# 9
# Planning Agents

In the next several chapters we discuss the application of the methodology for representing knowledge about dynamic domains and ASP programming to the design of intelligent agents capable of acting in a changing environment. The design is based on the agent architecture from Section 1.1. In this chapter we address *planning* – one of the most important and well studied tasks that an intelligent agent should be able to perform (see step 3 of the agent loop from Section 1.1).

## 9.1 Classical Planning with a Given Horizon

We start with **classical planning** in which a typical problem is defined as follows:

- A **goal** is a set of fluent literals that the agent wants to become true.
- A **plan** for achieving a goal is a sequence of agent actions that takes the system from the current state to one that satisfies this goal.
- **Problem**: Given a description of a deterministic dynamic system, its current state, and a goal, find a plan to achieve this goal.

A sequence $\alpha$ of actions is called a *solution* to a classical planning problem if the problem's goal becomes true at the end of the execution of $\alpha$.

In this chapter we show how to use ASP programming techniques to solve a special case of the classical planning problem in which the agent has a limit on the length of the allowed plans. The limit is often referred to as the **horizon** of the planning problem.

To solve a classical planning problem $\mathcal{P}$ with horizon $n$, we construct a program $plan(\mathcal{P}, n)$ such that solutions of $\mathcal{P}$ whose length do not exceed $n$ correspond to answer sets of $plan(\mathcal{P}, n)$. The program consists of ASP encodings of the system description of $\mathcal{P}$, the current state, and the goal, together with a small, domain-independent ASP program called the **simple**

**planning module**. The encoding of the system description is identical to the one used for temporal projection (meaning that, variable $I$ for steps ranges over integers from 0 to $n$). (In the original encoding, it was enough for us to consider $n = 1$.) The encoding of the current state is identical to the encoding of the initial state from Chapter 8.

To encode the goal, we first introduce a new relation $goal(I)$ that holds if and only if all fluent literals from the problem's goal $G$ are satisfied at step $I$ of the system's trajectory. This relation can be defined by a rule:

```
goal(I) :- holds(f_1,I), ..., holds(f_m,I),
           -holds(g_1,I), ..., -holds(g_n,I).
```

where $G = \{f_1, \dots, f_m\} \cup \{\neg g_1, \dots, \neg g_n\}$.

Now let us describe the domain-independent part of the program (i.e., the simple planning module). The first two rules define the success of the search for a plan:

```
success :- goal(I),
           I <= n.
:- not success.
```

The first rule defines success as the existence of a step in the system's trajectory that satisfies the relation $goal$. The second states that failure is not acceptable – a plan satisfying the goal should be found.

The second part of the planning module generates sequences of actions of the appropriate length that can possibly be the desired plans. One can imagine that the consequences of the execution of such a sequence in the current state are computed by the encoding of the system description of the problem. If these consequences include $success$, a plan is found. (Actual computation of plans is, of course, quite different – this is just one possible way to think about the program's behavior.)

In Gringo syntax, the corresponding generator can be written as a simple choice rule:

```
1{occurs(A,I): action(A)}1 :- step(I),
                              not goal(I),
                              I < n.
```

This guarantees that every answer set of our program includes an action sequence containing exactly one occurrence of an action at each step prior to the goal being achieved, and no occurrences of actions after the goal has been achieved. Note that if $n$ is equal to the length of the shortest plan, then the planner will produce all plans of length $n$. For $n$ larger than the length of the shortest plan, some of the plans produced will be longer than necessary – they may include irrelevant actions executed before the goal is achieved.

A similar effect can be achieved with DLV as shown next. The choice rule is replaced by several new rules where, as before, `n` stands for the horizon.

```
success :- goal(I),
           I <= n.
:- not success.

occurs(A,I) | -occurs(A,I) :- action(A), step(I),
                              not goal(I),
                              I < n.

%% Do not allow concurrent actions:
:- action(A1), action(A2),
   occurs(A1,I),
   occurs(A2,I),
   A1 != A2.

%% An action occurs at each step before
%% the goal is achieved:

something_happened(I) :- occurs(A,I).

:- step(J),
   goal(I), not goal(I-1),
   J < I,
   not something_happened(J).
```

The encoding of the simple planning module is the last step in the construction of our program $plan(\mathcal{P}, n)$. The following proposition establishes the relationship between answer sets of this program and the solutions of $\mathcal{P}$.

**Proposition 9.1.1.** *Let $\mathcal{P}$ be a classical planning problem with a deterministic system description and let $0 < n$. A sequence of actions $a_0, \ldots, a_k$ where $0 \leq k < n$ is a solution of $\mathcal{P}$ with the horizon $n$ iff there is an answer set $S$ of $plan(\mathcal{P}, n)$ such that*

*(i) For any $0 < i \leq k$, $occurs(a_i, i-1) \in S$,*
*(ii) $S$ contains no other atoms formed by $occurs$.*

As mentioned earlier, if a horizon $n$ is larger than the shortest plan needed to satisfy our goal, the planner may find that the plans contain irrelevant, unnecessary actions. To avoid this problem, we can use the planner to look for plans of lengths 1,2, and so on, until a plan is found. This can be accomplished by finding answer sets of programs $plan(\mathcal{P}, k)$ with $k = 1, 2, \dots, n$. If $k = m$ is the smallest number for which this program is consistent, the shortest solutions of the planning problem $\mathcal{P}$ are given by answer sets of $plan(\mathcal{P}, k)$. If $plan(\mathcal{P}, k)$ is inconsistent for every $1 \leq k \leq n$, then problem $\mathcal{P}$ has no solution. There is no known way of automatically finding a minimal-length plan without multiple calls; however, Section 9.5 describes two simple extensions of ASP that achieve the task.

