_s$. In fact, this means that, when we combine a completely satisfied constraint and a constraint which is satisfied at some level l, we get exactly l. Moreover, since the overall satisfaction should not depend on the way elementary satisfactions are combined, combination (that is, $\times_s$) is required to be commutative and associative.

To define the ordering over the preference levels, operator $+_s$ is used: if $a +_s b = b$, it means that b is preferred to a, and we write this as $b \succcurlyeq_s a$. If $a +_s b = c$, and c is different from both a and b, a and b are incomparable. To make sure that this ordering has the right properties, operator $+_s$ is required to be associative, commutative and idempotent. This generates a partial order, and more precisely a lattice. In all cases, $a +_s b$ is the least upper bound of a and b in the lattice $< E, \succcurlyeq_s >$. The fact that **1** (resp. **0**) is a neutral (respectively, annihilator) element for $+_s$ follows from the fact that it is a maximum (respectively, minimum) element for $\succcurlyeq_s$.

Finally, assume that a is better than b, and consider two complete assignments, one that satisfies a constraint at level a and the other one that satisfies the same constraint at level b. Then, if all the other constraints are satisfied equally by the two assignments, it is reasonable to expect that the assignment satisfying at level a is overall better than the one satisfying at level b. For comparable a and b, this is equivalent to saying that $\times_s$ distributes over $+_s$. In a c-semiring, this property is required in all cases, even if a and b are incomparable.

This gives the *semiring-based soft constraint problems* (SCSPs), where constraints have several levels of satisfiability, that are (totally or partially) ordered according to the semiring structure. More formally, a *semiring constraint problem* is a tuple $\langle X, D, C, S \rangle$ where:

- $X = \{x_1, \ldots, x_n\}$ is a finite set of n variables.
- $D = \{D_1, \ldots, D_n\}$ is the collection of the domains of the variables in X such that D_i is the domain of x_i.
- $S = \langle E, +_s, \times_s, \mathbf{0}, \mathbf{1} \rangle$ is a c-semiring.
- C is a finite set of soft constraints. A soft constraint is a function f on a set of variables $V \subseteq X$, called the scope of the constraint, such that f maps assignments (of variables in V to values in their domains) to semiring values, that is, $f : \prod_{x_i \in V} D_i \rightarrow E$. Thus, a soft constraint can be viewed as a pair $\langle f, V \rangle$ also written as f_V.

Fuzzy, lexicographic, probabilistic, weighted, and MaxCSPs are all instances of the semiring-based framework. Indeed, even hard constraints are an instance of the semiring-based framework. With hard constraints, are either satisfied or violated, and this can be modeled by having two levels of preferences (true and false). Moreover, a complete variable assignment is a solution of a CSP when all constraints are satisfied, and this can be modeled by choosing the logical and operator to combine constraints. The logical or instead models the ordering between the two preference levels, ensuring that true is better than false. Thus, for hard constraints, the c-semiring is $= \langle \{false, true\}, \vee, \wedge, false, true \rangle$. For fuzzy constraints, the c-semiring is $\langle$ [0, 1], max, min, 0, 1$\rangle$. In fact, preference levels are between 0 and 1, with 1 being the best and 0 being the worst value (which is achieved by choosing the max operator to define the ordering over preference levels), and preference combination is performed via the min operator (only the worst preference value is used to assess the quality of a variable assignment).

Valued constraints. Similar to semiring-based constraints, *valued constraints* were also introduced as a general formalism to model constraints with several levels of satisfiability [174]. Valued constraints are indeed very similar to semiring-based soft constraints, the main difference being that the preference values cannot be partially ordered [20].

The possibility of partially ordered preference values can be useful is several scenarios. For example, when the preference values are tuples of elements (such as when we need to combine several optimization criteria), it is natural to have a Pareto-like approach in combining such criteria, and this leads to a partial order. Also, even if we have just one optimization criterion, we may want to insist on declaring some preference values as incomparable because of what they model. In fact, the elements of the semiring structure do not need to be numbers, but can be any kind of object, that we associate to variable assignments. If, for example, they are all the subsets of a certain set, then we can have a partial order over such items under subset inclusion.

2.2.3 COMPUTATIONAL PROPERTIES OF SOFT CONSTRAINTS

Solving a semiring-based problem is a difficult task since semiring-based constraints properly generalize hard constraints, where the problem is already NP-hard. However, as with hard constraints, there are easy cases also for soft constraints such as when the constraint graph does not have cycles [47].

If the computation of $a \times_s b$ and $a +_s b$ need polynomial time in the size of their arguments (that is, a and b), then deciding if the consistency level of a problem is higher than a given threshold is an NP-complete task. Under the same conditions, given two solutions of a soft constraint problem, checking whether one is preferable to the other is polynomial: we simply compute the desirability values of the two solutions and compare them in the preference order.

Many search techniques have been developed to solve specific classes of soft constraints, like fuzzy or weighted constraints. However, all have an exponential worst-case complexity. Systematic approaches like backtracking search and constraint propagation can be adapted to soft constraints. For example, branch-and-bound can use the preference values of a subset of the constraints to compute the bound for pruning the search tree. Also, constraint propagation techniques like arc consistency can be generalized to certain classes of soft constraints, including fuzzy and weighted constraints [134].

