CHAPTER 8

Constraint Satisfaction Techniques

8.1 Introduction

Constraint satisfaction is a general and powerful problem-solving paradigm that is applicable to a broad set of areas, ranging from planning and scheduling to computer vision, pattern recognition, CAD, modeling, and decision support systems.

The general formulation of a constraint satisfaction problem (CSP) is as follows. Given (1) a set of variables and their respective domains, and (2) a set of constraints on the compatible values that the variables may take, the problem is to find a value for each variable within its domain such that these values meet all the constraints.

Three instances of this general problem are of special interest to planning.

- Boolean values and Boolean constraints, dealt with in the previous chapter on SAT techniques (Chapter 7).
- Variables ranging over real values or integer values with linear constraints. This is the case of the *linear programming* or *integer programming techniques.* We will refer to these mainly in Chapter 15 on planning and resource scheduling.
- Finite domains and finite tuples of constraints.

CSPs over finite domains can be used in planning in two different ways:

1. Directly, by stating a planning problem as a CSP. It is indeed possible to follow an approach similar to that of SAT, i.e., to encode a planning problem into a CSP and to rely entirely on CSP tools for planning.
2. Indirectly, by using CSP techniques within approaches specific to planning.

This latter use of CSPs in planning is much more frequent and broader than the former direct encoding. We already met several instances of particular constraint satisfaction problems in previous chapters. CSP techniques are even more

important in plan-space or Graphplan approaches within the disjunctive-refinement framework. Other instances of CSP techniques will be discussed in the rest of book, in particular with regard to planning with time and resources (Part IV). CSPs are also relevant for planning because several planning heuristics (Chapter 9) and extensions to classical planning, in particular, can rely on CSP techniques.

This chapter provides a general introduction to CSP techniques, as well as some developments on their specific use in planning. The chapter first reviews the essential definitions and concepts of CSPs over finite domains (Section 8.2). It then presents an approach for encoding a planning problem into a CSP and extracting a plan from the solution of that CSP (Section 8.3). Algorithms for solving a CSP and filtering techniques for testing its consistency are introduced (Section 8.4). The chapter briefly discusses two particular classes of CSPs of relevance to planning: active CSPs and valued CSPs (Section 8.5). CSP techniques in the plan-space approach and the Graphplan approach are discussed in Section 8.6. The chapter ends with a discussion and exercises.

8.2 Constraint Satisfaction Problems

A CSP over finite domains is defined to be a triple $\mathcal{P} = (X, \mathcal{D}, \mathcal{C})$ where:

- $X = \{x_1, \ldots, x_n\}$ is a finite set of n variables.
- $\mathcal{D} = \{D_1, \ldots, D_n\}$ is the set of finite domains of the variables, $x_i \in D_i$.
- $\mathcal{C} = \{c_1, \ldots, c_m\}$ is a finite set of constraints. A constraint c_j of some arity k restricts the possible values of a subset of k variables $\{x_{j1}, \ldots, x_{jk}\} \subseteq X$. c_j is defined as a subset of the cartesian product: $c_j \subseteq D_{j1} \times \ldots \times D_{jk}$, i.e., as the set of tuples of values allowed by this constraint for its variables: $\{(v_{j1}, \ldots, v_{jk}) \in D_{j1} \times \ldots \times D_{jk} \mid (v_{j1}, \ldots, v_{jk}) \in c_j\}$.

Definition 8.1 A *solution* to a CSP $(X, \mathcal{D}, \mathcal{C})$ is an n-tuple $\sigma = (v_1, \ldots, v_n)$ such that $v_i \in D_i$ and the values of the variables $x_i = v_i$, for $1 \leq i \leq n$, meet *all* the constraints in $\mathcal{C}$. A CSP is *consistent* if such a solution σ exists. ■

A tuple σ is a solution iff for every constraint $c_j \in \mathcal{C}$, the values specified in σ for the variables $x_{j1}, \ldots, x_{jk}$ of c_j correspond to a tuple $(v_{j1}, \ldots, v_{jk}) \in c_j$.

A set of constraints is interpreted throughout this book as a *conjunction* of the constraints, unless explicitly stated otherwise. We will say that the set of constraints $\{c_1, \ldots, c_m\}$ is consistent or, equivalently, that the conjunction $c_1 \wedge \ldots \wedge c_m$ holds, when there is a solution σ that meets every constraint in the set. Note that an empty set of constraints always holds because it is met by every tuple of values, whereas a set containing an empty constraint never holds.

A constraint can be specified *explicitly* by listing the set of its allowed tuples or the complementary set of forbidden tuples. It can also be specified *implicitly* by

using one or more relation symbols, e.g., $x_i \neq x_j$. The tuples in a constraint may be conveniently denoted by giving the variables and their values, e.g., $(x_i = v_i, x_j = v_j)$. We will often refer to the *universal constraint*, which is satisfied by every tuple of values of its variables, i.e., a constraint $c_j = D_{j1} \times \ldots \times D_{jk}$. If there is no constraint between a subset of variables, then this is equivalent to the universal constraint on these variables. We will also refer to the *empty constraint*, which forbids all tuples and cannot be satisfied.

A CSP is *binary* whenever all its constraints are binary relations. A binary CSP can be represented as a constraint network, i.e., a graph in which each node is a CSP variable x_i labeled by its domain D_i, and each edge (x_i, x_j) is labeled by the corresponding constraint on the two variables x_i and x_j. In a binary CSP, a constraint on x_i and x_j is simply denoted c_{ij}.

The symmetrical relation of a binary constraint c is the set of pairs $c' = \{(w, v) \mid (v, w) \in c\}$. A binary CSP is *symmetrical* if for every constraint $c_{ij} \in \mathcal{C}$, the symmetrical relation c'_{ji} is in $\mathcal{C}$. A symmetrical CSP $\tilde{P} = (X, D, \tilde{C})$ can be defined from $\mathcal{P} = (X, \mathcal{D}, \mathcal{C})$ by letting $\forall i, j, \tilde{c}_{ij} = c_{ij} \cap c'_{ji}$.

A unary constraint c_i on a variable x_i is simply a subset of D_i, the domain of x_i. One can replace D_i with c_i and remove this unary constraint. Hence, if a CSP involves unary and binary constraints, it is simple to express the former directly as reduced domains, keeping only binary constraints. In the remainder of this chapter, we'll focus mainly on binary symmetrical CSPs. Many popular puzzles and combinatorial problems, such as the eight-queens problem (place eight queens on a chessboard without threats between them) and the map coloring problem, are easily formalized as binary CSPs.

Let us illustrate the forthcoming definitions with a simple example.

Example 8.1 A constraint network for a symmetrical binary CSP that has five variables $\{x_1, \ldots, x_5\}$ and eight constraints $\{c_{12}, c_{13}, \ldots, c_{45}\}$ is given in Figure 8.1. All the variables have the same domain: $D_i = \{\alpha, \beta, \gamma\}, 1 \leq i \leq 5$. A solution to this CSP is, e.g., the tuple $(\alpha, \gamma, \beta, \gamma, \alpha)$, which satisfies all eight constraints:

$$(\alpha, \gamma) \in c_{12}; \quad (\alpha, \beta) \in c_{13}; \quad (\alpha, \gamma) \in c_{14}; \quad (\alpha, \alpha) \in c_{15}$$
$$(\gamma, \beta) \in c_{23}; \quad (\gamma, \gamma) \in c_{24}; \quad (\beta, \alpha) \in c_{35}; \quad (\gamma, \alpha) \in c_{45}$$

The other solutions are $(\alpha, \beta, \beta, \alpha, \beta)$, $(\alpha, \gamma, \beta, \alpha, \beta)$, and $(\beta, \gamma, \gamma, \alpha, \gamma)$. ■

It is convenient to consider a constraint network for a binary CSP as a *complete* directed graph. If it is not complete, then a missing edge from x_i to x_j corresponds implicitly to the symmetrical relation of c_{ji} if edge (x_j, x_i) exists in the network, or to the universal constraint if (x_j, x_i) does not exist. In Example 8.1, the edge (x_2, x_1) is implicitly labeled with the symmetrical relation of c_{12}, and the edge (x_2, x_5) is labeled with the universal constraint.