## 9.2 Examples of Classical Planning

### 9.2.1 Planning in the Blocks World

Let us start with considering the blocks world from Section 8.5.2. The ASP encoding of the system description of this domain is given by program `bw.lp` from that section. Our goal is to create a new program, `bwplan.lp`, which will find plans for the blocks world. The main thing we have to do is add one of the simple planning modules we just defined to `bw.lp`. Since our transition system in `bw.lp` was only defined for steps 0 and 1, we need to set the horizon by changing the value of `n` from 1 to 8:

```
#const n=8.
step(0..n).
```

As an added convenience, `gringo` (and therefore `clingo`) allows us to override the constant definition from the command line. Therefore, whenever we wish to run the program with a different horizon, we could simply change our call. For example, the call

```
clingo 0 -c n=9 bwplan.lp
```

would set the horizon to 9.

Further, we wish to format the output by adding a show statement to display only the positive *occurs* statements.

```
#show occurs/2.
```

These extract the plans from the corresponding answer sets. If the program is written in DLV, use option `-pfilter=occurs`.

We now have our blocks world planner, `bwplan.lp`. Try giving it a variety of initial states and goals. For example, you can use the initial state as encoded in Section 8.1 on lines 21–33, and the goal defined by the following rule:

```
goal(I) :-
  holds(on(b4,t),I), holds(on(b6,t),I),
  holds(on(b1,t),I), holds(on(b3,b4),I),
  holds(on(b7,b3),I), holds(on(b2,b6),I),
  holds(on(b0,b1),I), holds(on(b5,b0),I).
```

Let the horizon of our planning problem be $n = 8$. Now we have a planning problem and its logic programming encoding.

The call,

```
clingo 0 bwplan.lp
```

will find the problem's solutions. For instance, one of the answer sets of the program contains the following sequence:

```
occurs(put(b2,t),0)
occurs(put(b4,b1),1)
occurs(put(b3,b4),2)
occurs(put(b7,b3),3)
occurs(put(b6,t),4)
occurs(put(b2,b6),5)
occurs(put(b0,b1),6)
occurs(put(b5,b0),7)
```

To make the output more readable, we normally sort the predicates with respect to the second parameter. This may or may not be done by the solver, which knows nothing about the significance of this order.

If we try to run the same program with $n = 7$, we discover that the program is inconsistent. Hence, the earlier sequence is a shortest solution of our problem.

Now let's change $n$ to 9 and ask our solver for all possible answer sets. It outputs quite a few. Some of them only have eight steps. Here is one with nine steps:

```
occurs(put(b2,t),0)
occurs(put(b7,t),1)
occurs(put(b6,t),2)
occurs(put(b4,t),3)
occurs(put(b3,b4),4)
occurs(put(b0,b1),5)
occurs(put(b5,b0),6)
occurs(put(b7,b3),7)
occurs(put(b2,b6),8)
```

Notice that this plan contains a wasteful action; by putting $b2$ on the table first, the planner misses the opportunity to put it directly on $b6$ at a later time. As we noted earlier, there is nothing in the code that insists that the planner find *only* the shortest plan – just that it not exceed the horizon. Therefore, it finds all plans that meet the conditions – shortest and otherwise.

Program `bwplan.lp` is a typical example of **Answer Set Planning**. It consists of the theory of the blocks world, the description of the initial state, the goal, the horizon, and the planning module. Our solution is completely independent of the problem description. We can change the initial state, the goal, and the horizon at will, without changing the rest of the program. Note that just because the theory is independent of the planning module does not mean that we cannot use specific domain knowledge to guide a planner. *The separation between domain knowledge and search strategy makes it easy to write domain-specific rules describing actions that can be ignored in the search.* This is covered in Section 9.3. As we see later, the blocks theory is also completely independent of planning – it can be used for multiple purposes.

Answer set planning does not require any specialized planning algorithm. The "planning" query is answered by the same reasoning mechanism used for other types of queries. *Therefore, the planning program can be easily generalized and improved.*

To illustrate, we begin by giving a few more examples of planning in the blocks world, followed by examples in a variety of other domains.

**Example 9.2.1.** *(Multiple Goal States)*
Note that each block of our domain is accounted for in the previous goal and there is only one possible goal state. Having only one goal state is certainly not a requirement, and it is perfectly fine to have multiple goal states. For example, we may only wish to require that $b3$ be on the table, in which case we write:

```
goal(I) :- holds(on(b3,t),I).
```

We let $n = 2$ and call the new program `bwplanb3.lp`. The solutions of the new problem can be found by calling an ASP solver. For example,

```
clingo 0 bwplanb3.lp

Answer: 1
occurs(put(b2,b4),0) occurs(put(b3,t),1)
Answer: 2
occurs(put(b2,b7),0) occurs(put(b3,t),1)
Answer: 3
occurs(put(b2,t),0) occurs(put(b3,t),1)
```

**Example 9.2.2.** *(Using Defined Fluents in the Goal)*
Let us now consider an extension of the system description of the blocks world by a new defined fluent, $occupied(block)$. The corresponding definition is given by the following state constraint:

$$occupied(B) \textbf{ if } on(B_1, B)$$

We create a new program, `occupied.lp`, that incorporates this concept into our current blocks-world planner as specified by `bwplan.lp`. First, we expand our planner by the logic programming translation of this law; i.e., we add statements

```
1 fluent(defined, occupied(B)) :- block(B).
2 holds(occupied(B),I) :- block(B),
3                         holds(on(B1,B),I).
```

Note that rule

```
-holds(occupied(B),I) :- block(B), step(I),
                         not holds(occupied(B),I).
```

is not necessary because we already have the CWA for defined fluents in `bwplan.lp`.