2.2.4 BIPOLAR PREFERENCES

Bipolarity is an important topic in several fields, such as psychology and multi-criteria decision making, and it has recently attracted interest in the AI community, especially in argumentation [5], qualitative reasoning [57, 58], and decision theory [120]. Bipolarity in preference reasoning can be seen as allowing one to state both degrees of satisfaction (that is, *positive* preferences) and degrees of rejection (that is, *negative* preferences).

Positive and negative preferences can be thought as symmetric concepts, and thus one might try to deal with them using the same operators. However, this may not model what one usually expects

in real scenarios. For example, if we have a dinner menu with fish and white wine, and we like them both, then having both should be more preferred than having just one of them. On the other hand, if we don't like any of them, then the preference of having them both should be smaller than the preferences of having each of them alone. In other words, the combination of positive preferences should produce a higher (positive) preference, while the combination of negative preferences should give us a lower (negative) preference. Thus, we need operators with different properties to combine positive and negative preferences.

When dealing with both kinds of preferences, it is natural to express also *indifference*, which means that we express neither a positive nor a negative preference. For example, we may say that we like peaches, we don't like bananas, and we are indifferent to apples. Then, a desired behavior of indifference is that, when combined with any preference (either positive or negative), it should not influence the overall preference. For example, if we like peaches and we are indifferent to apples, a dish with peaches and apples should have overall a positive preference.

Moreover, we also want to be able to combine positive with negative preferences. The most natural and intuitive way to do so is to allow for *compensation*. Comparing positive against negative aspects and compensating them with respect to their strength is one of the core features of decision-making processes, and it is, undoubtedly, a tactic universally applied to solve many real life problems.

Positive and negative preferences might seem as just two different criteria to reason with, and thus techniques such as those usually adopted by multi-criteria optimization [61], such as Pareto-like approaches, could appear suitable for dealing with them. However, this interpretation would hide the fundamental nature of bipolar preferences, that is, positive preferences are naturally the opposite of negative preferences.

Semiring-based constraints can model only negative preferences, since in this framework preference combination returns lower preferences. However, they have been generalized to model both positive and negative preferences [21], as well as indifference and preference compensation. This is done by adding to the usual c-semiring structure another algebraic structure to model positive preferences, and by setting the highest negative preference to coincide with the lowest positive preference, to link the two structures and to model indifference. To find optimal solutions of bipolar preference problems, it is possible to adapt the notions of soft constraint propagation and branch-and-bound search.

Bipolarity has also been considered in qualitative preference reasoning [14, 15], where fuzzy preferences model the positive knowledge and negative preferences are interpreted as violations of constraints. Precedence is given to negative preference optimization, and positive preferences are used to distinguish among the optimal solutions found in the first phase, thus not allowing for compensation. Another approach [95] considers totally ordered unipolar and bipolar preference scales and defines an operator, the *uninorm*, which can be seen as a restricted form of compensation.

2.3 CP-NETS

Soft constraints are one of the main tools for representing and reasoning about preferences in constraint satisfaction problems. However, they require specifying a semiring value for each variable assignment in each constraint. In many applications, it may be more natural for users to express preferences via generic qualitative (usually partial) preference relations over variable assignments. For example, it is often more intuitive for the user to state "I prefer red wine to white wine", rather than "Red wine has preference 0.7 and white wine has preference 0.4" (with the assumption that a higher preference value expresses higher desirability). Although the former statement provides us with less information, it does not require the careful selection of preference values for (possibly partial) variable assignments, which is instead required in soft constraints.

CP-nets [23, 24] (that is, Conditional Preference networks) are graphical models for representing and reasoning about certain types of qualitative preference statements, interpreted under the *ceteris paribus (cp)* assumption. For instance, under the ceteris paribus interpretation, the statement *"I prefer red wine to white wine if meat is served"* asserts that, given two meals that differ *only* in the kind of wine served *and* both containing meat, the meal with a red wine is preferred to the meal with a white wine. Observe that this interpretation corresponds to a "least committing" interpretation of the information provided by the user, and many philosophers (see [100] for an overview) and AI researchers [55] have argued for this interpretation of preference assertions. To emphasize the ceteris paribus interpretation, such statements are usually called cp-statements.

Informally, each CP-net compactly captures the preference relation induced by a set of such (possibly conditional) cp-statements. Structurally, CP-nets bear some similarity to Bayesian networks [118], as both utilize directed graphs where each node stands for a variable (usually called a *feature* in the CP-net literature). Thus, there is a set of features $\{X_1, \ldots, X_n\}$ with finite, discrete domains $\mathcal{D}(X_1), \ldots, \mathcal{D}(X_n)$, which play the same role as the variables in soft constraints. Another similarity between CP-nets and Bayesian networks is that graphical structure in both models relies on a notion of independence between the variables: Bayesian nets utilize the notion of probabilistic independence, while CP-nets utilize the notion of preferential independence [71].

2.3.1 CONDITIONAL PREFERENCES

During preference elicitation, for each feature X_i the user is asked to specify a set of *parent* features $Pa(X_i)$, the values of which affect her preferences over the values of X_i. This information is used to create the directed *dependency graph* of the CP-net in which each node X_i has $Pa(X_i)$ as its immediate predecessors. Given this structural information, the user is asked to specify explicitly her preference over the values of X_i for *each complete assignment* of $Pa(X_i)$, and this preference is assumed to take the form of a total [23] or partial [24] order over $\mathcal{D}(X_i)$. These conditional preferences over the values of X_i are captured by a *conditional preference table* $CPT(X_i)$, which is annotated with the node X_i in the CP-net. Each cp-preference statement has therefore the form $x_1 = v_1, \ldots, x_n = v_n : y = w_1 \succ \ldots \succ y = w_k$, where y is the variable for which we are specifying our preferences, $\{w_1, \ldots, w_k\}$ is the domain of y, $x_1, \ldots, x_n$ are the parents of y (that is, the variables

on which y depends), and each v_i is in the domain of x_i, for $i = 1, \ldots, n$. Here $\succ$ means *preferred to*.