Definition 8.2 Two CSPs $\mathcal{P}$ and $\mathcal{P}'$ on the same set of variables X are *equivalent* if they have the same set of solutions. ■

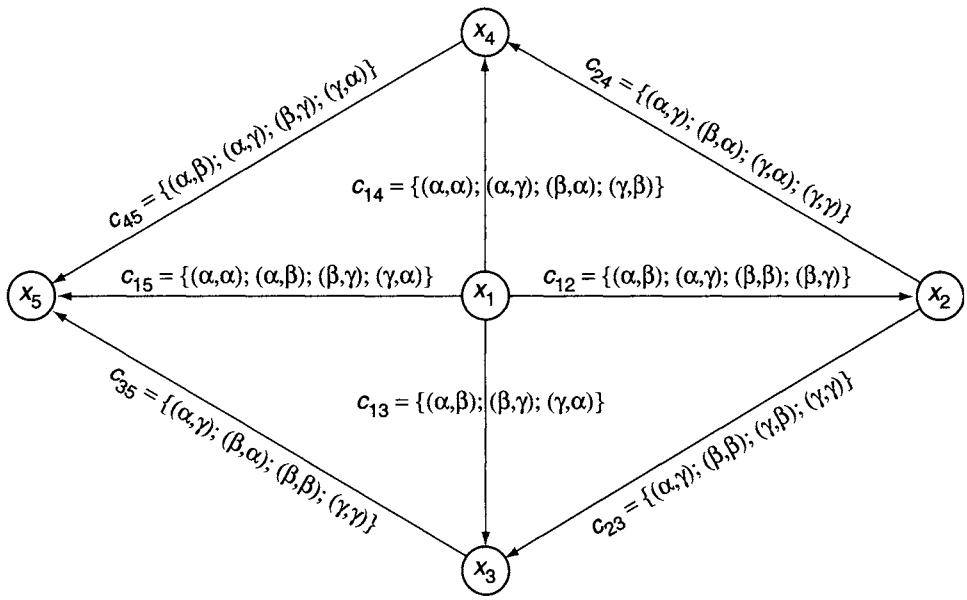

Figure 8.1 A CSP network.

For example, a binary CSP $\mathcal{P}$ and the symmetrical CSP $\tilde{P}$ obtained from $\mathcal{P}$ are equivalent (see Exercise 8.2). Given a CSP $\mathcal{P}$, one may prefer to solve an equivalent CSP $\mathcal{P}'$ if $\mathcal{P}'$ appears simpler, e.g., if $\mathcal{P}'$ has smaller domains or fewer constraints than $\mathcal{P}$. These extra values are said to be redundant in $\mathcal{P}$ and the extra constraints to be entailed from the other constraints.

Definition 8.3 A value v in a domain D_i is *redundant* if it does not appear in any solution; a tuple in a constraint c_j is redundant if it is not an element of any solution. ■

Example 8.2 Assume that the only solutions of the CSP in Figure 8.1 are the four tuples given in Example 8.1. It is easy to check that the value γ is redundant in D_1, α is redundant in D_2, the pair (β, β) is redundant in c_{12}, and (α, γ) is redundant in c_{13}. ■

The removal of a value from a domain or a tuple from a constraint corresponds to *tightening* the CSP unless that value or tuple is redundant. The removal of a redundant value from a domain or a redundant tuple from a constraint gives an equivalent CSP because it does not change the solution set. Note that if all values in a domain or if all tuples in a constraint are redundant, then the corresponding CSP is not consistent.

Definition 8.4 A CSP is *minimal* if it has no redundant values in the domains of $\mathcal{D}$ *and* no redundant tuples in the constraints of $\mathcal{C}$. ■

Definition 8.5 A set of constraints $\mathcal{C}$ is *consistent with* a constraint c iff the following holds: when $(X, \mathcal{D}, \mathcal{C})$ is consistent, then $(X, \mathcal{D}, \mathcal{C} \cup \{c\})$ is also consistent. ■

In other words, when the CSP $(X, \mathcal{D}, C)$ has a solution and the additional constraint c does not reduce its set of solutions to the empty set, then $\mathcal{C}$ is consistent with c.

Example 8.3 The constraint $c_{25} = \{(\alpha, \alpha), (\beta, \beta), (\gamma, \gamma)\}$ is consistent with the CSP in Figure 8.1: it leaves the tuples $(\alpha, \beta, \beta, \alpha, \beta)$ and $(\beta, \gamma, \gamma, \alpha, \gamma)$ as solutions to $(X, \mathcal{D}, \mathcal{C} \cup \{c_{25}\})$. ■

Definition 8.6 A set of constraints $\mathcal{C}$ *entails* a constraint c, denoted as $\mathcal{C} \models c$, iff the CSP $(X, \mathcal{D}, \mathcal{C})$ is equivalent to $(X, \mathcal{D}, \mathcal{C} \cup \{c\})$, i.e., the two CSPs have the *same* set of solutions. ■

In other words, C entails c when c does not restrict the set of solutions further than the constraints in $\mathcal{C}$. This is the case, e.g., for constraints that are transitive relations over an ordered domain: $\{x < y, y < z\}$ entails the constraint $x < z$. Note that an entailed constraint c may or may not be in $\mathcal{C}$: if c belongs to $\mathcal{C}$, then it is trivially entailed by $\mathcal{C}$. Note also that if $\mathcal{C}$ entails c, then $\mathcal{C}$ is consistent with c.

Example 8.4 The constraint $c_{25} = \{(\alpha, \alpha), (\beta, \beta), (\gamma, \gamma)\}$ given in Example 8.3 is not entailed by $\mathcal{C}$ because it reduces the set of solutions: the two tuples $(\alpha, \gamma, \beta, \gamma, \alpha)$ and $(\alpha, \gamma, \beta, \alpha, \beta)$ are not consistent with c_{25}. However, the constraint $c'_{25} = \{\beta, \beta), (\gamma, \gamma), (\gamma, \alpha), (\gamma, \beta)\}$ is entailed by $\mathcal{C}$ because the four solution tuples are consistent with this constraint. ■

Definition 8.7 A constraint $c \in \mathcal{C}$ is *redundant* iff the CSP $(X, \mathcal{D}, \mathcal{C})$ is equivalent to $(X, \mathcal{D}, \mathcal{C} - c)$, i.e., the removal of c from $\mathcal{C}$ does not change the set of solutions. ■

Example 8.5 In the CSP of Figure 8.1, the constraint c_{13} is redundant. ■

In general, removing a constraint c from $\mathcal{C}$ or adding tuples to c corresponds to *relaxing* the CSP problem unless that constraint c is redundant. Contrarily, removing tuples from a constraint corresponds to *tightening* the problem unless that tuple is redundant. Note that removing a redundant constraint from a set $\mathcal{C}$ is equivalent to replacing it with the universal constraint. This is due to the notion of removing a redundant tuple within a constraint. In a symmetrical binary CSP, for each (i, j) either c_{ij} or c_{ji} are redundant. Hence, when a binary CSP is transformed into a symmetrical CSP, one may keep only one edge and one constraint between each pair of nodes.

Given a CSP, one may be interested in addressing: (1) a *resolution* problem: finding a solution tuple; (2) a *consistency* problem: checking whether such a solution exists; (3) a *filtering* problem: removing some redundant values from domains or some redundant tuples from constraints; and (4) a *minimal reduction* problem: removing every redundant value and redundant tuple.

One may define several levels of filtering, but filtering is weaker than minimal reduction because remaining values in a filtered domain may or may not belong to a solution. Minimal reduction solves the consistency problem because a CSP has no solution iff minimal reduction reduces a domain or a constraint to an empty set. Minimal reduction greatly simplifies the resolution problem, providing potentially all solutions.

The consistency problem of a binary CSP over finite domains is NP-complete. Resolution and minimal reduction are NP-hard problems. Filtering is a polynomial problem. Its practical efficiency makes it desirable as an approximation to consistency checking: it provides a *necessary* but not a *sufficient* condition of consistency. If filtering reduces a domain or a constraint to an empty set, then the CSP is inconsistent, but the converse is not true. Filtering is useful in particular when a CSP is specified incrementally, as is typical for CSPs in planning, by adding at each step new variables and constraints. One needs to check whether a step is consistent before pursuing it further. This incremental checking can be approximated by filtering techniques.