Suppose now that we want blocks $b0$ and $b1$ to be unoccupied, but we do not care about the rest of the blocks. We state the goal as follows:

```
4 goal(I) :- -holds(occupied(b0),I),
5            -holds(occupied(b1),I).
```

Running program `occupied.lp` (`bwplan.lp` plus lines 1–5) with horizon 3 and the usual initial state, we get 45 answer sets corresponding to the shortest plans. Here are three examples:

```
Answer: 1
occurs(put(b4,t),0) occurs(put(b2,b7),1)
    occurs(put(b3,b2),2)
Answer: 2
```

```
occurs(put(b4,t),0) occurs(put(b2,b4),1)
    occurs(put(b3,b2),2)
Answer: 3
occurs(put(b4,t),0) occurs(put(b2,t),1)
    occurs(put(b3,b2),2)
```

Note that setting the horizon to 2 correctly produces no answer sets and setting it to 4 gives 837 answer sets, most of which contain a useless action.

This example shows that defined fluents can be useful in stating our goals. In fact, we may wish to define fluents specifically for this purpose.

**Example 9.2.3.** *(Defining Complex Goals)*
Further extension of the basic blocks-world domain can be obtained by supplying blocks with colors. This can be done by simply expanding our blocks-world $\mathcal{AL}$ system description from the previous example by a new sort, say *color*, with values $white$ and $red$ and a new static relation

$$is\_colored(B, C)$$

that holds iff block $B$ is of color $C$. We assume that each block can have at most one color; i.e., the states of our domain should satisfy the following requirement:

$$\neg is\_colored(B, C_1) \textbf{ if } is\_colored(B, C_2), C_1 \neq C_2.$$

A possible goal for such a domain could be the general requirement that all towers must have a red block on top of them. To express this we use a defined fluent $wrong\_config$ (wrong configuration) that holds iff there is an unoccupied block $B$ that is not red. (Of course, a block is unoccupied if and only if it is located on top of some (possibly empty) tower.) This can be specified using the fluent *occupied* from Example 9.2.2:

$$\begin{array}{l} fluent(defined, wrong\_config(B)) \\ wrong\_config \textbf{ if } \neg occupied(B), \neg is\_colored(B, red). \end{array}$$

The goal of our problem is of the form:

```
goal(I) :- -holds(wrong_config,I).
```

All these examples show that our planning methods have a reasonable degree of elaboration tolerance with respect to possible extensions of the initial planning domain and modifications of goals and initial situations. Note also that our translation of system descriptions into ASP programs is modular (i.e., an extension of a system description is simply translated into ASP and added to the original program without necessitating further changes).

The next two examples show the application of our planning methodology to very different domains:

### *9.2.2 Igniting the Burner*

Consider the following domain:

A burner is connected to a gas tank through a pipeline. The gas tank is on the left-most end of the pipeline and the burner is on the right-most end (see Fig. 9.1). The pipeline is made up of sections connected with each other by valves. The pipe sections can be either pressurized by the tank or unpressurized. Opening a valve causes the section on its right side to be pressurized if the section to its left is pressurized. Moreover, for safety reasons, a valve can be opened only if the next valve in the line is closed. Closing a valve causes the pipe section on its right side to be unpressurized.

We associate this domain with a planning problem of starting a flame in the burner. We start by describing an $\mathcal{AL}$ representation of this domain. The signature contains names for sections of the pipeline, $s_1, s_2, \dots$, and for valves $v_1, v_2, \dots$. The pipeline is described by static relations $connected\_to\_tank(S)$, $connected\_to\_burner(S)$, and $connected(S_1, V, S_2)$. (The latter holds if sections $S_1$ and $S_2$ are connected by valve $V$ and the flow of gas is directed from $S_1$ to $S_2$.) Relation *connected* defines the pipeline as the directed graph with source $s_0$ and sink $s_n$ where $s_0$ is connected to the tank and $s_n$ is connected to the burner. We also need inertial fluents – $opened(V)$ and $burner\_on$ – and defined fluent $pressurized(S)$. The actions are $open(V)$, $close(V)$, and $ignite$ – open and close the corresponding valves and ignite the burner. The state of the domain and its actions are characterized by the following system description:

$$
\begin{array}{l}
pressurized(S) \textbf{ if } connected\_to\_tank(S).\\
pressurized(S2) \textbf{ if } connected(S1, V, S2),\\
\qquad\qquad\qquad\qquad opened(V),\\
\qquad\qquad\qquad\qquad pressurized(S1).\\
\neg burner\_on \textbf{ if } connected\_to\_burner(S),\\
\qquad\qquad\qquad \neg pressurized(S).\\
open(V) \textbf{ causes } opened(V).\\
\textbf{impossible } open(V) \textbf{ if } opened(V).\\
\textbf{impossible } open(V1) \textbf{ if } connected(S1, V1, S2),\\
\qquad\qquad\qquad\qquad\qquad connected(S2, V2, S3),\\
\qquad\qquad\qquad\qquad\qquad opened(V2).\\
close(V) \textbf{ causes } \neg opened(V).\\
\textbf{impossible } close(V) \textbf{ if } \neg opened(V).\\
ignite \textbf{ causes } burner\_on.\\
\textbf{impossible } ignite \textbf{ if } connected\_to\_burner(S),\\
\qquad\qquad\qquad\qquad \neg pressurized(S).
\end{array}
$$

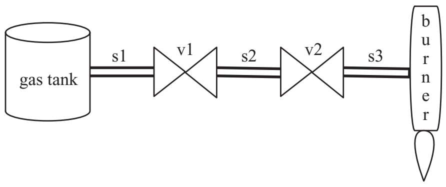

Figure 9.1. Pipeline Configuration

We use this description in conjunction with the specification of a pipeline that includes the pipeline configuration satisfying the above conditions and the status of the pipeline valves. We also assume that initially the burner is off.

