CHAPTER 15

Solving Single-Player Games with Logic

15.1 Answer Set Programming

The GDL description of a single-player game can be viewed as the specification of a logical problem. A perfect way to play any such game is by computing a solution upfront. A player can then simply unwind this solution step by step to reach a winning terminal state.

Answer Set Programming (ASP) is one of the fastest existing method for domain-independent solving of problems described in logic. It particularly lends itself to use in general game playing because the input language for ASP is very similar to GDL. Translating one into the other is therefore straightforward and can be easily automated. A variety of ASP solvers are available for free download and can be plugged into your player.

In ASP, problems are described as collections of logic program clauses. A solution to such a problem is any minimal model for the specification. A special type of input formula in ASP are clauses without head. Called *constraints*, they are written as

$$\texttt{:-}\ L_1\ \&\ L_2\ \&\ \ldots\ \&\ L_n$$

Their purpose is to exclude any model as solution in which all of $L_1, L_2, \ldots, L_n$ are simultaneously true.

As a first example, consider the ASP problem specification - also called answer set program - listed below.

```
1  p0
2
3  p2 :- a1 & p1
4  p2 :- b1 & ~p1
5
6  a1 :- ~b1
7  b1 :- ~a1
8
9  :- p2
```

Figure 15.1 - An answer set program.

The clauses together have exactly one (minimal) model. To see why, observe first that `p1` must obviously be true in any model according to the fact in line 1. By the constraint in line 9, `p2` must be false. Hence, by clause 3, `a1` must be false, because otherwise `p2` would be true given that `p1` is true. Knowing that `a1` is false, from clause 7 it follows that `b1` must be true. We thus obtain the dataset M = {`p1,b1`} as the only candidate for a model. It is easy to verify that this candidate also satisfies the program rules we have not considered yet, that is, clause 6 (since `~b1` is false in M) and clause 4 (since `~p1` is false in M). Hence, we have found a minimal model of the program.

In addition to being minimal, an ASP solution must also be *supported*. It means that every atom in a candidate model needs to be the head of a clause whose body is true under the model. Our solution M = {`p1,b1`} satisfies this requirement: `p1` is supported by rule 1 and `b1` is supported by rule 7 (since `~a1` is true in M).

Thet example program in Figure 15.1 can be interpreted as the specification of a simple one-step puzzle. A single state proposition is true at time 1 (fact `p1`). Whether it is still true at time 2 (literal `p2`) depends on which of two actions are chosen, encoded by the literals `a1` and `b1`. Taking the first action does not change the state proposition between times 1 and 2 (rule 3) but taking the second one does (rule 4). Clauses 6-7 together stipulate that exactly one of the two actions is chosen. Finally, the constraint in line 9 can be interpreted as the goal to make the state proposition false: Answers in which `p2` holds are excluded. The model that we have just computed for this program contains `b1`. This provides us with a solution to the problem, namely, that the goal is achieved by taking the second action, `b1`.

15.2 Adding Time to GDL Rules

We can generalize the idea behind the example to single-player games whose solution needs more than just one action. This requires to incorporate a linear temporal dimension into the game rules so that different instances can refer to different time points.

Let's illustrate this with a non-factored variant of the Buttons and Light game from Chapter 11. There are three lights and three buttons. Pushing the first button toggles the first light; pushing the second button interchanges the first and second lights; and pushing the third button interchanges the second and third lights.

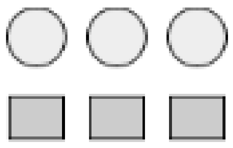

Figure 15.2 - Buttons and Lights

According to the GDL description below, initially the lights are all off. The goal is to turn them on. A step counter ensures that the game terminates after six moves.

```
role(white)

base(p)
base(q)
base(r)
base(step(1))
base(step(N)) :- successor(M,N)
input(white,a)
input(white,b)
input(white,c)

legal(white,a)
legal(white,b)
legal(white,c)

init(step(1))

next(p) :- does(white,a) & ~true(p)
next(p) :- does(white,b) & ~true(q)
next(p) :- ~does(white,a) & ~does(white,b) & true(p)
next(q) :- does(white,b) & true(p)
next(q) :- does(white,c) & true(r)
next(q) :- ~does(white,b) & ~does(white,c) & true(q)
next(r) :- does(white,c) & true(q)
next(r) :- ~does(white,c) & true(r)
next(step(N)) :- true(step(M)) & successor(M,N)

terminal :- true(step(7))
goal(white,100) :- true(p) & true(q) & true(r)
goal(white,  0) :- ~true(p)
goal(white,  0) :- ~true(q)
goal(white,  0) :- ~true(r)

successor(1,2)
successor(2,3)
successor(3,4)
successor(4,5)
successor(5,6)
successor(6,7)
```

Given a set of GDL rules, time steps can be incorporated as follows.