8.3 Planning Problems as CSPs

This section presents a technique for encoding a bounded planning problem[1] P into a constraint satisfaction problem P'. This encoding has the following property: given P and a constant integer k, there is a one-to-one mapping between the set of solutions of P of length $\leq k$ and the set of solutions of P'. From a solution of the CSP problem P', if any, the mapping provides a solution plan to planning problem P. If P' has no solution, then there is no plan of length $\leq k$ for the problem P.

We will focus here on encoding classical planning problems. However we will not rely on the two popular representations we have been using so far, the set-theoretic or the classical representation, but on the state-variable representation introduced in Section 2.5. The state-variable representation is convenient for our purpose because it leads to a compact encoding into CSPs.

Recall that a state-variable representation for planning relies on the following elements.

- *Constant symbols* are partitioned into disjoint classes corresponding to the objects of the domain, e.g., the classes of robots, locations, cranes, containers, and piles in the DWR domain.

1. As defined in Section 7.2.3, a bounded planning problem is restricted to the problem of finding a plan of length at most k for an a priori given integer k.

- *Object variable symbols* are typed variables; each ranges over a class or the union of classes of constants, e.g., $r \in$ robots, $l \in$ locations, etc.
- *State variable symbols* are functions from the set of states and one or more sets of constants into a set of constants, such as:

  rloc: robots$\times S \rightarrow$ locations
  rload: robots$\times S \rightarrow$ containers $\cup$ {nil}
  cpos: containers$\times S \rightarrow$ locations $\cup$ robots
- *Relation symbols* are *rigid* relations on the constants that do not vary from state to state for a given planning domain, e.g., adjacent(loc1,loc2), belong(pile1,loc1), attached(crane1,loc1). The type of an object variable can also be specified with a rigid relation, e.g., locations(l), robots(r).

A planning operator is a triple: o = (name(o),precond(o),effects(o)) where precond(o) is a set of expressions that are conditions on state variables and on rigid relations, and effects(o) is a set of assignments of values to state variables. The statement of a bounded planning problem is $P = (O, R, s_0, g, k)$, where O, s_0, and g are as usual, R is the set of rigid relations of the domain, and k is the length bound.

The encoding of a planning problem into a CSP will be illustrated on a simple example that we presented earlier (Example 2.8).

Example 8.6 This simplified DWR domain has no piles and no cranes, and robots can load and unload autonomously containers. The domain has only three operators:

move(r, l, m)
  ;; robot r at location l moves to an adjacent location m
  precond: rloc(r) = l, adjacent(l, m)
  effects: rloc(r) $\leftarrow$ m

load(c, r, l)
  ;; robot r loads container c at location l
  precond: rloc(r) = l, cpos(c) = l, rload(r) = nil
  effects: rload(r) $\leftarrow$ c, cpos(c) $\leftarrow$ r

unload(c, r, l)
  ;; robot r unloads container c at location l
  precond: rloc(r) = l, rload(r) = c
  effects: rload(r) $\leftarrow$ nil, cpos(c) $\leftarrow$ l

The domain has three state variables, i.e., rloc(r), rload(r), and cpos(c), and a rigid relation adjacent(l, m). To keep the notation simple, the state argument in state variables is implicit. The expressions in precond(a) for an action a refer to the values of state variables in a state s, whereas the updates in effects(a) refer to the values in the state $\gamma(s, a)$. ■

8.3.1 Encoding a Planning Problem into a CSP

A bounded planning problem $P = (O, R, s_0, g, k)$ in the state-variable representation is encoded into a constraint satisfaction problem P' in four steps corresponding respectively to (1) the definition of the CSP variables of P', (2) the definition of the constraints of P' encoding the initial state s_0 and the goal g, (3) the encoding of the actions that are instances of operators in O, and (4) the encoding of the frame axioms.

This encoding of $P = (O, R, s_0, g, k)$ is with respect to the given integer k: the set of solutions of the CSP P' is intended to correspond to the set of plans $\langle a_1, \ldots, a_k \rangle$ of P of length $\leq k$. We are interested in characterizing the sequences of states $\langle s_0, s_1, \ldots, s_k \rangle$ corresponding to such plans. For convenience, let us refer to the state s_j in this sequence by its index j, for $0 \leq j \leq k$.

Step 1: CSP Variables. The CSP variables of P' are defined as follows.

- For each ground state variable x_i of P ranging over D_i and for each $0 \leq j \leq k$, there is a CSP variable of P', $x_i(j, v_u, \ldots, v_w)$ whose domain is D_i.
- For each $0 \leq j \leq k - 1$, there is a CSP variable of P', denoted act(j), whose domain is the set of all possible actions in the domain, in addition to a no-op action that has no preconditions and no effects, i.e., $\forall s, \gamma(s, \text{noop}) = s$. More formally:

  act: $\{0, \ldots, k-1\} \rightarrow D_{\text{act}}$
  $D_{\text{act}} = \{a(v_u, \ldots, v_w)$ ground instance of $o \in O\} \cup \{\text{noop}\}$

Hence, the CSP variables are all the ground state variables of P, plus one variable act($\hat{j}$) whose value corresponds to the action carried out in state j.

Example 8.7 Let $P = (O, R, s_0, g)$, where O are the operators given in Example 8.6 for a simplified DWR domain with one robot (r1), three containers (c1, c2, c3), and three locations (l1, l2, l3). Let s_0 be the state s_0 = {rloc(r1) = l1, rload(r1) = nil, cpos(c1) = l1, cpos(c2) = l2, cpos(c3) = l2}; and let g = {cpos(c1) = l2, cpos(c2) = l1}. Assume that we are seeking a plan of at most $k = 4$ steps. The corresponding CSP P' has the following set of variables:

- rloc(j,r1) $\in$ {l1, l2, l3}, for $0 \leq j \leq 4$
- rload(j,r1) $\in$ {c1, c2, c3, nil}, for $0 \leq j \leq 4$
- cpos(j, c) $\in$ {l1, l2, l3, r1}, for $c \in$ c1, c2, c3 and $0 \leq j \leq 4$
- act(j) $\in$ {move(r, l, m), load(c, r, l), unload(c, r, l), no-op}, for all the possible instances of these operators and for $0 \leq j \leq 3$

Note that act(k) is not a variable in P'. In this problem there are $6 \times 5 - 1 = 29$ CSP variables. ■

Now that we have the CSP variables in P', the constraints of P' will enable us to encode the initial state s_0, the goal g, the actions, and the frame axioms.

Step 2: Constraints Encoding s_0 and g. The encoding of the state s_0 and the goal g into constraints follows directly from the definition of the CSP variables.

Every ground state variable x_i whose value in s_0 is v_i is encoded into a unary constraint of the corresponding CSP variable for $j = 0$ of the form:

$$(x_i(0) = v_i) \tag{8.1}$$

Every ground state variable x_i whose value is v_i in the goal g is encoded into a unary constraint of the corresponding CSP variable for $j = k$:

$$(x_i(k) = v_i) \tag{8.2}$$

Note that there is no constraint for s_0 and g on the CSP variables act(j).