Suppose the pipeline looks like the one modeled in Figure 9.1. Then an example initial situation could be

$$\{\neg burner\_on, \neg opened(v1), opened(v2)\}.$$

A goal can be defined as

$$burner\_on.$$

This completes the $\mathcal{AL}$ system description. The translation into ASP is straightforward. Please see Appendix D.1 for the ASP encoding; we call the program `ignite.lp`. The result of invoking `clingo` with the program is

```
occurs(close(v2),0)
occurs(open(v1),1)
occurs(open(v2),2)
occurs(ignite,3)
```

### *9.2.3 Missionaries and Cannibals*

Another classical planning problem is stated as follows:

Three missionaries and three cannibals come to a river and find a boat that holds at most two people. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. How can they all cross?

Here is one possible implementation. Our objects are missionaries, cannibals, a boat, and two locations corresponding to the two banks. We use variables $N$, $N1$, $N2$, $NC$, $NM$, $NCSource$, and $NMSource$ to stand for the number of cannibals or missionaries; they are integers in range 0 to 3.

Similarly, variable names starting with $NB$ stand for the number of boats and can have values of either 0 or 1. We choose to represent the number of missionaries, cannibals and boats at a location with inertial fluents as follows:

$$m(Loc, N)$$

$$c(Loc, N)$$

$$b(Loc, NB)$$

where $Loc$ is location $bank1$ or $bank2$. Letters $m$, $c$, and $b$ stand for missionaries, cannibals, and the boat, respectively. For example, $m(bank1, 3)$ means that there are three missionaries on $bank1$; $b(bank1, 0)$ states that there is no boat at $bank1$. Another important inertial fluent is

$$casualties$$

which is true if the cannibals outnumber the missionaries on the same bank.

Possible actions in this story are ones having to do with the movements of missionaries and cannibals. We use

$$move(NC, NM, Dest)$$

to represent moving, by boat, $NC$ cannibals and $NM$ missionaries to destination $Dest$. For example, $move(0, 1, bank2)$ means to move one missionary to $bank2$.

To define some of our laws, we need to know the source of the movement, not just the destination. We know that the source is always the opposite bank, so we define a static relation:

$$opposite(bank1, bank2)$$

$$opposite(bank2, bank1).$$

The $\mathcal{AL}$ system description consists of the following laws:

- Moving objects increases the number of objects at the destination by the amount moved:

  $move(NC, NM, Dest)$ **causes** $m(Dest, N + NM)$ **if** $m(Dest, N)$
  $move(NC, NM, Dest)$ **causes** $c(Dest, N + NC)$ **if** $c(Dest, N)$
  $move(NC, NM, Dest)$ **causes** $b(Dest, 1)$

- The number of missionaries/cannibals at the opposite bank is 3 − number_on_this_bank. The number of boats at the opposite bank is

1 – number_of_boats_on_this_bank:

$$\begin{aligned}
&m(Source, 3 - N) \textbf{ if } m(Dest, N),\\
&\qquad\qquad opposite(Source, Dest)\\
&c(Source, 3 - N) \textbf{ if } c(Dest, N),\\
&\qquad\qquad opposite(Source, Dest)\\
&b(Source, 1 - NB) \textbf{ if } b(Dest, NB),\\
&\qquad\qquad opposite(Source, Dest)
\end{aligned}$$

- There cannot be different numbers of the same type of person at the same location:

$$\begin{aligned}
&\neg m(Loc, N1) \textbf{ if } m(Loc, N2), N1 \neq N2\\
&\neg c(Loc, N1) \textbf{ if } c(Loc, N2), N1 \neq N2
\end{aligned}$$

- A boat cannot be in and not in a location:

$$\neg b(Loc, NB1) \textbf{ if } b(Loc, NB2), NB1 \neq NB2$$

- A boat cannot be in two places at once:

$$\neg b(Loc1, N) \textbf{ if } b(Loc2, N), Loc1 \neq Loc2$$

- There will be casualties if cannibals outnumber missionaries:

$$\begin{aligned}
casualties \textbf{ if } & m(Loc, NM),\\
& c(Loc, NC),\\
& NM > 0, NM < NC
\end{aligned}$$

- It is impossible to move more than two people at the same time; it is also impossible to move less than one person:

$$\begin{aligned}
&\textbf{impossible } move(NC, NM, Dest) \textbf{ if } (NC + NM) > 2\\
&\textbf{impossible } move(NC, NM, Dest) \textbf{ if } (NM + NC) < 1
\end{aligned}$$

- It is impossible to move objects without a boat at the source:

$$\begin{aligned}
\textbf{impossible } move(NC, NM, Dest) \textbf{ if } & opposite(Source, Dest),\\
& b(Source, 0)
\end{aligned}$$