As an example, consider the CP-net in Figure 2.5 a). This CP-net has three features, which are graphically represented by nodes, of which two (*main course* and *fruit*) do not depend on any other feature, while *wine* depends on *main course*. The dependency is graphically represented by the arrow. For the independent features, we just have a total order over their values (such as *fish* $\succ$ *meat* for *main course*), while for the dependent feature we have a total order for each assignment of the feature it depends upon (thus one for *main course* = *fish* and another one for *main course* = *meat*). These total orders are shown close to the node representing the feature. Thus, this CP-net includes the following cp-statements: *fish is preferred to meat*; *peaches are preferred to strawberries*; *white wine is preferred to red wine if fish is served; otherwise, red wine is preferred to white wine*.

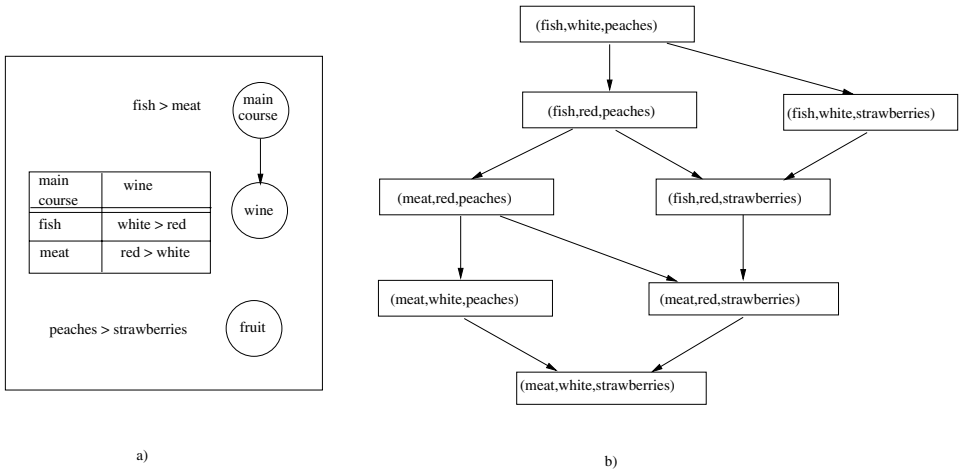

Figure 2.5: A CP-net and its outcome ordering.

2.3.2 PREFERENCE ORDERING

The ceteris paribus interpretation. A CP-net induces an ordering over the variable assignments, which is based on the *ceteris paribus* interpretation of the conditional preference statements. *Ceteris paribus* in Latin means *all else being equal*. This is intended to mean that two variable assignments that differ for the value of one variable, while all else is the same, are always ordered (that is, one is preferred to the other one). To understand which is preferred to which, we just have to look at the CP-table of the changing variable. For example, in the CP-net of Figure 2.5 a), assignment $\langle fish, white, peaches \rangle$ is preferred to $\langle fish, red, peaches \rangle$ since the CP-table of variable *wine* tells us that, when there is fish, we prefer white wine to red wine. In order to compute the entire ordering over all variable assignments, we use the concept of a *worsening flip*.

Worsening flips. A worsening flip is a change in the value of a single feature to a value which is less preferred according to a cp-statement for that feature. For example, in the CP-net in Figure 2.5 a), moving from ⟨*fish,red,peaches* ⟩ to ⟨*meat,red,peaches* ⟩ is a worsening flip, since, according to the CP table for the main course, fish is preferred to meat. Another worsening flip is moving from ⟨*fish,white,berries* ⟩ to ⟨*fish,red,berries* ⟩, since, according to the CP table of *wine*, white is preferred to red in the presence of fish.

Outcome ordering. In CP-nets, complete assignments of all the features to values in their domains are called *outcomes*. An outcome α is preferred to an outcome β, written $\alpha \succ \beta$, if and only if there is a chain of worsening flips from α to β. This definition induces a strict preorder over the outcomes, which defines the so-called *induced graph*, where nodes represent outcomes and directed arcs represent worsening flips to [24]. An outcome is optimal if there is no other outcome which is preferred to it in this preorder. Notice that CP-nets cannot model any preorder over the outcomes: for example, two outcomes that differ in the value of only one feature are always ordered (that is, they are never incomparable).

Figure 2.5 b) shows the outcome preference ordering for the CP-net in Figure 2.5 a). Outcomes appearing higher in the figure are more preferred. The preference ordering is the transitive closure of the relation given by the arrows of the figure. For example, we can see that the outcome ⟨*fish, red, strawberries* ⟩ is preferred to ⟨*meat, white, strawberries*⟩ since there is a chain of worsening flips from the first outcome to the second. On the other hand, outcomes ⟨*fish, red, peaches* ⟩ and ⟨*fish, white, berries* ⟩ are incomparable since there is no chain of worsening flips from any one to the other. We can also see that outcome ⟨*fish, white, peaches* ⟩ is the optimal outcome.