Example 8.8 The state s_0 of Example 8.7 is translated into the following constraints:
rloc(0,r1) = l1, rload(0,r1) = nil, cpos(0,c1) = l1, cpos(0,c2) = l2,
cpos(0,c3) = l2.
The goal g is translated into the two constraints:
cpos(4,c1) = l2, and cpos(4,c2) = l1. ■

Step 3: Constraints Encoding Actions. This encoding step translates the actions of the planning problem P into binary constraints of P'. Let $a(v_u, \ldots, v_w)$ be an action instance of some operator $o \in O$ such that the constants $v_u, \ldots, v_w$ meet the rigid relations in the preconditions of a. Then, $\forall j, 0 \leq j \leq k-1$:

- Every condition of the form $(x_i = v_i)$ in precond(a) is translated into a constraint with a single tuple of the form:

$$(\text{act}(j) = a(v_u, \ldots, v_w), x_i(j) = v_i) \tag{8.3}$$

- Every condition of the form $(x_i \in D'_i)$ in precond(a) is translated into a constraint corresponding to the set of pairs:

$$\{(\text{act}(j) = a(v_u, \ldots, v_w), x_i(j) = v_i) \mid v_i \in D'_i\} \tag{8.4}$$

- Every assignment $x_i \leftarrow v_i$ in effects(a) is translated into a constraint with a single tuple:

$$(\text{act}(j) = a(v_u, \ldots, v_w), x_i(j+1) = v_i) \tag{8.5}$$

We can write these sets of constraints with arguments that are object variables instead of constants. However, we restrict the instances of these object variables to tuples that meet the rigid relation in the preconditions. This is illustrated in the following example with the adjacent relation.

Example 8.9 Let us give the constraints translating the move and load operators in Example 8.7. The move operator has only one condition and one effect; it is encoded into the following constraints:

$$\{(\mathsf{act}(j) = \mathsf{move}(r, l, m), \mathsf{rloc}(j, r) = l) \mid \mathsf{adjacent}(l, m) \wedge 0 \leq j \leq 3\}$$
$$\{(\mathsf{act}(j) = \mathsf{move}(r, l, m), \mathsf{rloc}(j + 1, r) = m) \mid \mathsf{adjacent}(l, m) \wedge 0 \leq j \leq 3\}$$

The operator load has three conditions and two effects, which are encoded into the following constraints:

$$\{(\mathsf{act}(j) = \mathsf{load}(c, r, l), \mathsf{rloc}(j, r) = l) \mid 0 \leq j \leq 3\}$$
$$\{(\mathsf{act}(j) = \mathsf{load}(c, r, l), \mathsf{rload}(j, r) = \mathsf{nil}) \mid 0 \leq j \leq 3\}$$
$$\{(\mathsf{act}(j) = \mathsf{load}(c, r, l), \mathsf{cpos}(j, c) = l) \mid 0 \leq j \leq 3\}$$
$$\{(\mathsf{act}(j) = \mathsf{load}(c, r, l), \mathsf{rload}(j + 1, r) = c) \mid 0 \leq j \leq 3\}$$
$$\{(\mathsf{act}(j) = \mathsf{load}(c, r, l), \mathsf{cpos}(j + 1, c) = r) \mid 0 \leq j \leq 3\}$$

■

Step 4: Constraints Encoding Frame Axioms. This final step encodes the frame axioms constraints. A frame axiom constraint says that any state variable that is invariant for an action a (i.e., not explicitly modified in an assignment in effects(a)) remains unchanged between s and $\gamma(s, a)$. A frame axiom is encoded into a *ternary* constraint involving three variables: act(j) and the invariant state variable but of in state j and in state $j + 1$. More precisely: for every action $a(v_u, \ldots, v_w)$ and every state variable x_i that is invariant for a, we have a constraint with the following set of triples:

$$\{(\mathsf{act}(j) = a(v_u, \ldots, v_w),\ x_i(j) = v_i,\ x_i(j + 1) = v_i) \mid v_i \in D_i\} \qquad (8.6)$$

Note that no-op has no action constraint, since it has no precondition and no effect, but every state variable is invariant for no-op.

Example 8.10 In Example 8.7, two state variables are a-invariant with respect to action move: rload and cpos. Consequently, the frame axioms for this operator are the following constraints, for $0 \leq j \leq 3$:

$$\{(\mathsf{act}(j) = \mathsf{move}(r, l, m), \mathsf{rload}(j, r) = v, \mathsf{rload}(j + 1, r) = v) \mid \mathsf{adjacent}(l, m) \wedge v \in D_{\mathsf{rload}}\}$$
$$\{(\mathsf{act}(j) = \mathsf{move}(r, l, m), \mathsf{cpos}(j, c) = v, \mathsf{cpos}(j + 1, c) = v) \mid \mathsf{adjacent}(l, m) \wedge v \in D_{\mathsf{cpos}}\}$$

where $D_{\mathsf{rload}} = \{\mathsf{c1}, \mathsf{c2}, \mathsf{c3}, \mathsf{nil}\}$ and $D_{\mathsf{cpos}} = \{\mathsf{l1}, \mathsf{l2}, \mathsf{l3}, \mathsf{r1}\}$.

■

Plan Extraction. We have encoded a planning problem P and an integer k into a CSP P'. Let us assume that we have a tool for solving CSPs. Given P' as input, this CSP solver returns a tuple σ as a solution of P' or failure if P' has no solution. The tuple σ gives a value to every CSP variable in P', in particular to the variables act(j). Let these values in σ be: $\mathsf{act}(j) = a_{j+1}$, for $0 \leq j \leq k - 1$. Each a_j is an action of P,

and the sequence $\pi = \langle a_1, \ldots, a_k \rangle$ is a valid plan of P that possibly includes no-op actions.

Proposition 8.1 *There is a one-to-one mapping between the set of plans of length $\leq k$ that are solutions of a bounded planning problem P and the set of solutions of the CSP problem P' encoding P.*

Proof Let σ be a tuple solution of P'. The value in σ of the variable $\mathsf{act}(0) = a_1$ meets all the constraints, in particular those specified in Formulas 8.3 and 8.4, for every condition in precond(a_1): $x_i(0) = v_i$. These values of $x_i(0)$ also meet the unary constraints in Formula 8.1 for state s_0. Consequently, action a_1, whose preconditions are met in state s_0, is applicable to s_0.

Consider now the state s_1 corresponding to the state variables defined by the values of $x_i(1)$ in the solution σ. These values of $x_i(1)$ meet all the constraints, in particular those of Formulas 8.5 and 8.6 for $j = 0$. This is exactly the definition of the state resulting from applying a_1 to s_0, hence $s_1 = \gamma(s_0, a_1)$.

The same argument can be repeated for $\mathsf{act}(1) = a_2$ and $s_2 = \gamma(s_1, a_2)$, ..., till $\mathsf{act}(k-1) = a_k$ and $s_k = \gamma(s_{k-1}, a_k)$. Now, the values $x_i(k) = v_i$ meet also the unary constraints of Formula 8.2, i.e., the goal g is satisfied in s_k. Hence $\pi = \langle a_1, \ldots, a_k \rangle$ is a valid plan of P.

Conversely, let $\pi = \langle a_1, \ldots, a_k \rangle$ be a solution plan of P and let $s_1 = \gamma(s_0, a_1)$, ..., $s_k = \gamma(s_{k-1}, a_k)$ be the corresponding sequence of states. Consider the tuple σ that gives to every CSP variable $x_i(j)$ the value corresponding to that of the state variable x_i in state s_j, for $0 \leq j \leq k$, and $\mathsf{act}(j) = a_{j+1}$, for $0 \leq j \leq k-1$. It is straightforward to show that σ meets all the constraints of P', hence it is a solution of P'.

This proof also shows that there is no plan of length $\leq k$ for the planning problem P iff the CSP P' is inconsistent. ■

8.3.2 Analysis of the CSP Encoding

It is interesting to compare the encoding of a planning problem P into a CSP to the encoding of P into a SAT problem (see Section 7.2.3). The two encodings are very similar except for the specific properties of the state-variable representation used here, which relies on functions instead of relations. Indeed, the encoding steps used here are similar to those used in SAT for encoding states, goals, actions, and the frame axioms. However, the encoding step in SAT called the *complete exclusion axioms* (Formula 7.13), which requires that only one action is performed per state, is not needed here, thanks to the state-variable representation. The act variables correspond to functions from states to possible actions. Hence, they take one and only one value for each state. Similarly, the encoding of a state is simpler than in SAT encoding, which required a specific formula (Formula 7.3) to restrict valid models to intended models. Other more elaborate SAT encodings were discussed in Section 7.4.2; they can also be considered for the CSP encoding in the state-variable representation.