- It is impossible to move N objects from a source if there are not at least N objects at the source in the first place:

$$\begin{aligned}
\textbf{impossible } move(NC, NM, Dest) \textbf{ if } & opposite(Source, Dest),\\
& m(Source, NMSource),\\
& NMSource < NM\\
\textbf{impossible } move(NC, NM, Dest) \textbf{ if } & opposite(Source, Dest),\\
& c(Source, NCSource),\\
& NCSource < NC
\end{aligned}$$

The ASP encoding is called `crossing.lp` and is in Appendix D.2. Invoking it with

```
clingo 0 crossing.lp
```

gives four answer sets, one of which we show next:

```
occurs(move(1,1,bank2),0)
occurs(move(0,1,bank1),1)
occurs(move(2,0,bank2),2)
occurs(move(1,0,bank1),3)
occurs(move(0,2,bank2),4)
occurs(move(1,1,bank1),5)
occurs(move(0,2,bank2),6)
occurs(move(1,0,bank1),7)
occurs(move(2,0,bank2),8)
occurs(move(0,1,bank1),9)
occurs(move(1,1,bank2),10)
```

## 9.3 Heuristics

The efficiency of ASP planners can be substantially improved by expanding a planning module by domain-dependent heuristics represented by ASP rules. As an example consider our blocks-world planning problem. Note that plans produced by the planning module in `bwplan.lp` may contain actions of the form $put(B, L)$ even when $B$ is already located at $L$. Such an action can be executed by the robot's arm by lifting $B$ up and putting it back on $L$ or by simply doing nothing. In any case the action is completely unnecessary and can be eliminated from consideration by the planner. This can be done by the following heuristic rule:

```
:- holds(on(B,L),I),
   occurs(put(B,L),I).
```

The additional information guarantees that the program will not generate plans containing this type of useless action.

The planning module can be expanded by another useful heuristic that tells the planner to only consider moving those blocks that are out of place. Notice that this heuristic is defined naturally in terms of subgoals because our towers are defined in terms of individual block placement. However, in our current encoding, this information is hidden in the rule defining relation $goal(I)$. To make it explicit we introduce a new relation

$$subgoal(fluent, boolean)$$

and expand our program by the following:

```
subgoal(on(b4,t),true).
subgoal(on(b6,t),true).
subgoal(on(b1,t),true).
subgoal(on(b3,b4),true).
subgoal(on(b7,b3),true).
subgoal(on(b2,b6),true).
subgoal(on(b0,b1),true).
subgoal(on(b5,b0),true).
```

*This idea can be generalized for all heuristics that are based on knowledge of subgoal interaction.*

Now the heuristic can be defined by the following rules:

```
in_place(B,I) :- subgoal(on(B,B1),true),
                 holds(on(B,B1),I),
                 in_place(B1,I).
in_place(t,I) :- step(I).

:- in_place(B,I),
   occurs(put(B,L),I).
```

Again the heuristic allows the elimination of some "nonoptimal" plans. More importantly, it may have a substantial positive effect on the efficiency of the planning program. This is especially true if achieving the agent's goal does not require disassembling some of the towers located on the table. To see the effect of our two heuristics let us look at the following example.

Consider a blocks world with 17 blocks, $b0, \ldots, b16$, an initial block configuration represented by Figure 9.2, and a goal represented by Figure 9.3. The initial configuration is represented by the following collection of facts:

$$holds(on(b0,t),0).\ \ holds(on(b3,b0),0).\ \ldots$$

The goal configuration is given by this rule:

$$goal(I) \leftarrow holds(on(b4,t),I), holds(on(b1,b4),I), \ldots$$

and a collection of subgoals:

$$subgoal(on(b4,t),true).\ \ subgoal(on(b1,b4),true).\ \ldots$$

It may be instructive to run this input with and without the two heuristics described earlier for $n = 7$ (no solution), $n = 8$ (shortest solutions), and

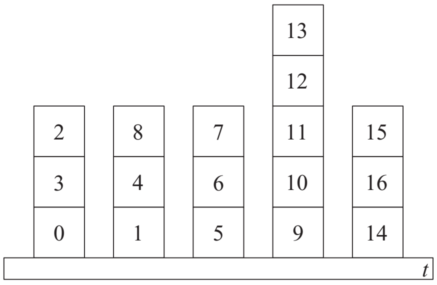

Figure 9.2. Initial Configuration

$n = 9$. Of course the result will depend on the solver you use, but most likely, you will be able to see that addition of the heuristics improves performance of the program. On our computer we have an approximately fivefold increase in efficiency of the planner tested with `clingo`. There is, however, only a slight improvement in the quality of plans. For $n = 8$ both planners (with and without heuristics) find five best plans of length $8$. If $n = 9$ then the planner without heuristics finds $2{,}041$ plans whereas the one with the heuristics finds $2{,}014$ plans. Only $27$ nonoptimal plans are eliminated.

Additional heuristics may help further improve efficiency and eliminate a much larger number of nonoptimal plans. Consider, for instance, another domain-dependent heuristic that gives priority to actions that increase the number of blocks placed in the right position. This can be written as follows:

```
good_move(B,L,I) :- subgoal(on(B,L),true),
                    in_place(L,I),
                    -occupied(L,I),
                    -occupied(B,I).

occupied(B,I) :- block(B),
```

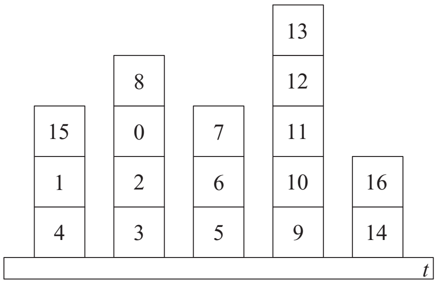

Figure 9.3. Goal Configuration

```
                    holds(on(B1,B),I).
-occupied(t,I) :- step(I).
-occupied(B,I) :- block(B), step(I),
                  not occupied(B,I).
```

The first rule suggests that, if possible, it is good to move block $B$ onto its required position $L$. The last three simply define relation *occupied*.