2.3.3 COMPUTATIONAL PROPERTIES

From the point of view of reasoning about preferences, several types of queries can be answered using CP-nets. First, given a CP-net N one might be interested in finding an optimal assignment to the features of N. In general, finding the optimal outcome of a CP-net is NP-hard. However, in acyclic CP-nets, there is only one optimal outcome and this can be found in linear time [23]: we simply sweep through the CP-net, following the arrows in the dependency graph and assigning at each step the most preferred value in the current feature's preference table. Second, one might be interested in comparing two assignments. Determining if one outcome is preferred to another (that is, a dominance query) is PSPACE-complete for acyclic CP-nets, even if feature domains have just two values [94]: intuitively, to check whether an outcome is preferred to another one, one should find a chain of worsening flips, and these chains can be exponentially long.

Various extensions of the notion of CP-net have been defined and studied over the years. For example, TCP-nets introduce the notion of trade-offs in CP-nets based on the idea that variables may have different importance levels [27]. Also, efficient dominance testing have been defined for preference languages more general than CP-nets [191].

2.4 SOFT CONSTRAINTS VS. CP-NETS

It is clear, at this point, that soft constraints and CP-nets have complementary advantages and drawbacks. CP-nets allow us to represent conditional and qualitative preferences, but dominance testing is expensive. On the other hand, soft constraints allow us to represent both hard constraints and quantitative preferences, and to have a cheap dominance test. We will now formally compare their expressive power and investigate whether one formalism can be approximated by the other one.

2.4.1 EXPRESSIVENESS COMPARISON

We say that a formalism B is at least as expressive as a formalism A if and only if from a problem P expressed using A it is possible to build in polynomial time (in the size of P) a problem expressed using B such that the optimal solutions are the same.

If we apply this definition to classes of soft constraints, we get, for example, that fuzzy CSPs and weighted CSPs are at least as expressive as hard constraints. If instead we use it to compare CP-nets and various instances of soft constraints, we see that hard constraints are at least as expressive as CP-nets. In fact, given any CP-net, we can obtain in polynomial time a set of constraints whose solutions are the optimal outcomes of the CP-net [26]: it is enough to take, for each preference statement of the form $x_1 = v_1, \ldots, x_n = v_n : y = w_1 \succ \ldots \succ y = w_k$, the constraint $x_1 = v_1, \ldots, x_n = v_n \Rightarrow y = w_1$. On the contrary, there are some hard constraint problems for which it is not possible to build in polynomial time a CP-net with the same set of optimals. This means that, if we are just interested in the set of optimal solutions, hard constraints are at least as expressive as CP-nets.

However, we can be more fine-grained in the comparison, and say that a formalism B is at least as expressive as a formalism A if and only if from a problem P expressed using A it is possible to build in polynomial time (in the size of P) a problem expressed using B such that the orderings over solutions are the same. Here not only we must maintain the set of optimal solutions, but also the rest of the ordering over the solutions. In this case, CP-nets and soft constraints are incomparable, since each can do something that the other one cannot do. More precisely, dominance testing (that is, comparing two complete assignments to decide which is preferred (if either)) is a difficult task in CP-nets [52]. On the contrary, it is polynomial in soft constraints (if we assume $\times_s$ and $+_s$ to be polynomially computable). Thus, unless P=NP, it is not possible to generate in polynomial time a soft constraint network which has the same solution ordering of a given CP-net. On the other hand, given any soft constraint network, it is not always possible to generate a CP-net with the same ordering. This follows from the fact that CP-nets cannot represent all possible preorders, but only some of them. For example, as we noted above, no CP-net can generate an outcome ordering where two solutions differing in just one flip are unordered. On the other hand, soft constraint networks can represent any partial order over solutions. Thus, when we are interested in the solution ordering, CP-nets and soft constraints are incomparable. This continues to hold also when we augment the CP-nets with set of hard constraints for filtering out infeasible assignments.

2.4.2 APPROXIMATING CP-NETS VIA SOFT CONSTRAINTS

It is possible to approximate a CP-net ordering via soft constraints, achieving tractability while sacrificing precision to some degree. Different approximations can be characterized by how much of the original ordering they preserve, the time complexity of generating the approximation, and the time complexity of comparing outcomes in the approximation. It is often desirable that such approximations are information preserving; that is, what is ordered in the given ordering is also ordered in the same way in the approximation. In this way, when two outcomes are ordered in the approximation, we can infer that they are either ordered or incomparable in the CP-net. If instead two outcomes are in a tie or incomparable in the approximation, they are surely incomparable in the CP-net. Another desirable property of approximations is that they preserve the ceteris paribus property. CP-nets can be approximated by both weighted and fuzzy constraints. In both cases, the approximation is information preserving and satisfies the ceteris paribus property [53].

2.4.3 CP-NETS AND HARD CONSTRAINTS

CP-nets provide a very intuitive framework to specify conditional qualitative preferences, but they do not allow for hard constraints. However, real-life problems often contain both constraints and preferences. While hard constraints and quantitative preferences can be adequately modeled and handled within the soft constraint formalisms, when we have both hard constraints and qualitative preferences, we need to either extend the CP-net formalism [22] or at least develop specific solution algorithms. Notice that reasoning with both constraints and preferences is difficult as often the most preferred outcome is not feasible, and not all feasible outcomes are equally preferred. When we have preferences expressed via a CP-net and a set of hard constraints, it is however possible to obtain all the optimal outcomes by solving a set of hard *optimality constraints*. While, in general, the proposed algorithm needs to perform expensive dominance tests, there are special cases in which this can be avoided [157].

