Chapter 7

Propositional Satisfiability Techniques

7.1 Introduction

The general idea underlying the "planning as satisfiability" approach is to map a planning problem to a well-known problem for which there exist effective algorithms and procedures. A plan is then extracted from the solution to the new problem. More specifically, the idea is to formulate a planning problem as a *propositional satisfiability problem*, i.e., as the problem of determining whether a propositional formula is satisfiable. The approach follows this outline.

- A planning problem is *encoded* as a propositional formula.
- A *satisfiability decision procedure* determines whether the formula is satisfiable by assigning truth values to the propositional variables.
- A plan is *extracted* from the assignments determined by the satisfiability decision procedure.

Recent improvements in the performance of general purpose algorithms for propositional satisfiability provide the ability to scale up to relatively large problems. The ability to exploit the efforts and the results in a very active field of research in computer science is indeed one of the main advantages of this approach.

In this chapter, we focus on the encoding of planning problems into satisfiability problems and describe some existing satisfiability procedures that have been used in planning. We first discuss a way to translate a planning problem to a propositional formula (Section 7.2). Then we show how standard decision procedures can be used as planning algorithms (Section 7.3). We then discuss some different ways to encode a planning problem into a satisfiability problem (Section 7.4).

In this chapter we focus on classical planning problems, with all the restrictions described in Chapter 1. The "planning as satisfiability" approach has been recently extended to different kinds of planning problems, most notably to the problem of

planning in nondeterministic domains. These extensions are described in Part V, Chapter 18.

7.2 Planning Problems as Satisfiability Problems

We consider a classical planning problem $\mathcal{P}$ as defined in Sections 2.1 and 2.2, where $\mathcal{P} = (\Sigma, s_0, S_g)$; $\Sigma = (S, A, \gamma)$ is the planning domain, S the set of states, A the set of actions, γ the deterministic transition function, s_0 the initial state, and S_g the set of goal states.

In the "planning as satisfiability" approach, a classical planning problem is encoded as a propositional formula with the property that any of its models correspond to plans that are solutions to the planning problem. A *model* of a propositional formula is an assignment of truth values to its variables for which the formula evaluates to true.[1] A *satisfiability problem* is the problem of determining whether a formula has a model. We say that a formula is *satisfiable* if a model of the formula exists.

In the following subsections we describe how a planning problem can be translated to the problem of determining whether a propositional formula is satisfiable. The two key elements that need to be translated to propositional formulas are states and state transitions. We first provide some guidelines on how states and state transitions can be translated to propositional formulas (Sections 7.2.1 and 7.2.2, respectively). We then describe the encoding of a planning problem to a satisfiability problem (Section 7.2.3). The size of the encoding, i.e., the number of propositional variables and the length of the formula, are critical. Sizes of different encodings are discussed in Section 7.4.

7.2.1 States as Propositional Formulas

Similar to the set-theoretic representation (Section 2.2), we use propositional formulas to represent facts that hold in states. For instance, consider a simple domain where we have a robot r1 and a location l1. The states where the robot is unloaded and at location l1 can be represented with the following formula:

$$\mathsf{at}(\mathsf{r1}, \mathsf{l1}) \wedge \neg \mathsf{loaded}(\mathsf{r1}) \tag{7.1}$$

If we consider at(r1, l1) and loaded(r1) as propositional variables, Formula 7.1 is a propositional formula. A model of Formula 7.1 is the one that assigns true to the

1. See Appendix B for a review of basic concepts in propositional logic, e.g., propositional formulas, propositional variables, and models.

propositional variable at(r1, l1), and false to loaded(r1). Let us call this model μ. More formally, μ can be written as:

$$\mu = \{\mathsf{at}(\mathsf{r1}, \mathsf{l1}) \leftarrow \mathit{true}, \mathsf{loaded}(\mathsf{r1}) \leftarrow \mathit{false}\} \tag{7.2}$$

Consider now the case where we have two locations l1 and l2. As a consequence we have, beyond the propositional variables at(r1, l1) and loaded(r1), the propositional variable at(r1, l2), which intuitively represents the fact that robot r1 is at location l2. Suppose we want to represent the fact that the robot is unloaded and at location l1. Formula 7.1 does not "exactly" entail this. Indeed, we have two models:

$$\mu_1 = \{\mathsf{at}(\mathsf{r1}, \mathsf{l1}) \leftarrow \mathit{true}, \mathsf{loaded}(\mathsf{r1}) \leftarrow \mathit{false}, \mathsf{at}(\mathsf{r1}, \mathsf{l2}) \leftarrow \mathit{true}\}$$
$$\mu_2 = \{\mathsf{at}(\mathsf{r1}, \mathsf{l1}) \leftarrow \mathit{true}, \mathsf{loaded}(\mathsf{r1}) \leftarrow \mathit{false}, \mathsf{at}(\mathsf{r1}, \mathsf{l2}) \leftarrow \mathit{false}\}$$

μ_2 is the *intended model*, i.e., intuitively, the model we have in mind when we write Formula 7.1 because we think that a robot cannot stay in two locations at the same time. μ_1 is an *unintended model*, i.e., an assignment to the formula that we do not have in mind but makes the formula true. Formally, nothing distinguishes μ_1 from μ_2. They are both models. In order to avoid the unintended model, we have to add to Formula 7.1 a proposition that represents an obvious fact for us, i.e., the fact that the robot cannot be at location l2:

$$\mathsf{at}(\mathsf{r1}, \mathsf{l1}) \wedge \neg\mathsf{at}(\mathsf{r1}, \mathsf{l2}) \wedge \neg\mathsf{loaded}(\mathsf{r1}) \tag{7.3}$$

The only model of Formula 7.3 is μ_2, i.e., the intended model.

A propositional formula can represent sets of states rather than a single state. For instance,

$$(\mathsf{at}(\mathsf{r1}, \mathsf{l1}) \wedge \neg\mathsf{at}(\mathsf{r1}, \mathsf{l2})) \vee (\neg\mathsf{at}(\mathsf{r1}, \mathsf{l1}) \wedge \mathsf{at}(\mathsf{r1}, \mathsf{l2})) \wedge \neg\mathsf{loaded}(\mathsf{r1}) \tag{7.4}$$

represents both the state where the robot is at l1 and the one where it is at l2.

Encoding states as propositional formulas is therefore rather straightforward. However, notice that a propositional formula represents a state (or a set of states) without encoding the dynamics of the system. We need some different kind of propositional formulas to represent the fact that a system evolves from a state to another state.

7.2.2 State Transitions as Propositional Formulas

The behavior of deterministic actions is described by the transition function $\gamma : S \times A \rightarrow S$. Figure 7.1 depicts the state transition of an action move(r1,l1,l2)

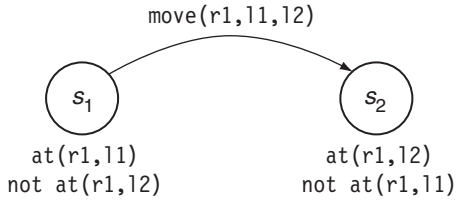

Figure 7.1 An example of state transition.

such that $\gamma(s_1, \mathsf{move}(\mathsf{r1}, \mathsf{l1}, \mathsf{l2})) = s_2$. In s_1, the robot r1 is at location l1, while in s_2 it is at l2. States s_1 and s_2 could be represented by Formulas 7.5 and 7.6, respectively:

$$\mathsf{at}(\mathsf{r1}, \mathsf{l1}) \wedge \neg\mathsf{at}(\mathsf{r1}, \mathsf{l2}) \tag{7.5}$$

$$\neg\mathsf{at}(\mathsf{r1}, \mathsf{l1}) \wedge \mathsf{at}(\mathsf{r1}, \mathsf{l2}) \tag{7.6}$$

However, these formulas cannot be used to represent the fact that the system evolves from state s_1 to state s_2. We need a propositional formula to assert that, in state s_1, before executing the action, Formula 7.5 holds, and in state s_2, after the execution of the action, Formula 7.6 holds. We need different propositional variables that hold in different states to specify that a fact holds in one state but does not hold in another state.