Note that it may be tempting to express our heuristic by a rule requiring an agent to execute a good move. This can be done by the rule

```
occurs(put(B,L),I) :- good_move(B,L,I).
```

Unfortunately, this rule may lead to inconsistency. Remember that the domain has only one robotic arm and hence can execute only one action at a time. There are, however, situations with more than one good move.

We need to find another representation. The following, slightly more complex rules will do the job:

```
exists_good_move(I) :- good_move(B,L,I).

:- exists_good_move(I),
   occurs(put(B,L),I),
   not good_move(B,L,I).
```

Instead of requiring good moves to be executed, these rules prohibit the execution of “bad” moves.

Let us now try a new planner containing all three heuristics on instances of the blocks-world problem from the previous example. This planner yields not only further improvement in efficiency but also in the number of nonoptimal plans eliminated by the heuristics. Instead of finding $2{,}014$ plans using the first two heuristics for $n = 9$, the planner that uses all three heuristics finds $42$ plans. With the increase of $n$ the advantages of using all three heuristics become even more obvious.

It is difficult to overestimate the importance of heuristic information for our ability to solve difficult search problems. There is, of course, a substantial amount of work related to heuristics. Among other things researchers are trying to discover criteria allowing us to estimate the usefulness of a heuristic and to find algorithms to automatically derive heuristics that are good for a given problem. Much of this work is done for planners based on specialized procedural planning algorithms. As previous examples show, heuristics can also be expressed declaratively, but much more work is needed to fully understand how to evaluate and automatically learn heuristics in the context of declarative planning methods.

## 9.4 Concurrent Planning

If our domain allows the simultaneous execution of several actions, we may want to look for so-called concurrent plans in which more than one action can be executed at each step. To adapt our $plan(\mathcal{P}, n)$ to this situation, all we need to do is change the planner's "generator" rule as follows:

```
1 {occurs(Action,I): action(Action)} m :- step(I),
                                          not goal(I),
                                          I < n.
```

Here `m` is the maximum number of actions that can be performed simultaneously. That's it. Answer sets of this new planning module, used together with the problem description, produce concurrent plans for achieving the problem's goal.

To better understand the behavior of concurrent planners, let us again look at the blocks world but now assume that the domain contains two robotic arms that are able to operate independently. We construct a new program, `twoarmplan.lp`, from `bwplan.lp` by changing the generator rule as described earlier (i.e., we simply set $m = 2$). As discussed in Section 8.5.3, we need to prohibit the program from putting something on a moving block. To achieve this, we add rule

```
-occurs(put(B1,L),I) | -occurs(put(B2,B1),I) :-
                                      step(I),
                                      action(put(B1,L)),
                                      action(put(B2,B1)).
```

to `twoarmsplan.lp`, just as we did in Section 8.5.3.

We set the horizon to 5 and run `clingo` as follows:

```
clingo twoarmsplan.lp
```

The program returns one of the possible plans, say,

```
occurs(put(b7,t),0)
occurs(put(b4,t),0)
occurs(put(b2,t),1)
occurs(put(b6,t),1)
occurs(put(b2,b6),2)
occurs(put(b3,b4),2)
occurs(put(b0,b1),3)
occurs(put(b7,b3),3)
occurs(put(b5,b0),4)
```

The plan contains nine actions executed in five steps. The goal cannot be reached in four steps (i.e., the program with $n = 4$ is inconsistent). Note, however, that this plan is not optimal because there are plans containing a smaller number of actions. For example, the plan

```
occurs(put(b4,t),0)
occurs(put(b2,t),0)
occurs(put(b3,b4),1)
occurs(put(b0,b1),2)
occurs(put(b7,b3),2)
occurs(put(b6,t),3)
occurs(put(b5,b0),4)
occurs(put(b2,b6),4)
```

consist of eight actions. In the next section we show how to find optimal plans for this and similar problems.

## 9.5 (*) Finding Minimal Plans

It is, of course, desirable not to be forced to make multiple calls to a solver or to be able to make very good guesses about the horizon of planning problems in order to find minimal plans. Unfortunately, there is no known simple way to reduce the automatic discovery of minimal plans to computing answer sets of ASP programs. In this section we discuss how finding minimal plans can be done using two extensions of ASP – CR-Prolog introduced in Section 5.5 and ASP with minimality statements implemented on top of many ASP systems such as Smodels and `clingo`.

To use CR-Prolog to find minimal solutions to planning problems, we use the following rules instead of the original simple planning module of ASP:

```
success :- goal(I),
           I <= n.
:- not success.

r1(A,I):occurs(A,I) +-.

something_happened(I) :- occurs(A,I).
:- step(I),
   not something_happened(I),
   something_happened(I+1).
```

```

-occurs(A2,I) :- occurs(A1,I),
                 action(A2),
                 A1 != A2.
```

The first two rules should be familiar. The next rule simply says that during the planning process the agent may consider occurrences of its actions if they are needed to resolve a contradiction (i.e., to achieve "success"). The next two rules guarantee that the agent does not plan to procrastinate (i.e., to remain idle at some time steps and continue actions afterward). The last rule is used to limit our planner to finding plans that allow the execution of at most one action per step.