The encoding of a planning problem into a CSP requires $m = k(n + 1) - 1$ CSP variables, where n is the number of state variables and k is the bound on the plan length. However, the domain of each CSP variable can be quite large. It is important to remark that both CSP and SAT are NP-complete problems, whereas planning, and more precisely the PLAN-LENGTH problem which is of interest to us here, is PSPACE- or NEXPTIME-complete (see Section 3.3 and Table 3.2). This is explained by the fact that the two encodings of planning, into SAT or into a CSP, lead to a blowup in the size of the resulting problem. For SAT, this blowup results in an exponential number of Boolean variables. For the CSP encoding, the number of variables remains linear in the size of P, but the total size of the CSP P' is exponential, i.e., it is $d = \prod_{i=1}^{i=m} |D_i|$, where D_i is the domain of the CSP variable x_i.

For the same value of d, it is often easier to solve a CSP that has a small number of CSP variables (small m) ranging over large domains than a CSP that has a large number of binary variables. However, this depends largely on the constraints involved.

A related issue here is that the frame axioms are encoded into ternary constraints. CSP solvers have less efficient techniques for handling ternary constraints than binary constraints. Ternary constraints can be translated into binary ones [35, 149], but at the expense of an increase in the size of the problem.

An interesting property of the CSP encoding is that the problem P' remains close enough to the original planning problem P to allow embedding in the CSP solver domain-specific control knowledge for planning, if such knowledge is available, in addition to general CSP heuristics.

8.4 CSP Techniques and Algorithms

This section is devoted to a brief presentation of the main algorithms used for handling a CSP, either for solving it (Section 8.4.1) or for filtering its domains and constraints (Section 8.4.2). These algorithms are the backbones of general CSP solvers over finite domains. They can also be useful for designing a constraint manager for a planner.

For simplicity, this section focuses on algorithms for binary CSPs. The management of higher arity constraints will be briefly discussed in Section 8.7.

8.4.1 Search Algorithms for CSPs

A backtracking search algorithm that solves a CSP is given in Figure 8.2. It is a nondeterministic recursive procedure. Its arguments are σ, the tuple of values corresponding to the current partial solution, and X, the set of remaining variables for which a value has to be found. Starting from an empty σ and the full set X of n variables, the procedure heuristically selects the next variable x_i. Then it propagates previous choices of the values in σ on the domain of x_i. This is done by reducing

```
Backtrack(σ, X)
    if X = ∅ then return(σ)
    select any x_i ∈ X
    for each value v_j in σ do
        D_i ← D_i ∩ {v ∈ D_i | (v, v_j) ∈ c_ij}
    if D_i = ∅ then return(failure)
    nondeterministically choose v_i ∈ D_i
    Backtrack(σ.(v_i), X − {x_i})
end
```

Figure 8.2 Backtracking search algorithm for a CSP.

D_i to the sole values consistent with those already in σ. In this reduced domain, a value v_i for x_i is nondeterministically chosen. This value is added to σ in a recursive call on remaining variables.

In a deterministic implementation, the nondeterministic step would be associated to a backtracking point over other untried values in the reduced D_i, if any, or a backtracking point further up in the recursion tree if needed, as for an empty D_i. Note that the step select is not a backtracking point: all variables have to be processed, and the order in which they are selected affects the efficiency of the algorithm, not its completeness (see Appendix A).

This algorithm is sound and complete. It runs in time $O(n^d)$ for $d = max_i\{|D_i|\}$. Its practical performance depends essentially on the quality of the heuristics used for ordering the variables and for choosing their values.

- *Heuristics for variable ordering* rely on the idea that a backtrack deep in the search tree costs more than an early backtrack. Hence the principle is to choose first the most constrained variables in order to backtrack, if needed, as early as possible in the search tree. These most constrained variables are heuristically evaluated by the cardinality of their domains. The heuristic chooses the variable x_i with the smallest $|D_i|$.
- *Heuristics for the choice of values* apply the opposite principle of preferring the least constraining value v_i for a variable x_i. These least constraining values v_i are heuristically evaluated by the number of pairs in constraints c_{ij} in which v_i appears. The heuristics chooses the most frequent v_i.

Both types of heuristics can be computed *statically*, before the algorithm starts, or *dynamically* at each recursion node, taking into account current domains and constraints as reduced by earlier choices. Heuristics computed dynamically are more informed than static ones.

Another improvement to the Backtrack algorithm is to propagate a potential choice v_i for x_i on the domains of remaining variables before committing to that

choice in the following recursions. Propagation consists of removing values inconsistent with v_i. It is preferable to find out at this point whether such a propagation leads to an empty domain D_j than to discover it much later in the recursion tree, when x_j is considered. The algorithm implementing this technique is called Forward-Checking. It is given in Figure 8.3. Forward-Checking has been shown empirically and analytically to be a faster algorithm than backtracking [334].

Other improvements to backtrack search are based on *intelligent backtracking* or *dependency-directed backtracking* and on *learning techniques.* These techniques can be seen as *look-back improvements* of the search, while Forward-Checking provides a *look-ahead improvement.* The principle of intelligent backtracking is to identify at a failure position the variables that can be usefully backtracked to. If a failure occurs at x_i with an empty D_i, then there is no need to backtrack to the next previous point x_j if the value v_j in σ did not contribute toward reducing D_i. The *conflict set* at a failure position corresponds to the tuple included in σ that made D_i empty. The algorithm called Backjumping [148] backtracks to the next previous variable within the conflict set. Learning relies on recording and analyzing these conflict sets as nogood tuples never to be committed to again [148, 298].

8.4.2 Filtering Techniques

Despite good heuristics and many improvements, the resolution of a CSP remains in general a costly combinatorial problem. However, it is possible to test the consistency of a CSP with fast algorithms that provide a necessary but not a sufficient condition of consistency. These algorithms address the *filtering problem* introduced earlier, i.e., they remove redundant values from domains or redundant tuples from constraints. When such a removal leads to an empty domain or to an empty constraint, then the CSP is inconsistent.

Filtering techniques rely on a *constraint propagation operation.* Propagating a constraint on a variable x consists of computing its local effects on variables adjacent

```
Forward-Checking(σ, X)
    if X = ∅ then return(σ)
    select any x_i ∈ X
    nondeterministically choose v_i ∈ D_i
    for each x_j ∈ X, j ≠ i do
        D_j ← D_j ∩ {v ∈ D_j | (v_i, v) ∈ c_ij}
        if D_j = ∅ then return(failure)
    Forward(σ.(v_i), X − {x_i})
end
```

Figure 8.3 Forward-checking search algorithm for a CSP.

to x in the constraint network, removing redundant values and tuples. This removal in turn leads to new constraints that need to be propagated. Hence, filtering is a *local propagation operation* that is pursued until no further change occurs, i.e., until a fixed-point termination condition.

Filtering through constraint propagation is particularly adequate when a CSP is defined incrementally by adding to it new variables and constraints, as in many planning systems. Incremental filtering algorithms allow a fast although partial testing of the consistency of the current state of a CSP.

Arc Consistency. A straightforward filter, called *arc consistency*, consists of removing from a domain D_i any value that does not satisfy constraints c_{ij} involving x_i. Such a value is redundant because it necessarily violates a constraint.

A naive algorithm for arc consistency is to perform an iteration over all pairs of variables, (x_i, x_j), $i \neq j$, with the two following updates:

$$D_i \leftarrow \{v \in D_i \mid \exists v' \in D_j : (v, v') \in c_{ij}\};$$
$$D_j \leftarrow \{v' \in D_j \mid \exists v \in D_i : (v, v') \in c_{ij}\}.$$

Once a pair of variables (x_i, x_j) is filtered out, the next pair involving one of these two variables, e.g., (x_i, x_k), may further reduce D_i. This may entail other redundant values in D_j and so on. One may keep repeating the naive iteration until a fixed point is reached. Such a fixed point is a complete iteration during which no domain is reduced. At this point, one is sure that the only values left in a domain D_i meet all the constraints involving variable x_i. If a domain D_i is reduced to an empty set, then the CSP is inconsistent. If no D_i is reduced to an empty set, then the CSP is said to be *arc-consistent* or *2-consistent*. Arc-consistent CSPs are not necessarily consistent.