For instance, we can have two distinct propositional variables $\mathsf{at}(\mathsf{r1}, \mathsf{l1}, \mathsf{s1})$ and $\mathsf{at}(\mathsf{r1}, \mathsf{l2}, \mathsf{s2})$. The first one intuitively means that the robot is at location l1 in state s_1, while the second means that the robot is at location l2 in state s_2. They are different and can be assigned different truth values. The transition in Figure 7.1, i.e., the fact that the system evolves from state s_1 to state s_2, can be represented by the following propositional formula:

$$\mathsf{at}(\mathsf{r1}, \mathsf{l1}, \mathsf{s1}) \wedge \neg\mathsf{at}(\mathsf{r1}, \mathsf{l2}, \mathsf{s1}) \wedge \neg\mathsf{at}(\mathsf{r1}, \mathsf{l1}, \mathsf{s2}) \wedge \mathsf{at}(\mathsf{r1}, \mathsf{l2}, \mathsf{s2}) \tag{7.7}$$

where at(r1,l1,s1), at(r1,l2,s1), at(r1,l1,s2), and at(r1,l2,s2) are four different propositional variables.[2] A model of Formula 7.7 is

$$\begin{aligned} \mu_3 = \{\, & \mathsf{at}(\mathsf{r1}, \mathsf{l1}, \mathsf{s1}) \leftarrow \mathit{true}, \\ & \mathsf{at}(\mathsf{r1}, \mathsf{l2}, \mathsf{s1}) \leftarrow \mathit{false}, \\ & \mathsf{at}(\mathsf{r1}, \mathsf{l1}, \mathsf{s2}) \leftarrow \mathit{false}, \\ & \mathsf{at}(\mathsf{r1}, \mathsf{l2}, \mathsf{s2}) \leftarrow \mathit{true}\,\} \end{aligned}$$

Formula 7.7 encodes the transition from state s_1 to state s_2. We can now represent the fact that it is the action move(r1,l1,l2) that causes this transition. One possible

2. Notice that Formula 7.7 encodes the transition from state s_2 to s_1 as well.

way to do this is to encode move(r1,l1,l2) as a propositional variable. We always have to take into account the fact that we need to distinguish when the action is executed in state s_1 from when it is executed in another state. We therefore introduce a propositional variable, move(r1,l1,l2,s1), whose intended meaning is that the action is executed in state s_1. The function $\gamma(s_1, \mathsf{move}(\mathsf{r1}, \mathsf{l1}, \mathsf{l2}))$ can therefore be encoded as

$$\mathsf{move}(\mathsf{r1}, \mathsf{l1}, \mathsf{l2}, \mathsf{s1}) \wedge \mathsf{at}(\mathsf{r1}, \mathsf{l1}, \mathsf{s1}) \wedge \neg\mathsf{at}(\mathsf{r1}, \mathsf{l2}, \mathsf{s1}) \wedge \neg\mathsf{at}(\mathsf{r1}, \mathsf{l1}, \mathsf{s2}) \wedge \mathsf{at}(\mathsf{r1}, \mathsf{l2}, \mathsf{s2}) \quad (7.8)$$

We have one model of Formula 7.8:

$$\begin{aligned}\mu_4 = \{\ & \mathsf{move}(\mathsf{r1}, \mathsf{l1}, \mathsf{l2}, \mathsf{s1}) \leftarrow \mathit{true},\\ & \mathsf{at}(\mathsf{r1}, \mathsf{l1}, \mathsf{s1}) \leftarrow \mathit{true},\\ & \mathsf{at}(\mathsf{r1}, \mathsf{l2}, \mathsf{s1}) \leftarrow \mathit{false},\\ & \mathsf{at}(\mathsf{r1}, \mathsf{l1}, \mathsf{s2}) \leftarrow \mathit{false},\\ & \mathsf{at}(\mathsf{r1}, \mathsf{l2}, \mathsf{s2}) \leftarrow \mathit{true}\ \}\end{aligned}$$

This encoding is similar to the situation calculus representation (see Chapter 12). The main difference is that situation calculus allows for variables denoting states, called *situations*, in a first-order logical language, while propositional satisfiability encodes the planning problem with propositions.

Notice also that this encoding is conceptually different from the set-theoretic representation (Section 2.2), where state transitions are represented as operators, i.e., as functions that map states to states that are represented as sets of propositions. Here the encoding represents the state transition as a formula in propositional logic.

7.2.3 Planning Problems as Propositional Formulas

Given the fact that we can encode states and state transitions as propositional formulas (Sections 7.2.1 and 7.2.2), we can then encode a planning problem to a propositional formula, say, Φ. The construction of Φ is based on two main ideas.

- Restrict the planning problem to the problem of finding a plan of known length n for some fixed n. This problem is called the *bounded planning problem.* We call each i, $0 \leq i \leq n$, a *step* of the planning problem.
- Transform the bounded planning problem into a satisfiability problem. The description of states and actions of the bounded planning problem are mapped to propositions that describe states and actions at each step, from step 0 (corresponding to the initial state) to step n (corresponding to the goal state).

Technically, each predicate symbol with k arguments is translated into a symbol of $k+1$ arguments, where the last argument is the step. In the case of predicate symbols describing the state such as at(r1,l1) we have at(r1,l1,i), $0 \leq i \leq n$. The intended

meaning is that the robot r1 is at location l1 at step i. We call *fluents* the ground atomic formulas that describe states at a given step, e.g., at(r1,l1,i).[3] In the case of actions, such as move(r1,l1,l2), we have move(r1,l1,l2,i), $0 \leq i \leq n-1$. The intended meaning is that the robot r1 moves from location l1 at step i and gets to l2 at step $i+1$. Intuitively, the proposition encoding an action at step i represents an action that is executed at step i and its effects hold at step $i+1$.

A bounded planning problem can be easily extended to the problem of finding a plan of length $\leq n$, for some fixed n, with the use of a dummy action that does nothing. In principle, if a solution plan exists, the plan has a maximum length less than or equal to the number of all states, i.e., $|S|$, which is a double exponential in the number of constant symbols and predicate arity: $n \leq 2^{|D|^{A_p}}$, where $|D|$ is the number of constants in the domain and A_p is the maximum arity of predicates. However, this is of little practical relevance because such a bound is too large, and looking for plans of such length leads to failure. The hope is that there might exist a plan with a relatively short length so that it will not be necessary to go through this large space. In practice, therefore, the plan length is not known in advance. Since the "planning as satisfiability" approach can deal only with bounded planning problems, the execution of planning algorithms based on this approach needs to be repeated for different tentative lengths, e.g., with a binary search on plan lengths. For instance, the algorithm can be iteratively run (e.g., each time with plan length fixed at 2, 4, 8, etc.) until a plan is found.

The formula Φ encoding a bounded planning problem can be constructed as the conjunction of formulas describing the initial and goal states (where atoms are instantiated at steps 0 and n, respectively) and formulas describing the behavior of actions (e.g., preconditions and effects) through the n steps.

Here we describe one of the possible ways to construct a formula Φ that encodes a bounded planning problem into a satisfiability problem. (Section 7.4 discusses different possibilities.) We write as f_i a fluent at step i. For instance, if f is at(r1,l1), we write at(r1,l1,i) as f_i. We write as a_i the propositional formula representing the action executed at step i. For instance, if a is move(r1,l1,l2), we write move(r1,l1,l2,i) as a_i.

Φ is built with the following five kinds of sets of formulas.