The **Simple Planning Module of CR-Prolog** then consists of the above rules and the cardinality-based preference relation of CR-Prolog. It allows us to find the shortest one-action-per-step solutions of the planning problem with horizon $n$ without resorting to multiple calls to ASP solvers. To obtain the concurrent version of the planner, we simply remove the planner's last rule. The new planner will find solutions of a planning problem that involve the minimal number of actions.

For example, let's replace the simple planning module of `bwplan.lp` with the CR-Prolog simple planning module to create program `crbwplan.lp`. Recall that running `bwplan.lp` with n=9 gave us plans with useless actions. To run `crbwplan.lp` with horizon 9, use the following command:

```
crmodels -m 0 -c n=9 --min-card --smodels clasp crbwplan.lp
```

- `-m 0` means "find all models"
- `-c n=9` means "set program constant n to 9"
- `--min-card` means "use the cardinality-based preference relation"
- `--smodels clasp` means "use clasp as the solver"

You will see that `crmodels` correctly finds the plans with eight actions, not nine.

Similarly, we can create `crtwoarmsplan.lp` from `twoarmsplan.lp` from the previous section. In this case, we replace the planning module with the CR-Prolog simple planning module without the last rule. Running `crmodels` to find minimal cardinality plans gives us only 2 models, as opposed to the 298 models for `twoarmsplan.lp`, even when the horizon is minimal ($n = 5$). Note also that because of the concurrency, minimal plans can be of different lengths. For example, if we use `crmodels` to run `crtwoarmsplan.lp` with the horizon set to 6, we get 20 models. Each one has only eight actions, but some have five steps and some have six.

We can also compute minimal plans by using a special form of the minimize statement of Lparse and `gringo`. Syntactically, the statement has the form

$$\#minimize\{q(X_1, \dots, X_n) : s_1(X_1) : \cdots : s_n(X_n)\}$$

where $s_1, \dots, s_n$ are sorts of parameters of $q$. The statement, which can be viewed as a directive to an ASP solver, instructs it to compute only those answer sets of the program that contain the smallest number of occurrences of atoms formed by predicate symbol $q$. To use this for planning, we simply add the statement:

```
#minimize{occurs(Action, K) : action(Action) : step(K)}.
```

to our programs. To get all optimizations, we need to call `clingo` with the command option `--opt-all`. To see how this affects the computations, try running `bwplan.lp` with and without the minimize statement as follows:

```
clingo 0 -c n=9 bwplan.lp --opt-all
```

## Summary

In this chapter, we described declarative methodology for solving classical planning problems. Its main steps are as follows:

1. Use an action language (in our case $\mathcal{AL}$) to represent information about an agent and its domain.
2. Automatically translate this representation into a program of Answer Set Prolog.
3. Expand the program by the description of the initial state and the goal.
4. Use answer set solvers to compute the answer sets of the resulting program combined with a small program called the planning module. The maximum number of steps in plans computed by this program is parametrized by non-negative integer $n$.
5. A collection of facts formed by relation *occurs* that belong to such an answer set corresponds to a plan for achieving the goal in at most $n$ steps.

This methodology was demonstrated by several examples.

The chapter is of interest for at least two reasons. First, it explains how to declaratively implement one of the most important reasoning steps in the agent loop – searching for plans to achieve the agent's goal. Second, it can be viewed as another typical example of answer set programming. Intuitively the task is solved by generating sequences of actions and testing if their execution satisfies the goal. This basic generate-and-test procedure is independent of the domain. Of course, real generate-and-test is done by answer set solvers and is much smarter than the "blind" generate-and-test of our theoretical models. We also discussed how additional knowledge given in terms of logic programming rules containing heuristic information can further increase the efficiency of the search.

Our solution has the typical advantages of ASP. It has a reasonably high level of elaboration tolerance. We do not need to alter an existing knowledge base of an agent to add planning capability. Very small changes allow us to go from one-action-at-a-time planning to planning allowing concurrent actions. The resulting programs are provably correct – they are shown to find plans that guarantee success (assuming of course that the world does not change in some unexpected way to interfere with our plans). Existing answer set solvers are sufficiently efficient to provide acceptable solutions for a substantial number of nontrivial planning problems. In many cases they successfully compete with specialized planners. In other cases no procedural solutions are known. This is especially true in domains where planning requires a large amount of knowledge.

There are, of course, a number of remaining problems. First, ASP-based planners may be inefficient if the required plans are very long because grounding programs with a large number of steps can be too costly. There is extensive ongoing work on efficient grounding in the ASP community that may help alleviate this problem (see, for instance, a new incremental ASP solver called `iClingo`). Second, ASP solvers may produce plans with a number of redundant, unnecessary actions. A short discussion in the last section of this chapter described several ways of finding optimal plans that do not contain such actions.

Finally, it is worth mentioning that the chapter only dealt with classical planning. There are many different types of planning undergoing extensive study in AI. For instance, if the initial information of the agent is incomplete or some actions in the domain are nondeterministic, the limited approach given in this chapter is not enough. In this situation the transition diagram of our domain may have multiple trajectories that start at possible initial states and are labeled by the same actions. Some of them will lead to the goal, whereas others will not. Our method allows us to find paths leading to the goal, but cannot guarantee that the execution of the corresponding actions will always do so. A sequence of actions such that *all paths* in the diagram that start in possible initial states and are labeled by actions of the sequence lead to states satisfying the goal is called a *conformant* plan. Finding conformant plans is computationally more difficult than simple planning but, when successfully solved, can be very useful. There are also planning and scheduling problems, the problem of finding plans that succeed with a high degree of probability, and so on. There are substantial advances in these areas as well. Both procedural and declarative methods for solving such problems (including those based on ASP) have been developed, but much more remains to be done.