Example 8.11 In the CSP of Example 8.1, filtering out the variables (x_1, x_2) reduces the two domains to $D_1 = \{\alpha, \beta\}$ and $D_2 = \{\beta, \gamma\}$ because no pair in c_{12} starts with a γ or ends with an α. ■

A better arc-consistency algorithm, called AC3 (Figure 8.4), keeps a list L of pairs of variables whose domains have to be filtered. Initially L is the set of all constrained pairs in the CSP, i.e., $L = \{(x_i, x_j) \mid c_{ij} \text{ or } c_{ji} \text{ are in } \mathcal{C}\}$. L is updated whenever filtering reduces a domain, which may entail further filtering. Note that when (x_i, x_j) is in L, then (x_j, x_i) is also in L. Hence the two domains D_i and D_j are both filtered out.

AC3 runs in time $o(md^2)$, where $m = |\mathcal{C}|$ and $d = max_i\{|D_i|\}$. It is an *incremental* algorithm: it can filter out a CSP to which variables and constraints are added incrementally. In that case, L is initially the set of pairs concerned with newly added variables and constraints. Other arc-consistency algorithms, improving on the idea of AC3, have been developed, e.g., algorithm AC4 [399].

Path Consistency. A more thorough filter, called *path consistency*, consists of testing all triples of variables x_i, x_j, and x_k, checking that they have values that meet the

```
AC3(L)
    while L ≠ ∅ do
        select any pair (x_i, x_j) in L and remove it from L
        D ← {v ∈ D_i | ∃v' ∈ D_j : (v, v') ∈ c_ij}
        if D ≠ D_i then do
            D_i ← D
            L ← L ∪ {(x_i, x_k), (x_k, x_i) | ∃c_ik or c_ki ∈ C, k ≠ j}
end
```

Figure 8.4 An arc-consistency algorithm.

three constraints c_{ij}, c_{jk}, and c_{ik}. A pair of values (v_i, v_j) can be part of a solution if it meets the constraint c_{ij} and if there is a value v_k for x_k such that (v_i, v_k) meets c_{ik} and (v_k, v_j) meets c_{kj}. In other words, the two constraints c_{ik} and c_{kj} entail by transitivity a constraint on c_{ij}.

Let us define a composition operation between constraints, denoted $\bullet$:

$$c_{ik} \bullet c_{kj} = \{(v, v'), v \in D_i, v' \in D_j \mid \exists w \in D_k: (v, w) \in c_{ik} \text{ and } (w, v') \in c_{kj}\}.$$

The composition $c_{ik} \bullet c_{kj}$ defines a constraint from x_i to x_j entailed by the two constraints c_{ik} and c_{kj}. For example, in Figure 8.1, $c_{24} \bullet c_{45} = \{(\alpha,\alpha), (\beta,\beta), (\beta,\gamma), (\gamma,\beta), (\gamma,\gamma), (\gamma,\alpha)\}$.

A pair (v_i, v_j) has to meet c_{ij} as well as the composition $c_{ik} \bullet c_{kj}$ for every k; otherwise, it is redundant. Hence the filtering operation here is:

$$c_{ij} \leftarrow c_{ij} \cap [c_{ik} \bullet c_{kj}], \text{ for every } k \neq i, j$$

When the filterings lead to an empty constraint, then the CSP is inconsistent. Note that while arc consistency removes redundant values from domains, path consistency removes redundant tuples from constraints.

Example 8.12 In the CSP of Figure 8.1, $c_{14} \bullet c_{45} = \{(\alpha,\alpha), (\alpha,\beta), (\alpha,\gamma), (\beta,\gamma)\}$; hence $c_{15} \cap [c_{14} \bullet c_{45}]$ filters out the tuple (γ,α). ■

A simple algorithm called PC (Figure 8.5) filters out constraints for all triples (x_i, x_j, x_k) on the basis of this operation until a fixed point with no further possible reduction is reached. This algorithm assumes a complete symmetrical graph, or a graph implicitly completed for missing edges with the symmetrical constraint or with the universal constraint. Note that the symmetrical of $[c_{ik} \bullet c_{kj}]$ is $[c_{jk} \bullet c_{ki}]$.

```
PC(C)
    until stabilization of all constraints in C do
        for each k : 1 ≤ k ≤ n do
            for each pair i, j : 1 ≤ i < j ≤ n, i ≠ k, j ≠ k do
                c_ij ← c_ij ∩ [c_ik • c_kj]
                if c_ij = ∅ then exit(inconsistent)
end
```

Figure 8.5 Path consistency in a CSP

A CSP is said to be *path-consistent* or *3-consistent* if a path-consistency algorithm does not detect an inconsistency. However, algorithm PC, as any other filtering algorithm, is not complete for consistency checking: it may not detect an inconsistent network, as illustrated in the following example.

Example 8.13 Consider the graph coloring problem, i.e., assigning distinct colors to adjacent vertices in a graph, for a complete graph with four vertices and only three colors. This is a CSP with four variables that range over a set of three colors; the six constraints require different colors at their vertices. This CSP is obviously inconsistent although it is path-consistent. ■

Furthermore, a network that is initially arc-consistent may not be arc-consistent after being filtered out by PC because several redundant tuples have been removed from constraints. It is easy to extend this algorithm for maintaining both path and arc consistency through a uniform representation obtained by defining $c_{ii} = D_i^2$ and modifying the filtering operation as follows:

$$c_{ij} \leftarrow c_{ij} \cap [c_{ik} \bullet c_{kk} \bullet c_{kj}] \text{ for all triples, including for } i = j$$

An incremental version of algorithm PC (Figure 8.6) works like AC3 by updating the list of constraints to be filtered out. This *Updatelist* is initialized to the entire set $\mathcal{C}$ or to the subset of newly added or modified constraints. The filtering operation is performed by procedure Revise, which returns either the constraint to be added to *Updatelist* or $\emptyset$ if no updating is needed. An improved path-consistency algorithm, called PC3 [399], runs in $O(n^3 d^3)$.

Other filters can be defined by checking the consistency of all k-tuples. A CSP is said to be *k-consistent* if for any subset of k variables $\{x_1, \ldots, x_k\}$ and for any tuple of $(k-1)$ consistent values $\sigma = (v_1, \ldots, v_{k-1})$ there is a value for variable x_k that is consistent with σ. A CSP is *strongly k-consistent* if for all i: $1 \leq i \leq k$ the CSP is i-consistent. Note that for $k = n$, strong n-consistency is equivalent to consistency [150]. In practice, filtering techniques are useful mainly for $k = 2$

```
IPC(Updatelist)
    while Updatelist ≠ ∅ do
    select any c_ij in Updatelist and remove it from Updatelist
    for each k : 1 ≤ k ≤ n and k ≠ i, j do
        Updatelist ← Updatelist ∪ Revise(i, k, [c_ij • c_jk])
        Updatelist ← Updatelist ∪ Revise(j, k, [c_ji • c_ik])
end
```

```
Revise(i, j, c)
    c ← c ∩ c_ij
    if c = ∅ then exit(inconsistent)
    if c = c_ij then return(∅)
    c_ij ← c
    return(c_ij)
end
```

Figure 8.6 Incremental path-consistency algorithm.

(arc consistency) and $k = 3$ (path consistency). They are seldom used for higher levels of k-consistency, giving the computational cost that grows exponentially with k and the incompleteness of filtering.

8.4.3 Local Search Techniques and Hybrid Approaches

Local search methods present an alternative to filtering techniques for the same purpose of avoiding the combinatorial explosion of a complete search. Local search methods are not complete: they can fail to find a solution when there is one. But they may find a solution very efficiently.

The main ingredient of a local search method is a neighborhood $\mathcal{N}(\sigma)$ of a candidate solution tuple $\sigma = (v_1, \ldots, v_n)$. $\mathcal{N}(\sigma)$ is a set of other tuples that can be obtained easily from σ, e.g., by changing the value of one variable x_i.

A local search algorithm starts from some initial σ_0. If this tuple does not meet the constraints, then it moves to another candidate tuple $\sigma \in \mathcal{N}(\sigma_0)$ that appears closer to a solution according to some distance estimate, i.e., σ meets more constraints than the initial tuple. Local search follows basically a local gradient according to this distance estimate. It may rely on several stochastic strategies, such as simulated annealing and tabu search as illustrated in Section 7.3.2 for SAT.