1. The *initial state* is encoded as a proposition that is the conjunction of fluents that hold in the initial state and of the negation of those that do not hold, all of them instantiated at step 0.

$$\bigwedge_{f \in s_0} f_0 \wedge \bigwedge_{f \notin s_0} \neg f_0 \tag{7.9}$$

   The initial state is thus fully specified.

3. Fluents here are propositional variables; this notion is therefore different from the notion of fluent introduced in Chapter 2. There is also a difference with the fluents introduced in Chapter 12, where fluents are terms in first-order logic.

2. The *set of goal states* is encoded as a proposition that is the conjunction of fluents that must hold at step n.

$$\bigwedge_{f \in g^+} f_n \wedge \bigwedge_{f \in g^-} f_n \tag{7.10}$$

The goal state is partially specified by the conjunction of the fluents that hold in all the goal states.

3. The fact that *an action, when applicable, has some effects* is encoded with a formula that states that if the action takes place at a given step, then its preconditions must hold at that step and its effects will hold at the next step. Let A be the set of all possible actions. For each $a \in A$ and for each $0 \leq i \leq n-1$, we have:

$$a_i \Rightarrow \left(\bigwedge_{p \in \mathsf{precond}(a)} p_i \wedge \bigwedge_{e \in \mathsf{effects}(a)} e_{i+1} \right) \tag{7.11}$$

4. We need to state that *an action changes only the fluents that are in its effects.* For instance, moving a container does not change the position of other containers. The need for formalizing and reasoning about this fact is known as the *frame problem* (see also Chapter 2). This fact can be reformulated equivalently in a slightly different way as the fact that, *if a fluent changes, then one of the actions that have that fluent in its effects has been executed.* More precisely, if a fluent f, which does not hold at step i, holds instead at step $i+1$, then one of the actions that has f in its positive effects has been executed at step i. Similarly, in the case that f holds at step i and not at step $i+1$, an action that has f in its negative effects has been executed. We have therefore a set of propositions that enumerate the set of actions that could have occurred in order to account for a state change. These formulas are called *explanatory frame axioms.* For each fluent f and for each $0 \leq i \leq n-1$, we have:

$$\begin{aligned} \neg f_i \wedge f_{i+1} &\Rightarrow \left(\bigvee_{a \in A | f_i \in \mathsf{effects}^+(a)} a_i \right) \wedge \\ f_i \wedge \neg f_{i+1} &\Rightarrow \left(\bigvee_{a \in A | f_i \in \mathsf{effects}^-(a)} a_i \right) \end{aligned} \tag{7.12}$$

5. The fact that only one action occurs at each step is guaranteed by the following formula, which is called the *complete exclusion axiom.* For each $0 \leq i \leq n-1$, and for each distinct $a_i, b_i \in A$, we have:

$$\neg a_i \vee \neg b_i \tag{7.13}$$

The propositional formula Φ encoding the bounded planning problem into a satisfiability problem is the conjunction of Formulas 7.9 through 7.13.

Example 7.1 Consider a simple example where we have one robot r1 and two locations l1 and l2. Let us suppose that the robot can move between the two locations. In the initial state, the robot is at l1; in the goal state, it is at l2. The operator that moves the robot is:

move(r, l, l')
  precond: at(r, l)
  effects: at(r, l'), $\neg$ at(r, l)

In this planning problem, a plan of length 1 is enough to reach the goal state. We therefore fix the length of the plan to $n = 1$. The initial and goal states are encoded as formulas (init) and (goal), respectively:

(init) at(r1,l1,0) $\wedge\neg$ at(r1,l2,0)
(goal) at(r1,l2,1) $\wedge\neg$ at(r1,l1,1)

The action is encoded as:

(move1) move(r1, l1, l2, 0) $\Rightarrow$ at(r1, l1, 0) $\wedge$ at(r1, l2, 1) $\wedge$ $\neg$at(r1, l1, 1)
(move2) move(r1, l2, l1, 0) $\Rightarrow$ at(r1, l2, 0) $\wedge$ at(r1, l1, 1) $\wedge$ $\neg$at(r1, l2, 1)

The explanatory frame axioms are:

(at1) $\neg$at(r1, l1, 0) $\wedge$ at(r1, l1, 1) $\Rightarrow$ move(r1, l2, l1, 0)
(at2) $\neg$at(r1, l2, 0) $\wedge$ at(r1, l2, 1) $\Rightarrow$ move(r1, l1, l2, 0)
(at3) at(r1, l1, 0) $\wedge$ $\neg$at(r1, l1, 1) $\Rightarrow$ move(r1, l1, l2, 0)
(at4) at(r1, l2, 0) $\wedge$ $\neg$at(r1, l2, 1) $\Rightarrow$ move(r1, l2, l1, 0)

The complete exclusion axioms are:

$\neg$move(r1, l1, l2, 0) $\vee$ $\neg$move(r1, l2, l1, 0)

■

We can easily extract a plan from the model of Φ. Given a bounded planning problem of length n, we take a sequence of propositional variables $\langle a_1(0), \ldots, a_n(n-1)\rangle$, where $a_i(j)$ is the propositional variable representing action a_i at step j, such that the model assigns true to all $a_{i+1}(i)$, $0 \leq i \leq n-1$. The plan that can be extracted is $\langle a_1, \ldots, a_n\rangle$.

Example 7.2 Consider Example 7.1. A model of the formula Φ assigns true to move(r1,l1,l2,0). The plan of length 1 that is extracted is move(r1,l1,l2). Suppose we now have a different planning problem where the robot can move from/to the two locations and load and unload containers. In the initial state, we have the robot r1 at location l1, the robot is unloaded, and a container c1 is in location l1. The goal is to move container c1 to l2. We fix the bound to three steps. The encoded planning problem has a model that assigns true to:

move(r1,l1,l2,1), load(l1,c1,r1,0), unload(l1,c1,r1,2)

The plan that is extracted is the sequence:

⟨ load(l1,c1,r1), move(r1,l1,l2), unload(l1,c1,r1) ⟩

■

We can now formalize the notion of encoding correctly a planning problem into a satisfiability problem. Let $\Sigma = (S, A, \gamma)$ be a deterministic state-transition system (as described in Section 1.4). Let $\mathcal{P} = (\Sigma, s_0, S_g)$ be a classical planning problem, where s_0 and S_g are the initial and goal states, respectively. Let Enc be a function that takes a planning problem $\mathcal{P}$ and a length bound n and returns a propositional formula Φ: $\mathsf{Enc}(\mathcal{P}, n) = \Phi$.

Definition 7.1 Enc *encodes the planning problem $\mathcal{P}$ to a satisfiability problem* when the following holds: Φ is satisfiable iff there exists a solution plan of length n to $\mathcal{P}$. We say, in short, that Enc encodes planning to satisfiability. ■

Proposition 7.1 *The encoding defined by Formulas 7.9 through 7.13 encodes planning to satisfiability.*

See Kautz *et al.* [312] for a proof of this proposition.

7.3 Planning by Satisfiability

Once a bounded planning problem is encoded to a satisfiability problem, a model for the resulting formula can be constructed by a satisfiability decision procedure. A variety of different procedures have been devised in the literature. In this section we describe some of the procedures that have been used extensively for planning. We describe mainly two different kinds of procedures.

1. The algorithms based on the *Davis-Putnam procedure* are sound and complete. A satisfiability procedure is *sound* if every input formula on which it returns a model is satisfiable; it is *complete* if it returns a model on every satisfiable input formula. When a bounded planning problem of length n is encoded as a satisfiability problem, this means that the returned plans are solution plans and that if no solution plan of length n exists, they can tell you that no plan exists. Note that soundness and completeness hold for *bounded* planning problems. The fact that a plan does not exist for a bounded planning problem does not imply that no plan exists for the original planning problem.
2. The procedures based on the idea of randomized local search, called *stochastic procedures*, are sound but not complete. This means that even for a bounded planning problem they cannot return failure and guarantee therefore that a plan does not exist. However, these procedures can sometimes scale up better than complete algorithms to large problems in the case a solution does exist.