Planning is a fascinating and important part of human reasoning, and of course, it is not surprising that it is not yet fully understood and automated. We hope, however, that this chapter gives some insight into a number of problems related to planning and outlines a feasible solution to the simplest (but nontrivial) form of this problem.

## References and Further Reading

A good introduction to various planning methods can be found in Geffner and Bonet (2013). The earliest declarative planning program known to the authors is due to Cordell Green (1969). In this work knowledge is represented in situation calculus, and planning is reduced to theorem proving. The origins of declarative planning discussed in this chapter can be found in Kautz and Selman (1992). In this paper Henry A. Kautz and Bart Selman show how the classical planning problem can be reduced to checking satisfiability of a propositional formula. The use of ASP in planning can be traced to Subrahmanian and Zaniolo (1995) and Dimopoulos, Koehler, and Nebel (1997). A good exposition can be found in Lifschitz (2002). For information about various forms of procedural planning see, for instance, Poole and Mackworth (2010) and Ghallab, Nau, and Traverso (2004). Conformant planning is described in Tu et al. (2011) and Tran et al. (2013). More information about the use of heuristics in ASP can be found in Son et al. (2006), Gebser et al. (2013), and Balduccini (2011); Nogueira et al. (2001) contains an example of the practical use of planning with heuristics knowledge. The minimality and other optimization statements were first introduced in ASP by Ilkka Niemela and Patrik Simons (1997). To learn more about solving optimization problems using ASP one can consult Brewka, Niemela, and Truszczynski (2003). A new ASP solver, called `iClingo` (Gebser et al. 2008), grounds and solves problems incrementally; this avoids some of the grounding difficulties related to a large horizon.

## Exercises

1. Replace rules

   ```
   success :- goal(I).
   :- not success.
   ```

   in program `bwplan.lp` by the single statement

   ```
   :- step(I), not goal(I).
   ```

   What happens? Why?
2. Add the planning module to the briefcase program, `bc.lp`, from Section 8.5.1 and set the horizon. Run the program to find a plan to unlock the briefcase from an initial situation in which both clasps are locked.
3. Create a new program, `colorplan.lp`, that extends `bwplan.lp` by incorporating logic programming translations of the laws defined in Examples 9.2.2 and 9.2.3. Test this program with the following colors of blocks in the initial situation: blocks 0, 3, 4, and 5 are red; the rest are white. Use the new goal and find a minimal plan. (Adjust the horizon accordingly.)
4. Modify the basic system description presented in Example 9.2.3 and use it to write a program that finds a minimal plan to have at least one tower consisting of only red blocks. The program must work with an arbitrary initial configuration (although you can adjust the horizon accordingly).
5. Expand the basic blocks-world system description $\mathcal{D}_{bw}$ by information on which blocks are heavy and which are light. Define a tower of blocks to be "uniform" if it consists entirely of one type of block, either all heavy or all light. Use the new system description to write an ASP program to find a minimal plan to create configurations of only uniform towers. The program must work with an arbitrary initial configuration (although you can adjust the horizon accordingly).
6. Give a simple $\mathcal{AL}$ action theory of shooting and use it to write an ASP program that finds a plan for shooting a turkey. Assume that the theory has actions $load$ and $shoot$ and fluents $loaded$ and $alive$.
7. Give an $\mathcal{AL}$ action theory and use it to create an ASP program to solve the following classic puzzle:

A farmer needs to get a chicken, some seed, and a fox safely across a river. He has a boat that will carry him and one "item" across the river. The chicken cannot be left with the seed or with the fox without the farmer's presence. How can the farmer get everything across intact? Note: Don't get cute – there is no bridge, the river is not dry or frozen, there is no cage he can build, etc. The farmer must use the boat to ferry one thing/creature at a time.

8. Create a new system description by extending that given in Example 9.2.3 by adding a new action, $paint(B, C)$, which changes the color of block $B$ to color $C$. Painting actions can be executed concurrently, but it is impossible to paint a moving block. Use the new theory to create a program with block colors in the initial situation as follows: Blocks 0, 3, 4, and 5 are red; the rest are white. Use the "towers must have a red block on top" goal. What is the shortest plan that the program comes up with if action "paint" is allowed? What is the shortest plan if action "paint" does not exist?
9. Convert the program from Exercise 7 in Chapter 8 to a planner that can find a plan for Jonathan to play the Wii.
10. Consider the farmers' market story from Exercise 6 in Chapter 8. Add the knowledge that customers will not buy a product from a vendor if they already have that product. Let us assume that Evaline is the first customer of the day and she knows what every vendor sells. For each of the following goals, write a program to find a plan to achieve it.
    (a) Evaline wants to buy something from every vendor.
    (b) She wants to buy one kind of fruit and two different kinds of vegetables.
    (c) Expand the problem's knowledge base by a new action $make\_coleslaw(P)$, which is executable if person $P$ has carrots and cabbage. Evaline wants to make coleslaw.
    (d) She wants to buy apples, carrots and lettuce from the vendors closest to the entrance who carry these products. Amy has the closest stand followed by Bruce, Carrie, and Don.