Hybrid techniques perform a combination of filtering, local search, and systematic search. For example, a systematic search may rely on filtering or local search at some node of the search tree for pruning and/or choosing the next step.

Alternatively, a local search may use a bounded systematic search for assessing a local neighborhood. For example, Jussien and Lhomme [292] propose a technique called decision-repair: at some node of the search a value is chosen for a variable. Filtering is applied from this partial tuple, and if an inconsistency is detected, a local search is performed to improve the current tuple. Good results on several CSP applications have been reported for these hybrid techniques [292, 470].

8.5 Extended CSP Models

The standard CSP model discussed so far can be extended in several ways. Two such extensions, *active CSPs* and *valued CSPs*, of interest to planning are briefly introduced here. A third extension, called *mixed CSPs* or *stochastic CSPs*, which involves normal CSP variables linked with constraints to contingent random variables whose values cannot be chosen by the CSP solver, will be briefly discussed in Section 8.7 and in the chapters on temporal constraints and temporal planning (see Part IV).

8.5.1 Active CSPs

The *active CSP* model extends the constraint satisfaction model with a notion of *activity* for variables.[2] The set X is partitioned into *active* and *inactive* variables. Initially, a subset of variables is active. Other inactive variables may become active depending on the values chosen for the already active variables. This is stated by a set of activity constraints of the form:

$$(x_i = v_i) \wedge \ldots \wedge (x_j = v_j) \Rightarrow x_k, \ldots, x_l \text{ are active}$$

One has to solve the CSP of finding consistent values only for active variables—those initially active and those that become active because of the triggered activity constraints.

Example 8.14 Let us modify three constraints in the network of Figure 8.1 as follows: remove the pair (β, α) from c_{35}, remove (α, β) from c_{45}, and remove (β, γ) from c_{15}. This leads to the network in Figure 8.7, which is an inconsistent CSP.

Assume now that this is an active CSP with the following activity constraints:

$(x_1 = \alpha) \Rightarrow x_2$ and x_3 active
$(x_2 = \beta) \Rightarrow x_4$ and x_5 active
$(x_3 = \beta) \Rightarrow x_4$ active

2. This type of CSP is often referred to in the literature as *dynamic CSPs*. In the context of planning, it is preferable to reserve the latter expression for the case where the CSP variables evolve dynamically as functions of time and where the problem is to find a set of such functions that meet all the constraints.

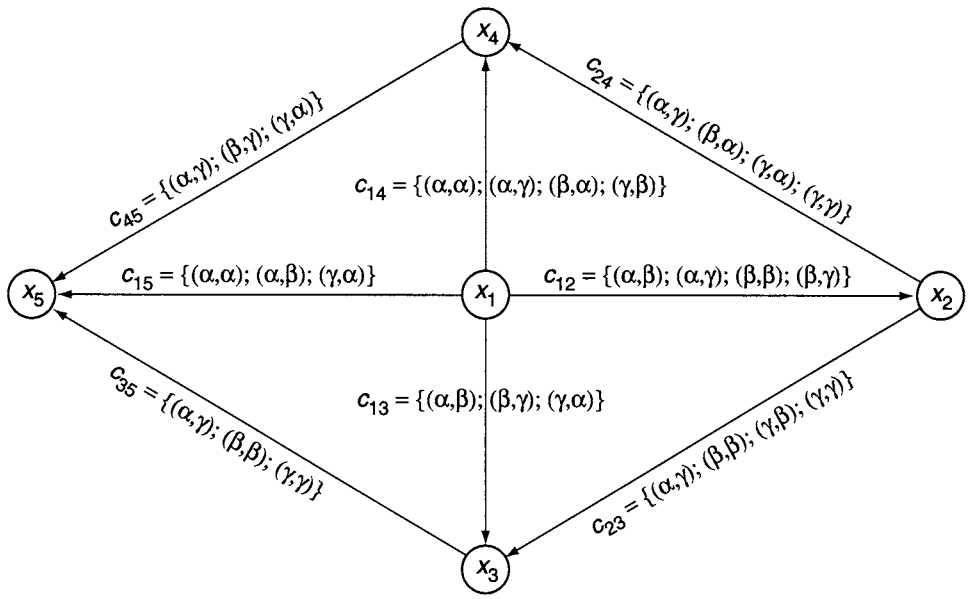

Figure 8.7 An active CSP network.

If only x_1 is initially active, then this active CSP has a single solution: $(x_1 = \alpha, x_2 = \beta, x_3 = \beta, x_4 = \alpha)$. ■

Search algorithms such as Backtrack and Forward-Checking can be extended to handle active CSPs. For example, in Backtrack one restricts the set X to the variables initially active. X is updated according to activity constraints whenever a value is chosen for a variable. Forward-Checking requires an additional restriction of the domains of newly active variables to values compatible with the current σ; it also requires a careful management of backtracking steps.

Instead, one may transform an active CSP into a normal CSP and apply to it all the standard resolution algorithms or filtering techniques and known heuristics. This transformation proceeds as follows.

- Extend the domain of every initially inactive variable with a particular value called inactive.
- Rewrite the activity constraints as normal constraints of the form: $(x_i = v_i) \wedge \ldots \wedge (x_j = v_j) \Rightarrow (x_k \neq \text{inactive}) \wedge \ldots \wedge (x_l \neq \text{inactive})$

This transformation of an active CSP into a normal one allows the direct use of all the algorithms, heuristics, and software tools developed for CSPs.

8.5.2 Valued CSPs

The *valued CSP* model extends the model of constraint satisfaction with utilities and an optimization criteria. A *utility value* is attached to each constraint. An n-tuple $\sigma = (v_1, \ldots, v_n)$ that meets a subset of the constraints in C has a utility $U(\sigma)$ equal to the sum of the utilities of the constraints satisfied by σ. The problem is to find a solution that maximizes the criteria U. A dual formulation is to associate costs to constraints and to seek a tuple that minimizes the sum of the costs of violated constraints.

Exact resolution techniques for valued CSPs are based on the Branch-and-Bound algorithm. Basically, one extends the search algorithm with heuristics and bounding estimates in order to prune the search tree.

The *local search techniques* offer approximation approaches that are more tractable for large problems with hundred of variables and constraints. One searches locally in the neighborhood of an n-tuple σ through local modifications in order to improve the criteria. Random techniques such as simulated annealing, tabu search, and genetic algorithms are examples of these approximation techniques.

8.6 CSP Techniques in Planning

As explained in Section 8.1, the use of CSP techniques within approaches specific to planning is much more frequent and broader than the direct encoding of a planning problem into a CSP. We already encountered constraint satisfaction issues in plan-space and planning-graph planning (Chapters 5 and 6, respectively). In the forthcoming chapters, particularly for planning with time and resources (Part IV), we will rely extensively on CSPs. Let us revisit here briefly the plan-space and planning-graph approaches in order to review possible improvements in the light of CSP techniques.

8.6.1 CSPs in Plan-Space Search

In plan-space planning, a partial plan involves precedence constraints between actions and binding constraints between variables. The precedence constraints define a very specific CSP that is efficiently handled with the usual graph-search techniques. The binding constraints, however, correspond to a general CSP.[3] It can be very time-consuming to check the consistency of these constraints at each step of the planner, when new binding constraints are added or modified. One may rely on filtering techniques to perform partial tests at each incremental refinement of a partial plan. A complete consistency check can be performed at regular stages, at least when a plan has been found. Here, backtracking will be more difficult to manage.

3. For example, a graph coloring problem is directly coded with binding constraints of the form $x \neq y$.

The relationship between plan-space planning and CSPs is more important if one generalizes the *flaw-repair refinement* framework of a plan-space algorithm. Recall that this framework involves three steps:

1. Analyze flaws, i.e., goals and threats.
2. Find resolvers for each flaw.
3. Choose one resolver for a flaw and refine the current plan with that resolver.