7.3.1 Davis-Putnam Procedure

The Davis-Putnam procedure, one of the first decision procedures, is still one of the most effective and widely used in the "planning as satisfiability" approach. A version of the Davis-Putnam procedure is shown in Figure 7.2. Calling Davis-Putnam(Φ,$\emptyset$), where Φ is a propositional formula, returns a model $\mu \neq \emptyset$ if Φ is satisfiable; otherwise, it returns $\mu = \emptyset$. We suppose Φ to be in Conjunctive Normal Form (CNF),[4] i.e., it is a conjunction of clauses, where a clause is a disjunction of literals. A *literal* is a propositional variable (positive literal) or its negation (negative literal). A *unit clause* is a clause with one literal. A CNF formula can thus be represented as a set of clauses, where each clause is represented as a set of literals. The model μ is represented as a set of of literals, with positive and negative literals corresponding to assignments to true and false, respectively.

Davis-Putnam performs a depth-first search through the space of all possible assignments until either it finds a model or explores the entire search space without finding any. Unit-Propagate eliminates in one shot all that can be eliminated and returns a smaller formula. Indeed, for any unit clause in the formula, i.e., for every clause composed by a single literal, if the literal is positive (negative), Unit-Propagate sets the literal to true (false) and simplifies. This step is called *unit propagation*. Then, Davis-Putnam selects a variable P to be simplified and recourses sequentially first

```
Davis-Putnam(Φ,μ)
    if ∅ ∈ Φ then return
    if Φ = ∅ then exit with μ
    Unit-Propagate(Φ,μ)
    select a variable P such that P or ¬P occurs in φ
    Davis-Putnam(Φ ∪ {P},μ)
    Davis-Putnam(Φ ∪ {¬P},μ)
end

Unit-Propagate(Φ,μ)
    while there is a unit clause {l} in Φ do
        μ ← μ ∪ {l}
        for every clause C ∈ Φ
            if l ∈ C then Φ ← Φ − {C}
            else if ¬l ∈ C then Φ ← Φ − {C} ∪ {C − {¬l}}
end
```

Figure 7.2 The Davis-Putnam satisfiability decision procedure.

4. There is a well-known transformation to convert any satisfiability problem to CNF form; see, e.g., Plaisted [442].

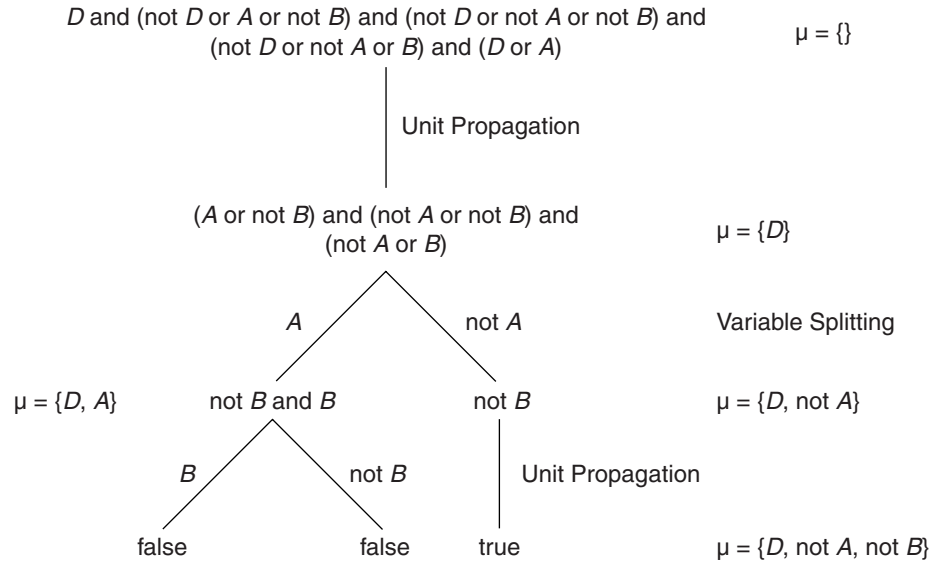

Figure 7.3 An example of a search tree constructed by Davis-Putnam.

with $P = true$ and then, if this choice fails, with $P = false$. The sequential choice of the truth assignment is a particular implementation of a nondeterministic step, where the nondeterminism resides in whether to assign true or false to the selected variable. This step is called *variable selection*.

The variable selection rule may be as simple as choosing the first remaining variable in Φ, or it may be quite sophisticated. For instance, it can select a variable occurring in a clause of minimal length, or it can select a variable with a maximum number of occurrences in minimum-size clauses. These heuristics try to eliminate clauses as early as possible in the search.

Example 7.3 Consider the following propositional formula in CNF:

$$\Phi = D \wedge (\neg D \vee A \vee \neg B) \wedge (\neg D \vee \neg A \vee \neg B) \wedge (\neg D \vee \neg A \vee B) \wedge (D \vee A)$$

In Figure 7.3 we depict the search tree dynamically constructed by Davis-Putnam. We label each node with the formula Φ that gets simplified step-by-step and with the current assignment μ. Edges represent either unit propagation or recursive calls to Davis-Putnam with a variable selection. We label edges with the literal that is selected. Given Φ and $\mu = \emptyset$, Unit-Propagate eliminates in one step the unit clause D. It returns

$$\Phi = (A \vee \neg B) \wedge (\neg A \vee \neg B) \wedge (\neg A \vee B)$$

and $\mu = \{D\}$. Let us suppose that the variable selection rule chooses A. Davis-Putnam$(\Phi \cup A, \mu)$ calls Unit-Propagate again, which eliminates A and returns

$$\Phi = \neg B \wedge B$$

with $\mu = \{D, A\}$. Unit propagation returns false ($\emptyset \in \Phi$) and Davis-Putnam backtracks by choosing $\neg A$. Unit-Propagate simplifies Φ to $\neg B$ with $\mu = \{D, \neg A\}$, and finally to the empty set. It returns $\mu = \{D, \neg A, \neg B\}$. ■

We discuss now how Davis-Putnam works when we encode a planning problem for a simplified DWR domain.

Example 7.4 Consider[5] Example 6.1, which we have used for Graphplan in Chapter 6. We have two locations (l1 and l2) and two robots (r1 and r2) that can transport containers (c1 and c2) from one location to the other (no piles, cranes, pallets, etc.). We have three operators—move(r,l,l'), load(c,r,l), and unload(c,r,l)—and the following predicates: at(r,l), in(c,l), loaded(c,r), and unloaded(r). In the initial state we have the container c1 at location l1, and c2 at l2. In the goal state we want the position of the containers to be exchanged, i.e., c1 at l2, and c2 at l1. In this simple case we know that a plan of length 6 is enough to solve the problem; therefore, we fix the plan length to $n = 6$. We encode the planning problem with Formulas 7.9 through 7.13 described in Section 7.2.3.