2.5 TEMPORAL PREFERENCES

Preference formalisms can be adapted to deal with specific kinds of preferences. For example, temporal preferences lets us express preferences over the time and duration of events. Soft constraints have been used to model such temporal preferences. Reasoning about time is a core issue in many real-life problems, such as planning and scheduling for production plants, transportation, and space missions. Several approaches have been proposed to reason about temporal information. The formalisms and approaches based on *temporal constraints* have been among the most successful in practice.

In temporal constraint problems, variables either represent instantaneous events, such has "when a plane takes off", or temporal intervals, such as "the duration of the flight". Temporal constraints allow us to put temporal restrictions either on when a given event should occur, e.g., "the plane must take off before 10am", or on how long a given activity should last, e.g., "the flight should not last more than two hours".

Several quantitative and qualitative constraint-based temporal formalisms have been proposed, stemming from pioneering works by Allen [4] and by Dechter, Meiri, and Pearl [50]. A qualitative temporal constraint defines which temporal relations, (e.g., *before, after, during*), are allowed between two temporal events or intervals, representing two activities. For example, one could say "fueling must be completed before boarding the plane". The quantitative version of this constraint would instead be "the time difference between the end of the fueling task and the start of the boarding must be between 5 and 20 minutes". Disjunctions can be expressed in both formalisms. "I will call you either before or after the flight", or "I'll be flying home either between 2pm and 4pm or between 6pm and 8pm".

Once the temporal constraints have been stated, the goal is to find an occurrence time, or duration, for all the events respecting all the constraints. In general, solving temporal constraint problems is difficult. However, there are tractable classes, such as quantitative temporal constraint problems where there is only one temporal interval for each constraint, called *simple temporal constraints* [50].

The expressive power of classical temporal constraints may be insufficient to model faithfully all the aspects of the problem. For example, while it may be true that "the plane must take off before 10am", one may want to say that "the earlier the plane takes off, the better". Moreover, one may want to state that "the flight should take no more than two hours, but the ideal time is actually one and a half hours". It is easy to see that these statements are common in the specification of most temporal problems arising in practice. Both qualitative and quantitative temporal reasoning formalisms have been extended with *quantitative preferences* to allow for the specification of such types of statements.

More precisely, Allen's approach has been augmented with fuzzy preferences [9] that allow one to express soft constraints such as "I will call you either before or after the flight, and my preference for calling you before the flight is 0.2, while that for calling you after the flight is 0.9". In other words, fuzzy preferences are associated with the relations among temporal intervals allowed by the constraints. Higher values represent the fact that such a relation is more preferred. Once such constraints have been stated, then, as usual with fuzzy preferences, the goal is to find a temporal assignment to all the variables with the highest overall preference, where the overall preference of an assignment is the lowest preference given by the temporal constraints on any constraint. Such problems are solved by exploiting specific properties of fuzzy preferences, in order to decompose the optimization problem, reducing it to solving a set of hard constraint problems.

Fuzzy preferences have been combined with both disjunctive and simple quantitative temporal constraints [115, 144]. The result is the notion of soft temporal constraints, where each allowed duration or occurrence time for a temporal event is associated with a fuzzy preference representing the desirability of that specific time. For example, it is possible to state "the time difference between the end of fueling and the start of boarding must be between 5 and 20 minutes, and the preference for 5 to 10 minutes is 1 while the preferences from 11 to 20 minutes decrease linearly from 0.8 to 0.1". The decomposition approach is the most efficient solution technique also in the quantitative setting [115, 116, 144].

The main differences between the two formalisms are the objects with which preferences are associated. In the qualitative model, preferences are associated with qualitative relations between intervals, while in the quantitative model they are associated with the explicit durations or occurrence times.

Quantitative temporal constraints have also been extended with *utilitarian* preferences: preferences take values in the set of positive reals where the goal is to maximize their sum. Such problems have been solved using branch-and-bound techniques, as well as SAT and weighted constraint satisfaction approaches [138, 145].

2.6 ABSTRACTING, EXPLAINING, AND ELICITING PREFERENCES

In constraint satisfaction problems, we look for a solution, while in soft constraint problems, we look for an optimal solution. Not surprisingly, soft constraint problems are typically more difficult to solve. To ease this difficulty, several AI techniques have been used. Here we discuss just two of them: abstraction and explanation generation. Abstraction works on a simplified version of the given problem, thus hoping to have a significantly smaller search space, while explanation generation helps understand the result of the solver: it is not always easy for a user to understand why no better solution is returned.

An added difficulty in dealing with soft constraints is related to the *modeling phase*, where a user has to understand how to faithfully model his real-life problem via soft constraints. In many cases, we may end up with a soft constraint problem where some preferences are missing, for example because they are too costly to be computed or because of privacy reasons. To reason in such scenarios, we may use techniques like machine learning and preference elicitation to solve the given problem.

2.6.1 ABSTRACTION TECHNIQUES

As noted above, soft constraints are more expressive than hard constraints, but it is also more difficult to model and solve a soft constraint problem. Therefore, sometimes it may be too costly to find all, or even a single, optimal solution. Also, although traditional propagation techniques like arc consistency can be extended to soft constraints [134], such techniques can be too costly, depending on the size and structure of the partial order associated to the problem. Finally, sometimes we may not have a solver for the class of soft constraints we need to solve, while we may have a solver for another "simpler" class of soft constraints.