In plan-space procedures such as PSP (Figure 5.6), this last step is a nondeterministic branching point where backtracking occurs whenever needed. In a more general framework, called *disjunctive refinement*, one does not branch nondeterministically on the choice of a particular resolver for a flaw. Instead one keeps as much as possible the entire disjunction of resolvers. This leads to a refinement *without* branching. The current node is refined with the disjunction of resolvers. Hence, a node is not a partial plan anymore but a disjunction of partial plans.

The main idea, introduced in the Descartes planner, is to associate with each flaw a CSP variable whose domain is the set of resolvers for that variable. One has to manage variables associated with threats differently than those associated with open goals. This is because a refinement for a threat adds only constraints to a node; it does not add new resolvers to other pending flaws. But a refinement for an open goal may add new resolvers to other flaws. We will explore more details on these techniques in the context of temporal planning (Section 14.3.5).

Finally, as a transition to the next section, let us mention some recent improvements of plan-space planning that involve CSP techniques as well as Graphplan-based distance heuristics [424].

8.6.2 CSPs for Planning-Graph Techniques

Graphplan already handles the mutex relations between propositions and actions in every level as binary constraints. In our discussion of this algorithm and its improvements (Section 6.4.2), we referred to several CSP techniques and heuristics for Graphplan, such as look-ahead enhancements, e.g., through forward-checking, or look-back improvements, e.g., through intelligent backtracking and learning.

But one may go even further in the use of CSP techniques and tools by encoding the planning graph for a given planning problem into a CSP, as proposed in the GP-CSP planner. Along that view, the graph expansion phase corresponds to the encoding of the problem into a particular CSP, while the graph extraction phase is the resolution of that CSP. In this encoding, the following conditions are met.

- Each proposition $p \in P_i$ is a CSP variable.
- The values in the variable p are the actions in A_i supporting p.
- The constraints between CSP variables are given by the mutex relations between actions. If a and b are two mutex actions, then for every p and q

supported respectively by a and b, we have a constraint that forbids the pair $(p = a, q = b)$.

This encoding leads to an active CSP. The goal propositions are the variables initially active. Activity constraints say that when a proposition p is associated to a value a, then all preconditions of a in the previous level become active, unless a is in the first level A_1 (because we do not need to support the propositions in s_0). Finally, mutex relations between propositions are encoded into constraints on activities: if p and q are mutex in some level, then we have the constraint $\neg(active(p) \wedge active(q))$.

A further encoding step makes it possible to transform this active CSP into a normal CSP, as explained in Section 8.5.1, and to run a standard CSP solver for its resolution. The iterative deepening procedure of Graphplan (which is equivalent to considering bounded planning problems over increasing bounds) and the termination condition can be directly handled in this encoding framework.

There can be several benefits in compiling a planning graph into a CSP and solving it with standard CSP tools. Among the benefits are the availability of several CSP filtering and local search techniques that can speed up the search and provide heuristics, pruning, and control information; the capability to perform a nondirectional search (unlike Graphplan, whose search follows the graph levels in decreasing order) as directed by available control information; and the potential to integrate planning to other problems that are naturally solved with CSP tools, such as scheduling and design.

Finally, there are also several techniques in constraints programming that enable one to provide additional control knowledge as constraints and to guide the search in an interactive user-oriented framework.

8.7 Discussion and Historical Remarks

CSPs were developed very early in the history of computer science within particular application areas such as CAD and pattern recognition. There is the well-known contribution of Waltz [542] for line labeling in computer vision, which started constraint propagation techniques, and the contribution of Stallman and Sussman [497] in CAD circuit analysis, which introduced intelligent backtracking techniques.

The pioneering work of Montanari [400] established CSPs as a generic problem-solving paradigm. There is now a very wide range of literature in this area. Several surveys [151, 339] offer a good introduction to the field. Russel and Norvig devote a chapter to CSPs in the new edition of their textbook [458]. The monographs of Tsang [515] and Mariott and Stuckey [379] detail extensively most of the CSP techniques we have discussed. This last reference develops CSPs as a paradigm for programming, following on a long tradition of work on logic programming and logic and constraints programming. Constraint programming is successfully used today for addressing resource allocation and scheduling problems [45, 178].

Search techniques for CSPs have been studied and improved through many contributions [148, 201, 373]. Montanari [400] started filtering techniques. Arc consistency received considerable attention, e.g., the AC3 algorithm is due to Mackworth [373], AC4 to Mohr and Henderson [399], and AC6 to Bessiere and Cordier [67]. More general k-consistency techniques are due to Freuder [202]. Heuristics for CSP have been proposed and analyzed in two sources [153, 395]. The performance of several search techniques has been studied empirically [34] and theoretically [334].

Local search methods for CSPs started with Kirkpatrick *et al.* [324], within an optimization framework, followed by several other contributions, in particular to hybrid approaches [292, 470].

The extended CSP models, such as active and valued CSPs, have attracted many contributions (e.g., [466, 529]). The work on the so-called mixed CSPs or stochastic CSPs, which involve normal CSP variables linked with constraints to contingent random variables whose values cannot be chosen by the CSP solver, is more recent (e.g., [185, 541]). It certainly is of relevance for planning, in particular for the management of temporal constraints (see Part IV).

Constraints in planning appeared quite early. For example, the Molgen planner [498] provides a good illustration of the management of binding constraints. Several HTN planners, particularly UMCP [174], make extensive use of CSP techniques to manage state, variable binding, and ordering constraints.

Advanced CSP techniques in plan-space planning were introduced in the Descartes planner [289, 290] for handling threats and open goals. The use of CSPs for managing threats or causal constraints has been analyzed [296]. CSPs for Graphplan have been discussed [305]; look-back enhancements such as data-directed backtracking and learning have been introduced [298, 299, 300]. The idea has been further developed for both types of flaws in the Descartes planner.

The encoding of a planning problem into a CSP has been explored in particular [158, 519, 539, 556]. The paper by van Beck and Chen [519] proposes a constraint programming approach in which the planning problem is directly specified by the user as a CSP problem. Two other approaches [539, 556] use encodings into Integer Linear Programming (ILP), a special class of CSP with integer variables and linear constraints (see Section 24.3). The approach of Do and Kambhampati [158], summarized in Section 8.6.2, encodes the planning graph into a CSP. It led to a planner called GP-CSP that relies on the CSPLIB software library [518]. Do and Kambhampati report a more compact and efficient encoding than the SAT-based or ILP-based encodings and a significant speedup with respect to standard Graphplan.

8.8 Exercises

8.1 In the CSP of Figure 8.1, find all the redundant values in D_i, $1 \leq i \leq 5$, and all the redundant tuples in the constraints.

8.2 Prove that a binary CSP $\mathcal{P}$ and the symmetrical CSP $\tilde{P}$ obtained from $\mathcal{P}$ are equivalent.

8.3 Define the five operators of the DWR domain in the state-variable representation (see the three operators of the simplified DWR of Example 8.6; see page 173).

8.4 Define a planning problem P in the state-variable representation for the DWR domain of the previous exercise, with three locations, five containers initially in loc1, and one robot in loc2, with the goal of having all containers in loc3. Encode P into a CSP.

8.5 Run the Forward-Checking algorithm on the CSP of Example 8.1 (see page 169). Modify the pseudocode of this algorithm (given in Figure 8.3) so as to find all solutions and run it on this CSP. Compare with the four solutions given in Example 8.1 (see page 169).

8.6 Run the AC3 algorithm on the CSP of Example 8.1 (see page 169). What are the filtered domains of the variables?

8.7 Modify the pseudocode of algorithm PC (Figure 8.5) so as to filter with both arc and path consistency, i.e., to test for strong 3-consistency. Run this algorithm on the two networks in Figures 8.1 and 8.7.

8.8 Dowload the GP-CSP planner.[4] Run it on the same DWR domains and problems used for Exercise 6.16. Discuss the practical range of applicability of this planner for the DWR application.

8.9 Compare the performances of GP-CSP on the DWR domain with those of the Graphplan planner from Exercise 6.16.

8.10 Compare the performances of GP-CSP on the DWR domain with those of the HSPOP planner from Exercise 5.11.

4. See http://rakaposhi.eas.asu.edu/gp-csp.html.