Davis-Putnam first applies Unit-Propagate to the unit clauses in the encoding of the initial state and of the goal states. This first step rules out the propositional variables corresponding to the actions that are not applicable in the initial state at step 0 and the actions at step 5 that do not lead to the goal state at step 6. For instance, consider one of the formulas for the operator move that is instantiated at step 0 according to formulas like Formula 7.11:

$$\text{move}(\text{r1}, \text{l2}, \text{l1}, 0) \Rightarrow \text{at}(\text{r1}, \text{l2}, 0) \wedge \text{at}(\text{r1}, \text{l1}, 1) \wedge \neg\text{at}(\text{r1}, \text{l2}, 1) \tag{7.14}$$

Because in the initial state we have the unit clause $\neg$at(r1, l2, 0), it is easy to verify that unit propagation simplifies the CNF version of Formula 7.14 to $\neg$move(r1, l2, l1, 0). Then μ, at the second iteration of unit propagation, assigns false to move(r1,l2,l1,0). This corresponds to the fact that move(r1,l2,l1,0) is not applicable in the initial state. Consider now one of the formulas for the operator unload that is instantiated at step 5 according to formulas like Formula 7.11:

$$\begin{aligned} &\text{unload}(\text{c1}, \text{r1}, \text{l1}, 5) \Rightarrow \\ &\quad \text{loaded}(\text{c1}, \text{r1}, 5) \wedge \text{at}(\text{r1}, \text{l1}, 5) \wedge \\ &\quad \text{unloaded}(\text{r1}, 6) \wedge \text{in}(\text{c1}, \text{l1}, 6) \wedge \neg\text{loaded}(\text{c1}, \text{r1}, 6) \end{aligned} \tag{7.15}$$

5. This example is available at http://www.laas.fr/planning/.

Because in the goal state we have the unit clause $\neg\mathsf{in}(\mathsf{c1},\mathsf{l1},6)$, it is easy to verify that unit propagation simplifies the CNF version of Formula 7.15 to $\neg\mathsf{unload}(\mathsf{c1},\mathsf{r1},\mathsf{l1},5)$. Then μ, at the second iteration of unit propagation, assigns false to unload(c1,r1,l1,5). This corresponds to the fact that unload(c1,r1,l1,5) does not lead to the goal; indeed, c1 is required to be in l2 in the goal state.

Unit propagation[6] in its first step also simplifies frame axioms. For instance, the frame axiom

$$\neg\mathsf{at}(\mathsf{r1},\mathsf{l1},0)\wedge\mathsf{at}(\mathsf{r1},\mathsf{l1},1)\Rightarrow\mathsf{move}(\mathsf{r1},\mathsf{l2},\mathsf{l1},0) \tag{7.16}$$

gets simplified to true by the unit clause at(r1,l1,0), while the frame axiom

$$\neg\mathsf{at}(\mathsf{r1},\mathsf{l2},0)\wedge\mathsf{at}(\mathsf{r1},\mathsf{l2},1)\Rightarrow\mathsf{move}(\mathsf{r1},\mathsf{l1},\mathsf{l2},0) \tag{7.17}$$

gets simplified by the unit clause $\neg\mathsf{at}(\mathsf{r1},\mathsf{l2},0)$ to

$$\mathsf{at}(\mathsf{r1},\mathsf{l2},1)\Rightarrow\mathsf{move}(\mathsf{r1},\mathsf{l1},\mathsf{l2},0) \tag{7.18}$$

A similar simplification is performed for frame axioms that involve instantiations at step 6.

Complete exclusion axioms get simplified at the second iteration of unit propagation. For instance, since the first iteration has simplified to $\neg\mathsf{move}(\mathsf{r1},\mathsf{l2},\mathsf{l1},0)$, complete exclusion axioms like

$$\neg\mathsf{move}(\mathsf{r1},\mathsf{l2},\mathsf{l1},0)\vee\neg\mathsf{move}(\mathsf{r2},\mathsf{l1},\mathsf{l2},0) \tag{7.19}$$

simplify to true. This intuitively corresponds to the fact that since move(r1,l2,l1,0) corresponds to an action that is not applicable in the initial state, we do not need to eliminate the case in which it could get applied in the same step with another action, in this case, move(r2,l1,l2,0).

The first call to Unit-Propagate eliminates all these possible facts. When unit propagation terminates, the variable selection phase has to select a variable to eliminate. We can choose among different propositional variables that correspond to different applications of actions in different steps or to different propositions that may hold in different steps. How the search will proceed depends on this choice. Variable selection may end up in two main cases, depending on whether it chooses a variable instantiated at step i that will actually correspond or will not correspond to either an action of a solution plan at step $i-1$ or a fact that holds after executing an action of a solution plan at step $i-1$. If this is done at each recursive call of Davis-Putnam, then the recursive application of Davis-Putnam will lead to a solution plan. Otherwise, Davis-Putnam will need to backtrack.

For instance, let us suppose that variable selection chooses move(r1,l1,l2,0) its first time (the robot needs to be loaded first). This is an unlucky choice because it

6. Here, for simplicity, we do not write formulas in the CNF version.

rules out the possibility of finding a solution plan of length 6. Davis-Putnam is called recursively until it discovers that and has to backtrack.

A good choice is instead load(c1,r1,l1,0) because the solution plan of length 6 must first load one of the two robots. Unit propagation does its job by ruling out all the actions that are not applicable after loading the robot, e.g., load(c1,r1,l1,1). Some good alternatives to load(c1,r1,l1,0) are load(c2,r2,l1,0) (loading the other robot as the first action), move(r1,l1,l2,1) (moving the first robot as the second action), unload(r2,2) (unloading the second robot as the second action), and so on. Alternative good choices for predicates are loaded(c1,r1,1), ¬unloaded(r1, 1), at(r1,l1,1), etc.

Of course, the procedure does not have to choose first the propositions instantiated at step 0, then those at step 1, etc. The first (good) choice can be, for instance, unload(c2,r2,l1,5).

In our example, Davis-Putnam, depending on the choice of variables, can return many different totally ordered plans corresponding to different possible models. Here are two possibilities:

$$\mathsf{load}(\mathsf{c1},\mathsf{r1},\mathsf{l1},0),\ \mathsf{move}(\mathsf{r1},\mathsf{l1},\mathsf{l2},1),\ \mathsf{unload}(\mathsf{c1},\mathsf{r1},\mathsf{l2},2),$$
$$\mathsf{load}(\mathsf{c2},\mathsf{r2},\mathsf{l2},3),\ \mathsf{move}(\mathsf{r2},\mathsf{l2},\mathsf{l1},4),\ \mathsf{unload}(\mathsf{c2},\mathsf{r2},\mathsf{l1},5)$$

and

$$\mathsf{load}(\mathsf{c2},\mathsf{r2},\mathsf{l2},0),\ \mathsf{load}(\mathsf{c1},\mathsf{r1},\mathsf{l1},1),\ \mathsf{move}(\mathsf{r2},\mathsf{l2},\mathsf{l1},2),$$
$$\mathsf{move}(\mathsf{r1},\mathsf{l1},\mathsf{l2},3),\ \mathsf{unload}(\mathsf{c2},\mathsf{r2},\mathsf{l1},4),\ \mathsf{unload}(\mathsf{c1},\mathsf{r1},\mathsf{l2},5)$$

Notice that in this case a partially ordered plan (in the style of Graphplan) would help a lot to keep the planning length shorter. Indeed, we could devise a partially ordered plan of length 3: we load, move, and unload the two robots. The use of complete exclusion axioms prevents the possibility of searching for partially ordered plans. In Section 7.4 we will describe a different kind of encoding that allows for partially ordered plans. This encoding is called *conflict exclusion* and is similar in spirit to the mutual exclusion (mutex) relations of Graphplan. ■

Proposition 7.2 *Davis-Putnam is sound and complete.*

7.3.2 Stochastic Procedures

Davis-Putnam works with *partial assignments*: at each step, not all the variables are assigned a truth value. At the initial step, μ is empty; then it is incrementally constructed by adding assignments to variables. An alternative idea is to devise algorithms that work from the beginning on *total assignments*, i.e., on assignments to all the variables in Φ. A trivial algorithm based on this idea is the one that

randomly selects an initial total assignment, checks whether there is a model, and, if not, iteratively selects a different total assignment until a model is found or the space of all possible total assignments is exhausted. This is of course a sound and complete procedure, but it is of no practical use because it searches randomly through the huge space of all possible assignments. However, it can be used as the basic idea underlying a set of *incomplete satisfiability decision procedures* that have been shown experimentally to scale up to rather large problems.