For these reasons, it may be reasonable to work on a simplified version of the given soft constraint problem, while trying not to lose too much information. Such a simplified version can be defined by means of the notion of abstraction, which takes a soft CSP and returns a new one which is simpler to solve. Here, as in many other works on abstraction, "simpler" may mean many things, like the fact that a certain solution algorithm finds a solution, or an optimal solution, in fewer steps, or also that the abstracted problem can be processed by machinery which is not available for the

given problem (such as when we abstract from fuzzy to hard constraints and we only have a solver for hard constraints).

To define an abstraction, we can use the theory of *Galois insertions* [45], which provides a formal approach to model the simplification of a mathematical structure with its operators. Given a soft CSP (the *concrete* one), we may get an abstract soft CSP by simplifying the associated semiring, and relating the two structures (the concrete and the abstract one) via a Galois insertion. This way of abstracting constraint problems does not change the structure of the problem (the set of variables remains the same, as well as the set of constraints), but only the semiring values to be associated with the tuples of values for the variables in each constraint [18].

Abstraction has also been used also to simplify the solution process of hard constraint problems [126]. Also, the notion of value interchangeability has been exploited to support abstraction and reformulation of hard constraint problems [77]. In the case of hard constraints, abstracting a constraint problem means dealing with fewer variables and smaller domains.

Once we obtain some information about the abstracted version of a problem, we can bring back to the original problem some (or possibly all) of the information derived in the abstract context, and then continue the solution process on the transformed problem, which is a equivalent to the original. The hope is that, by following this route, we get to the final goal faster than just solving the original problem.

2.6.2 EXPLANATION GENERATION

One of the most important features of problem solving in an interactive setting is the capacity of the system to provide the user with justifications, or explanations, for its operations. Such justifications are especially useful when the user is interested in what happens at any time during search because he/she can alter features of the problem to ease the problem solving process.

Basically, the aim of an explanation is to show clearly why a system acted in a certain way after certain events. Explanations have been used for hard constraint problems, especially in the context of over-constrained problems [6, 112, 113], to understand why the problem does not have a solution and what can be modified in order to get one. In soft constraint problems, explanations also take preferences into account, and they provide a way to understand, for example, why there is no way to get a better solution.

In addition to providing explanations, interactive systems should be able to show the consequences, or implications, of an action to the user, which may be useful in deciding which choice to make next. In this way, they can provide a sort of "what-if" kind of reasoning, which guides the user towards good future choices. Fortunately, in soft constraint problems, this capacity can be implemented with the same machinery that is used to give explanations.

A typical example of an interactive system where constraints and preferences may be present, and where explanations can be very useful, are configurators [171]. A configurator is a system which interacts with a user to help him/her to configure a product. A product can be seen as a set of component types, where each type corresponds to a certain finite number of specific components,

and a set of compatibility constraints among subsets of the component types. A user configures a product by choosing a specific component for each component type, such that all the compatibility constraints as well as the preferences are satisfied. For example, in a car configuration problem, a user may prefer red cars but may also not want to completely rule out other colors.

Constraint-based technology is currently used in many configurators to both model and solve configuration problems [129]: component types are represented by variables, with a value for each concrete component, and both compatibility and personal constraints are represented as constraints (or soft constraints) over subsets of such variables. User choices during the interaction with the configurator are usually restricted to specifying unary constraints, in which a certain value is selected for a variable.

Whenever a choice is made, the corresponding (unary) constraint can be added to existing compatibility and personal constraints. Constraint propagation techniques like, for example, arc consistency [163] can then rule out (some of the) future choices that are not compatible with the current choice. While providing justifications based on search is difficult, arc consistency has been used as a source of guidance for justifications, and it has been exploited to help the users in some of the scenarios mentioned above. For example, it has been shown that arc consistency enforcement can be used to provide both justifications for choice elimination and also for conflict resolution [76].

This sort of approach has also been used for preference-based configurators, using the soft version of arc consistency, whose application may decrease the preferences in some constraints (while maintaining the same semantics overall). Explanations can then describe why the preferences for some variable values decrease, and they suggest at the same time which assignments can be retracted in order to get a more preferred solution [75].

Configurators with soft constraints help users not only avoid conflicts or make the next choice so that fewer later choices are eliminated, but also get to an optimal (or good enough) solution. More precisely, when the user is about to make a new choice for a component type, the configurator shows the consequences of such a choice in terms of conflicts generated, elimination of subsequent choices, and also quality of the solutions. In this way, the user can make a choice which leads to no conflict, and which presents a good compromise between choice elimination and solution quality.