Typically, incomplete procedures employ some notion of *randomized local search*. Figure 7.4 presents an algorithm based on randomized local search. Local-Search-SAT selects randomly a total assignment μ and, if it is not a model, tries to improve the choice by replacing it with a "better" assignment μ'. The assignment is "better" according to a function cost $\mathsf{Cost}(\mu,\Phi)$ that typically is the number of clauses of Φ that are not satisfied by μ. Condition $|\mu - \mu'| = 1$ imposes that the search is local because μ and μ' must differ for the assignment to one variable. In other words, μ' is obtained from μ by *flipping* one variable, i.e., by changing the assignment to one variable in Φ. It is easy to show that Local-Search-SAT is sound and incomplete. Incompleteness is due to the possible existence of local minima.

A widely used variation of Local-Search-SAT is the GSAT algorithm. A basic version of GSAT is presented in Figure 7.5. At each step of Basic-GSAT, one variable is flipped: $\mathsf{Flip}(P, \mu)$ returns the truth assignment obtained by flipping the variable P (this is equivalent to computing a μ' at hamming distance 1 from μ, see Local-search-SAT). The new truth assignment is selected by flipping the variable that leads to the best neighboring assignment: $\arg\min_{\mu_P} \mathsf{Cost}(\mu_P, \Phi)$ returns a μ_P such that $\mathsf{Cost}(\mu_P, \Phi)$ is minimal. Notice that, unlike Local-Search-SAT, Basic-GSAT does not impose any condition on the fact that the assignment must be improved, i.e., it does not impose that the number of unsatisfied clauses decreases. Indeed, even the best flip can increase the number of unsatisfied clauses. In this aspect, Basic-GSAT differs significantly from the classical notion of local search, in which an improvement is made at every step and search is terminated when no improving

```
Local-Search-SAT(Φ)
    select a total assignment μ for Φ randomly
    while μ does not satisfy Φ do
        if ∃μ′ s.t. Cost(μ′,Φ) < Cost(μ,Φ) and |μ − μ′| = 1
            then μ ← μ′
            else return failure
    return μ
```

Figure 7.4 A SAT algorithm based on randomized local search.

```
Basic-GSAT(Φ)
    select μ randomly
    while μ does not satisfy Φ do
        for every P ∈ Φ, μ_P ← Flip(P,μ)
        μ ← arg min_{μ_P} Cost(μ_P, Φ)
    return μ
```

Figure 7.5 A basic version of GSAT.

step is possible. One motivation for this choice is that it helps avoid local minima. Experimental results have indeed shown that GSAT manages often to avoid local minima, while procedures restricted to improving steps (like Local-Search-SAT) perform poorly. Basic-GSAT is not guaranteed to terminate. In the real implementations of GSAT, the algorithm *restarts* with a new initial guess after a maximum number of flips and terminates after a maximum number of restarts. Moreover, many different variations have been devised to try to avoid the local minima problem, including backtracking, different heuristics for restarts, and random flips.

Other alternative incomplete methods, still based on local search, are the algorithms based on the so-called *iterative repair approach*. The basic idea is to iteratively modify a truth assignment such that it satisfies one of the unsatisfied clauses selected according to some criterion. An unsatisfied clause is seen as a "fault" to be "repaired." This approach differs from local search in that, at every step, the number of unsatisfied clauses (the typical cost) may increase. In Figure 7.6, we present a generic algorithm based on this idea. It is easy to show that Iterative-Repair is sound and incomplete.

A well-known version of an iterative repair algorithm is Random-walk, which implements the step "modify μ to satisfy C" in a way that resembles GSAT, i.e., by

```
Iterative-Repair(Φ)
    select any μ
    while μ does not satisfy Φ do
        if iteration limit exceeded then return failure
        select any clause C ∈ Φ not satisfied by μ
        modify μ to satisfy C
    return μ
```

Figure 7.6 A general iterative repair algorithm.

flipping iteratively one variable in C. While Random-walk can be shown to solve any satisfiable CNF formula, it has been shown experimentally to suffer several problems on formulas of a certain complexity. However, a variation of Random-walk called Walksat has been shown experimentally to perform better than local search and GSAT. Walksat is a "probabilistically greedy" version of Random-walk. After C is selected randomly, Walksat selects randomly the variable to be flipped among the two following possibilities: (1) a random variable in C or (2) the variable in C that leads to the greatest number of satisfied clauses when flipped. The intuition underlying Walksat is that mixing up randomly nongreedy (case 1) and greedy (case 2) choices provides the ability to move toward a better choice while avoiding local minima as much as possible.

Conceptually, Walksat has similarities with another incomplete method, Simulated-Annealing, which has been shown to perform comparably. Simulated-Annealing is a method for finding global minima of an energy function. It was originally devised as a method for combinatorial optimization (i.e., for determining the minimum value of a function with several independent variables) and has been applied for the analysis of the equations of state in n-body systems (e.g., to analyze how liquids freeze or metals recrystallize during the process of annealing). In the satisfiability problem, the value of the energy can be seen as the number of unsatisfied clauses in the propositional formula. Simulated-Annealing performs the following steps.

1. Select an initial assignment randomly.
2. Select randomly any variable P.
3. Compute δ, i.e., the change in the number of unsatisfied clauses when P is flipped.
4. If $\delta \leq 0$, make the flip.
5. If $\delta > 0$, make the flip with probability $e^{\delta/T}$, where T is the temperature and is usually a decreasing function of the number of steps taken. For instance, the temperature $T(1)$ at the first step can be a predefined high value (the maximal temperature), and the temperature at step $i+1$ can be computed as

$$T(i+1) = T(i) - \frac{\Delta T}{i}$$

where ΔT is the difference between the maximal value of the temperature (i.e., $T(1)$) and a predefined minimal value.

Example 7.5 Consider the same propositional formula used in Example 7.3:

$$\Phi = D \wedge (\neg D \vee A \vee \neg B) \wedge (\neg D \vee \neg A \vee \neg B) \wedge (\neg D \vee \neg A \vee B) \wedge (D \vee A)$$

We describe some of the possible executions of the different stochastic procedures discussed in this section.

- Local-Search-SAT selects a random total assignment first. Let us suppose the assignment is $\mu = \{D, A, B\}$. Under this assignment, the only unsatisfied clause is $(\neg D \vee \neg A \vee \neg B)$, and therefore $\mathsf{Cost}(\mu,\Phi) = 1$. There is no μ' such that $|\mu - \mu'| = 1$ and $\mathsf{Cost}(\mu,\Phi) < 1$. Local-Search-SAT terminates execution with failure. It behaves differently if the initial guess is different, for instance, if $\mu = \{\neg D, A, \neg B\}$, then it finds the model in one iteration step.
- Basic-GSAT selects a random total assignment. Let us suppose it starts also with the initial guess $\mu = \{D, A, B\}$. Different than Local-Search-SAT, Basic-GSAT has two alternatives. It can flip either variable D or B because the corresponding cost is 1 (flipping A has a cost of 2). If D is flipped, then the alternative is to flip D again or to flip B. If B is flipped, then one of the alternatives is to flip A. If this alternative is taken, the solution is found.
- Iterative-Repair, with the same initial guess, has to repair clause $(\neg D \vee \neg A \vee \neg B)$. Different assignments can repair this clause. One is the solution $\mu = \{D, \neg A, \neg B\}$. If this is not selected, Iterative-Repair has to reiterate. Let us suppose $\mu = \{\neg D, \neg A, \neg B\}$ is selected. We then have two clauses to repair: D and $(D \vee A)$. If D is selected, then Iterative-Repair finds the solution by repairing it. If $D \vee A$ is selected, then Iterative-Repair has two different possibilities, and one of them leads to the solution.