2.6.3 LEARNING AND PREFERENCE ELICITATION

In a soft constraint problem, sometimes we may know a user's preferences over some of the solutions, but have no idea on how to encode this knowledge into the constraints of the problem. That is, we have a global idea about the quality of a solution, but we do not know the contribution of individual constraints to such a measure. In such a situation, it is difficult both to associate a preference with other solutions in a compatible way and to understand the importance of each tuple and/or constraint. In other situations, we may have just a rough estimate of the preferences, either for the tuples or for the constraints. Such a scenario has been addressed [161] by using machine learning techniques based on gradient descent. More precisely, it is assumed that the level of preference for some solutions (that is, the *examples*) is known, and a suitable learning technique is defined to learn, from these examples,

values to be associated with each constraint tuple, in a way that is compatible with the examples. Soft constraint learning has also been embedded in a general interactive constraint framework, where users can state both usual preferences over constraints and also preferences over solutions proposed by the system [162]. In this way, the modeling and the solution process are heavily interleaved. Moreover, the two tasks can be attacked incrementally, by specifying a few preferences at a time, and obtaining better and better solutions at each step. In this way, one requires fewer examples, since they are not given by the user all at the beginning of the solution process, but they are guided by the solver, that proposes the next best solutions and asks the user to give a feedback on them. Other approaches to learning soft constraints that provide a unifying framework for soft CSP learning have recently been developed [185]. Moreover, soft constraint learning has been exploited in the application domain of processing and interpreting satellite images [135].

Machine learning techniques have also been used to learn hard constraints (that is, to learn allowed and forbidden variable instantiations). For example, an interactive technique based on a hypothesis space containing the possible constraints to learn has been used to help the user formulate his constraints [141].

Preference learning is currently a very active area within machine learning and data mining (see [80] for a recent collection of survey and technical papers on this subject). The aim is to learn information about the preferences of an individual or a class of individuals, starting from some observations which reveal part of this information. Preference elicitation is thus a crucial ingredient in this task [34], since one would like to elicit as much information as possible without spending too many system or users' resources. An important special case of preference learning is learning how to rank: the goal here is to predict preferences in the form of total orders of a set of alternatives. The learnt preferences are often used for preference prediction, that is, to predict the preferences of a new individual or of the same individual in a new situation. This connects preference learning to several application domains, such as collaborative filtering [103, 177] and recommender systems [2, 133, 173].

2.7 OTHER PREFERENCE MODELING FRAMEWORKS

There are several other ways to model preferences, besides soft constraints and CP-nets. We will briefly describe some of them in this section.

Max SAT. SAT is the problem of deciding if a set of clauses in propositional logic can be satisfied. Each clause is a disjunction of literals, and each literal is either a variable or a negated variable. For example, a clause can be $x \vee not(y) \vee not(z)$. Satisfying a clause means giving values (either *true* or *false*) to its variables such that the clause has value *true*. MaxSAT is the problem of maximizing the number of satisfied clauses.

Since the satisfiability problem in propositional logic (SAT) is a subcase of the constraint satisfaction problem using Boolean variables and clauses, the problem MaxSAT is clearly a particular case of the weighted constraint satisfaction problem [46, 102].

Weighted and prioritized goals. An intuitive way to express preferences consists of providing a set of goals, each of which is a propositional formula, possibly adding also extra information such as priorities or weights (see [122] for a survey of goal-based approaches to preference representation). Candidates in this setting are variable assignments, which may satisfy or violate each goal. A *weighted goal* is a propositional logic formula plus a real-valued weight. The utility of a candidates is then computed by collecting the weights of satisfied and violated goals, and then aggregating them. Often only violated goals count, and their utilities are aggregated with functions such as sum or maximin. In other cases, we may sum the weights of the satisfied goals, or we may take their maximum weight. Any restriction we may impose on the goals or the weights, and any choice of an aggregation function, give a different language. Such languages may have drastically different properties in terms of their expressivity, succinctness, and computational complexity [182].

Sometimes weights are replaced by a *priority relation* among the goals. Often with prioritized goals candidates are evaluated via the so-called *discrimin* ordering: a candidate x is better than another candidate y when, for each goal g satisfied by y and violated by x, there is a goal g' satisfied by x and violated by y such that g' has priority over g.

Bayesian networks. We already mentioned Bayesian networks because of their similarities with CP-nets. Bayesian networks [143] can also be considered as specific soft constraint problems where the constraints are conditional probability tables (satisfying extra properties) using $[0, 1]$ as the semiring values, multiplication as $\times_s$ and the usual total ordering on $[0, 1]$. The Most Probable Explanation (MPE) task is then equivalent to looking for an optimal solution on such problems.

Bidding languages. Combinatorial auctions are auctions where there is a set of goods for sale. Potential buyers can make bids for subsets of this set of goods, and the auctioneer chooses which bids to accept. Bidding languages are languages used by the bidders to express their bids, that is, their preferences, to the auctioneer. In this context, usually a preference structure is given by a (monotonic) function mapping sets of goods to prices. In the OR/XOR family of bidding languages [140], bids are expressed as combinations of atomic bids of the form $\langle S, p\rangle$, where p is the price the bidder is willing to pay for the set of goods S. In the OR language, the valuation of a set is the maximal value that can be obtained by taking the sum over disjoint bids for subsets of the set. For example, the bid

$$\langle\{a\}, 2\rangle \, OR \, \langle\{b\}, 2\rangle \, OR \, \langle\{c\}, 3\rangle \, OR \, \langle\{a, b\}, 5\rangle$$

means that the bidders is willing to pay 2 for a or b alone, 3 for c alone, 5 for both a and b, and 8 for the whole set. In the XOR language, atomic bids are mutually exclusive, so the valuation of a set is the highest value offered for any of its subsets. The XOR language is more expressive than the OR language but is not very succinct since it may require the enumeration of all subsets with non-zero valuation. It is possible to combine these two languages to obtain bidding languages with better expressiveness and succinctness properties than either of the two. Many other concise bidding languages for combinatorial auctions have been defined and studied, which improve on OR/XOR languages (see, for example, the *generalized logical bids* (GLBs) in [25] and the *tree-based bidding language* (TBBL) in [32]).