■

7.4 Different Encodings

In Section 7.2, we presented one particular encoding of a bounded planning problem to a satisfiability problem. The encoding determines the number of propositional variables and the number of clauses in the satisfiability problem, i.e., the variables and the clauses in the formula Φ that is generated. Since the efficiency of satisfiability decision procedures (both complete and incomplete ones) depends on the number of variables and clauses (e.g., Walksat takes time exponential in the number of variables), the choice of encoding is critical. The encoding in Section 7.2 is based on two main choices.

1. The *encoding of actions*. Each ground action is represented by a different logical variable at each step. For instance, we encode operator $\mathsf{move}(r,l,l')$ with the propositional variables $\mathsf{move}(\mathsf{r1},\mathsf{l1},\mathsf{l2},i)$, $\mathsf{move}(\mathsf{r1},\mathsf{l2},\mathsf{l1},i)$, etc. This results in $|A| = n|O||D|^{A_0}$ variables, where n is the number of steps, $|O|$ is the number of operators, $|D|$ is the number of constants in the domain, and A_0 is the maximum arity of operators. One could devise a different type of encoding, e.g., provide m bits that can encode each ground action. This encoding is called "bitwise." For example, if we have four ground actions—$a_1 = \mathsf{move}(\mathsf{r1},\mathsf{l1},\mathsf{l2},i)$, $a_2 = \mathsf{move}(\mathsf{r1},\mathsf{l2},\mathsf{l1},i)$, $a_3 = \mathsf{move}(\mathsf{r2},\mathsf{l1},\mathsf{l2},i)$, and $a_4 = \mathsf{move}(\mathsf{r2},\mathsf{l2},\mathsf{l1},i)$—we can use just two bits: $\mathsf{bit1}(i)$ and $\mathsf{bit2}(i)$.

The formula $\mathsf{bit1}(i) \wedge \mathsf{bit2}(i)$ can represent a_1, $\mathsf{bit1}(i) \wedge \neg\mathsf{bit2}(i)$ represents a_2, etc. This reduces the number of variables to $\lceil \log_2 |A| \rceil$.

2. The *encoding of the frame problem.* We have chosen *explanatory frame axioms*, i.e., formulas that, for each fluent, enumerate the set of actions that could have occurred in order to account for a state change (see Formula 7.12). We could have used the most natural way to formalize the fact that actions change only the values of fluents that are in their effects by generating formulas that, for each action, state that the fluents that do not appear in their effects remain the same. For instance, for action move(r1,l1,l2), we could have a formula like $\mathsf{unloaded}(\mathsf{r1}, i) \wedge \mathsf{move}(\mathsf{r1}, \mathsf{l1}, \mathsf{l2}, i) \Rightarrow \mathsf{unloaded}(\mathsf{r1}, i+1)$, which states that moving an unloaded robot leaves the robot unloaded. The formulas resulting from this way of encoding are called *classical frame axioms.* In the case of classical frame axioms, the number of clauses in Φ is $O(n, |F|, |A|)$, where $|F|$ is the number of ground fluents. In the case of explanatory frame axioms, we have just $O(n, |F|)$ clauses if actions are represented as described in Section 7.2.

Different encodings can be classified depending on how actions are represented and how the frame problem is formalized. In the following we review some of the main alternatives proposed in the literature.

7.4.1 Action Representation

We discuss four alternatives for encoding actions.

Regular. This is the encoding proposed in Section 7.2: each ground action is represented by a different logical variable. It results in $|A| = n|O||D|^{A_0}$ variables.

Simple Operator Splitting. The idea is to replace each n-ary action ground proposition with n unary ground propositions. For instance, the propositional variable move(r1,l1,l2,i), resulting from the regular action representation, is replaced by $\mathsf{move1}(\mathsf{r1}, i) \wedge \mathsf{move2}(\mathsf{l1}, i) \wedge \mathsf{move3}(\mathsf{l2}, i)$. Intuitively, the advantage is that instances of each operator share the same variables. For instance, the same variable move2(l1,i) is used to represent move(r1,l1,l2,i) can be reused to represent move(r2,l1,l3,i) without generating a number of variables that is exponential on the arity of an operator. In other words, the same variable can be used to represent different cases where the starting location of the move operation is the same. Consider, for instance, the case in which we have three robots and three locations. We need $3^3 = 27$ variables in the regular representation, while we need just $3 + 3 + 3 = 9$ variables with the simple operator splitting. In general, simple operator splitting results in $n|O||D|A_0$ variables.

Overloaded Operator Splitting. This generalizes the idea of simple operator splitting by allowing different operators to share the same variable. This is done by representing the action (e.g., move) as the argument of a general action predicate Act. For instance, move(r1,l1,l2,i) is replaced by Act(move, i) $\wedge$ Act1(r1, i) $\wedge$ Act2(l1, i) $\wedge$ Act3(l2, i). Notice that a different action, like fly(r1,l1,l2,i), can share the same variables of move(r1,l1,l2,i): Act(fly, i) $\wedge$ Act1(r1, i) $\wedge$ Act2(l1, i) $\wedge$ Act3(l2, i). Overloaded splitting further reduces the number of variables to $n(|O| + |D|A_0)$.

Bitwise. This is the technique already introduced in this section. $n\lceil \log_2 |A| \rceil$ propositional variables can be used in a sort of binary encoding. For instance, in order to represent 30 ground actions, we need just 5 bits because $2^5 = 32$.

The conclusion of this analysis on action representation could let us think that the bitwise encoding is the most convenient. This is not true! The number of variables is not the only cause of complexity that satisfiability decision procedures have to deal with. The complexity depends also on the number of clauses generated, their length, and other factors. Moreover, action representation is not the only choice we have. Depending on the frame axioms we choose, one or another action representation might be more or less convenient. In the next subsection, we proceed along the other dimension of the space of encodings: frame axioms.

7.4.2 Frame Axioms

Let us discuss in more detail the two different possibilities we have already introduced: classical and explanatory frame axioms.

Classical Frame Axioms. This is the most obvious formalization of the fact that actions change only what is explicitly stated. With this choice, formulas like Formula 7.12 are replaced with formulas of the following kind. For each action a, for each fluent $f \notin \mathsf{effects}(a)$, and for each $0 \leq i \leq n-1$, we have:

$$f_i \wedge a_i \Rightarrow f_{i+1} \tag{7.20}$$

However, such a replacement is not enough. Indeed, notice that if action a_i does not occur at step i, a_i is false and Formula 7.20 is trivially true. Classical frame axioms do not constrain the value of f_{i+1}, which can therefore take an arbitrary value. The final consequence is that we have unintended models. For instance, consider this classical frame axiom:

$$\mathsf{unloaded}(\mathsf{r1}, i) \wedge \mathsf{move}(\mathsf{r1}, \mathsf{l1}, \mathsf{l2}, i) \Rightarrow \mathsf{unloaded}(\mathsf{r1}, i) \tag{7.21}$$

When the robot is not moved from l1 to l2 at step i the robot might become loaded "magically." We do not want this! A solution is to add the *at-least-one axioms*, i.e.,

a disjunction of every possible ground action at step i, that assures that at least one action is performed:[7]

$$\bigvee_{a \in A} a_i \tag{7.22}$$

While classical frame axioms force us to add at-least-one axioms, they avoid complete exclusion axioms (Formula 7.13) when we need to state that just one action occurs at a time. Indeed, classical frame axioms like Formula 7.20 combined with the axioms on action preconditions and effects (Formula 7.11) ensure that any two actions that occur at step i lead to an identical state at step $i + 1$. Therefore, if more than one action occurs in a step, then either one can be selected to form a valid plan.

Explanatory Frame Axioms. We used this kind of axiom in Section 7.3, Formula 7.12. No at-least-one axioms are required. On the other hand, we need to state complete exclusion axioms (Formula 7.13) to ensure that just one action occurs at a given step. The result is that we are searching for totally ordered plans, which might not be the best thing to do. We have seen that other planning approaches (e.g., Graphplan) take strong advantage of searching in the space of partial-ordered plans. While classical frame axioms impose total order, explanatory frame actions can allow for partial order. This can be done by limiting exclusion axioms to *conflicting actions*. The notion of conflicting actions is similar to the one provided in Graphplan: two actions are conflicting if one's precondition is inconsistent with the other's effect. We have axioms of the form $\neg a_i \vee \neg b_i$ only if a and b are conflicting. This form of encoding is called *conflict exclusion*, as opposed to *complete exclusion*.

In Figure 7.7, we report the worst-case size for the different possible encodings of actions and frame axioms. The size of the encoding is measured in terms of number of variables and number of clauses. The conversion, compared with the original problem statement (see Chapter 2), grows the size of the problem statement exponentially. Figure 7.7 gives an idea of the complexity of the formula resulting from the encodings. However, as reported in Ernst *et al.* [172], practical experimentation shows that some of the expectations from this complexity analysis are not fulfilled. From several experiments in planning domains from the literature, regular explanatory and simple splitting explanatory encodings are the smallest ones. Indeed, explanatory frame axioms are in practice smaller than classical ones because they state only what changes, rather than what does not change, when an action occurs, and an action usually affects just a few fluents. Regular explanatory encodings allow for parallel actions and as a consequence for shorter plans. Moreover, conflict exclusion axioms are a subset of complete exclusion axioms. It is surprising that bitwise encoding does not give advantages in practice. In the worst

7. Axioms like Formula 7.22 are not necessary with a bitwise representation of actions because all the models must assign values to bit1(i) such that at least one action is executed.

Actions	Number of variables
Regular	$n\|F\| + n\|O\|\|D\|^{A_0}$
Simple splitting	$n\|F\| + n\|O\|\|D\|A_0$
Overloaded splitting	$n\|F\| + n(\|O\| + \|D\|A_0)$
Bitwise	$n\|F\| + n\lceil \log_2 \|O\|\|D\|^{A_0} \rceil$

Action	Frame axiom	Number of clauses
Regular	Classical	$O(n\|F\|\|A\|)$
Regular	Explanatory	$O(n\|F\|\|A\| + n\|A\|^2)$
Simple splitting	Classical	$O(n\|F\|\|A\|A_0 + n\|A\|A_0{}^{\|A\|})$
Simple splitting	Explanatory	$O(n\|F\|A_0{}^{\|A\|} + n(\|A\|A_0)^2)$
Overloaded splitting	Classical	$O(n\|F\|\|A\|A_0 + n\|A\|A_0{}^{\|A\|})$
Overloaded splitting	Explanatory	$O(n\|F\|(\|A\|A_0)^2 + n\|F\|\|A\|A_0{}^{\|A\|})$
Bitwise	Classical	$O(n\|F\|\|A\| \log_2 \|A\|)$
Bitwise	Explanatory	$O(n\|F\|\|A\|(\log_2 \|A\|)^{\|A\|})$

Figure 7.7 Size of the different encodings: n is the number of steps of the plan; $|F|$ is the number of fluents, with $|F| = |P||D|^{A_P}$, where $|P|$ is the number of predicate symbols, $|D|$ is the number of constants in the domain, and A_P is the maximum arity of predicates; $|O|$ is the number of operators; A_0 is the maximum arity of operators; and $|A| = |O||D|^{A_0}$ is the number of ground actions.

case, it has the smallest number of variables. A possible explanation is that bitwise encoding cannot take advantage of possible simplifications during the search. Operator splitting allows for reasoning about parts of actions, which represent a sort of generalization of many possible instantiations. It is possible to deduce more from generalized formulas than from fully instantiated ones.

7.5 Discussion and Historical Remarks

The original idea of "planning as satisfiability" is due to Kautz and Selman [313, 314] and was subsequently incorporated into a planner called BlackBox [315]. Different possible encodings were first investigated by the work by Kautz, McAllester, and Selman [312]. A rationalization and further investigation of the encoding problem and of its automatic generation is due to Ernst, Millstein, and Weld [172]. Along these lines, different methods and encodings have been proposed (e.g., [232]). The ability to add control knowledge to gain efficiency has been shown in two papers [122, 315]. Finally, the approach has been extended to resource handling [556]. A good survey of complete and incomplete methods for deciding satisfiability can be found in Cook and Mitchell [132]. A detailed description and discussion of the simulated annealing method can be found in other sources [62, 284, 469].

The "planning as satisfiability" approach has demonstrated that general purpose logic-based reasoning techniques can be applied effectively to plan synthesis.[8] In early attempts, general purpose logic-based planning was done by deduction, e.g., by first-order resolution theorem proving (see the seminal work by Green in 1969 [249]). Deduction, however, could not compete with algorithms designed specifically for planning, such as search-based algorithms in the state or plan space. In the first AIPS Planning Competition in 1998, the BlackBox planning system [315] was competitive with the most efficient planners, including Graphplan and planners that do heuristic search. This was mainly for two main reasons: (1) the ability to exploit recent improvements in the performance of general purpose algorithms for propositional satisfiability [244, 469], and (2) probably most importantly, the use of an encoding based on conflict exclusion axioms based on the idea of mutex in Graphplan, called *Graphplan-based encoding* (see Section 7.4). Graphplan has since actually been used to efficiently generate the SAT encoding [41, 316].

In this chapter we have focused on planning in classical planning problems. The approach has been extended to deal with different kinds of problems (see Chapter 18). Finally, note that the SAT encoding is a particular case of the CSP encoding (see Chapter 8).

7.6 Exercises

7.1 Are the following formulas satisfiable?

$$(\neg D \vee A \vee \neg B) \wedge (\neg D \vee \neg A \vee \neg B) \wedge (\neg D \vee \neg A \vee B) \wedge (D \vee A)$$

$$(D \rightarrow (A \rightarrow \neg B)) \vee (D \wedge (\neg A \rightarrow \neg)B) \wedge (\neg D \vee \neg A \vee B) \wedge (D \rightarrow A)$$

Run the Davis-Putnam procedure on them and explain the result. Also run a stochastic procedure.

7.2 Complicate Example 7.1 (see page 150) with a loading and unloading operation. Describe the regular, explanatory, and complete exclusion encodings. How many clauses do the encodings generate?

7.3 Consider Example 7.1 (see page 150) with two robots. Write the different possible encodings.

7.4 Describe the different possible encodings of the complete DWR example. For each encoding, how many clauses do we have?

7.5 Can conflict exclusion be used with simple and overloading action representation? Why?

7.6 Can you find an example of a planning problem where Davis-Putnam is more effective than the stochastic procedures? Vice versa?

8. Notice, however, that the reasoning technique used in SAT is much closer to, indeed a particular case of, CSP techniques (see Chapter 8) than logical deduction.

7.7 In Example 7.1 (see page 150), suppose you do not know in which location the robot is initially. Suppose the planning problem is to find a plan such that no matter where the robot is initially, the plan leads to the goal "the robot must be in l2." Can you find an encoding such that the problem can be formulated as a satisfiability problem?

7.8 Run BlackBox (http://www.cs.washington.edu/homes/kautz/blackbox) on the DWR domain. Check the size of the encoding.

7.9 Describe the possible encodings of the planning problem described in Exercise 2.1. Run BlackBox on this planning problem.