Utility-based models. In the quantitative direction typical of soft constraints, there are also other frameworks to model preferences, mostly based on utilities. The most widely used assumes we have some form of independence among variables, such as *mutual preferential independence*. Preferences can then be represented by an additive utility function in deterministic decision making, or *utility independence*, which assures an additive representation for general scenarios [119]. However, this assumption often does not hold in practice since there is usually some interaction among the variables. To account for this, models based on interdependent value additivity have been defined which allows for some interaction between the variables while preserving some decomposability [72]. This notion of independence, also called *generalized additive independence* (GAI), allows for the definition of utility functions which take the form of a sum of utilities over subsets of the variables. GAI decompositions can be represented by a graphical structure, called a GAI net, which models the interaction among variables, and it is similar to the dependency graph of a CP-net or to the junction graph of a Bayesian network [8]. GAI decompositions have been used to provide CP-nets with utility functions, obtaining the so-called UCP networks [22].

Multi-criteria decision making. In multi-criteria decision making [81, 180], one assumes a set of alternatives which are decomposed into the Cartesian product of domains of some attributes (also called criteria). It is assumed that the agent has enough information to order the domain of each attribute, and the goal is to aggregate this information in order to obtain a preference ordering over the set of alternatives. Under a rather weak assumption (namely *order-separability*) [73], the ordering given on the domain of each attribute can be modeled via a utility function mapping each value in the domain to a real value representing the utility the agent has for that assignment. A very natural way to aggregate the utilities is to use a weighted sum, assuming the agent also provides some weights (usually in [0,1]) reflecting the importance of each attribute. Notice that this can be mapped directly into an equivalent weighted CSP where attributes are represented by variables, and there are only unary constraints mapping values in the domains of the variables to costs obtained by multiplying the weight of the attribute with the opposite of the corresponding utility. Despite an attractive simplicity and low complexity, this approach suffers a major drawback. In fact, using an additive aggregation operator, such as a weighted sum, is equivalent to assuming that all the attributes are independent [136]. In practice, this is not realistic, and therefore non-additive approaches have been investigated. Among them, it has been shown that fuzzy (or non-additive) measures and integrals can be used to aggregate mono-dimensional utility functions [137]. A fuzzy (or Choquet) integral, in fact, can be considered as a sort of very general averaging operator that can represent the notions of importance of an attribute and interaction between attributes.

2.8 CONCLUSIONS

Preference modeling is a very active area of research. It is important to know the properties of various preference modeling frameworks, such as their expressivity, conciseness, and computational complexity since the choice of formalism greatly influences the behaviour of the preference reasoning

environment. It would be ideal to have a single environment where users can choose how to model preferences, and where different modeling frameworks can be used even within the same problem, with the aim of allowing a more faithful representation of a real-life preference problem. Much work has been done towards this goal. However, a lot more work is still needed to fully achieve it.

Authors' Biographies

FRANCESCA ROSSI

Francesca Rossi is a full professor of Computer Science at the University of Padova, Italy. She works on constraint programming, preference reasoning, and multi-agent preference aggregation. She has published over 150 papers on these topics, and she has edited 16 volumes between collections of articles and special issues. She has been conference chair of CP 1998, program chair of CP 2003, and conference organizer of ADT 2009. She will be program chair of IJCAI 2013. She is a co-editor of the Handbook of Constraint Programming, with Peter Van Beek and Toby Walsh, published by Elsevier in 2006. She has been the president of the Association for Constraint Programming from 2003 to 2007. She is a member of the advisory board of JAIR (where she has been associate editor in 2005-2007) and of the editorial board of Constraints and of the AI Journal, an associate editor of AMAI, and a column editor for the Journal of Logic and Computation. She is an ECCAI fellow.

K. BRENT VENABLE

K. Brent Venable is currently an assistant professor in the Dept. of Pure and Applied Mathematics at the University of Padova (Italy). Her main research interests are within artificial intelligence and regard, in particular, compact preference representation formalisms, computational social choice, temporal reasoning and, more in general, constraint-based optimization. Her list of publications includes more than 50 papers, including journals and proceedings of the main international conferences on the topics relevant to her interests. She is involved in a lively international scientific exchange and, among others, she collaborates with researchers from NASA Ames, SRI International, NICTA-UNSW (Australia), University of Amsterdam (The Nederlands), 4C (Ireland) and Ben-Gurion University (Israel).

TOBY WALSH

Toby Walsh was most recently acting Scientific Director of NICTA, Australia's centre of excellence for ICT research. He is adjunct Professor at the University of New South Wales, external Professor at Uppsala University and an honorary fellow of Edinburgh University. He has been Editor-in-Chief of the Journal of Artificial Intelligence Research, and of AI Communications. He is both an AAAI and an ECCAI fellow. He has been Secretary of the Association for Constraint Programming (ACP) and is Editor of CP News, the newsletter of the ACP. Like Francesca, he is one of the Editors of the Handbook for Constraint Programming. He is also an Editor of the Handbook for Satisfiability. He has been Program Chair of CP 2001, Conference Chair of IJCAR 2004, Program and Conference Chair of SAT 2005, Conference Chair of CP 2008, and Program Chair of IJCAI 2011.