1. Replace `init(F)` by `true(F,0)`.
2. Replace `next(F)` by `true(F,T+1)`.
3. Replace every other atom *p*(…) by *p*(…,`T`), unless
   - *p* is either of the keywords `role`, `base`, or `input`; or
   - *p* is a supporting concept that depends on neither `true` nor `does`.
4. Add the condition `time(T)` to the body of every clause to which variable `T` has been added.

Transforming the game rules above in this fashion results in the program clauses shown below.

```
role(white)

base(p)
base(q)
base(r)
base(step(1))
base(step(N)) :- successor(M,N)
input(white,a)
input(white,b)
input(white,c)

legal(white,a,T) :- time(T)
legal(white,b,T) :- time(T)
legal(white,c,T) :- time(T)

true(step(1),1)

true(p,T+1) :- does(white,a,T) & ~true(p,T) & time(T)
true(p,T+1) :- does(white,b,T) & true(q,T) & time(T)
true(p,T+1) :- ~does(white,a,T) & ~does(white,b,T) & true(p,T) & time(T)
true(q,T+1) :- does(white,b,T) & true(p,T) & time(T)
true(q,T+1) :- does(white,c,T) & true(r,T) & time(T)
true(q,T+1) :- ~does(white,b,T) & ~does(white,c,T) & true(q,T) & time(T)
true(r,T+1) :- does(white,c,T) & true(q,T) & time(T)
true(r,T+1) :- ~does(white,c,T) & true(r,T) & time(T)
true(step(N),T+1) :- true(step(M),T) & successor(M,N) & time(T)

terminal(T) :- true(step(7),T) & time(T)
goal(white,100,T) :- true(p,T) & true(q,T) & true(r,T) & time(T)
goal(white,  0,T) :- ~true(p,T) & time(T)
goal(white,  0,T) :- ~true(q,T) & time(T)
goal(white,  0,T) :- ~true(r,T) & time(T)

successor(1,2)
successor(2,3)
successor(3,4)
successor(4,5)
successor(5,6)
successor(6,7)
```

The domain for time points should be defined as ranging from 0 to some maximum solution length, say 2. Instead of explicitly adding the three facts `time(1)`, `time(2)`, and `time(3)`, any standard ASP solver supports the abridged syntax

```
time(1..3)
```

Given an element `T` from this domain, the solver interprets the expression `T+1` in the way you would expect. Note, therefore, that `true` is the only predicate whose time argument extends one step beyond the given horizon.

The additional time argument allows us to compute the effect of a sequence of moves within the same program. Consider, for example, the move `a` followed by `b`, then `c`. We can compute the sequence of game states by adding the facts shown below to our time-enriched program from above.

```
    does(white,a,1)
    does(white,b,2)
    does(white,c,3)
```

The resulting set of clauses has exactly one minimal and supported model. This includes all of the following atoms.

Time 1	Time 2	Time 3	Time 4
legal(white,a,1)	legal(white,a,2)	legal(white,a,3)	
legal(white,b,1)	legal(white,b,2)	legal(white,b,3)	
legal(white,c,1)	legal(white,c,2)	legal(white,c,3)	
does(white,a,1)	does(white,b,2)	does(white,c,3)	
	true(p,2)		
		true(q,3)	
			true(r,4)
true(step(1),1)	true(step(2),2)	true(step(3),3)	true(step(4),3)
goal(white,0,1)	goal(white,0,2)	goal(white,0,3)	

To understand how such models are computed, observe first that `true(p,1)`, `true(q,1)`, and `true(r,1)` must all be false in any supported model because they do not occur in the head of any clause. The same holds for `does(white,b,1)`, `does(white,c,1)`, and `true(step(2),1), …, true(step(7),1)`. It follows that `true(step(1),1)` and `does(white,a,1)` - along with the other facts in the leftmost column of the table above - are the only atoms with time argument 1 that can occur in a solution. From this we can compute all atoms with time argument 2 using the instantiated rules with the heads `true(p,2)`, `true(q,2)`, `true(r,2)`, and `true(step(1),2), … true(step(7),2)`. Finally, the same rules but now instantiated with `T = 2` can be used to generate the atoms in the solution with time argument 3.

All time-independent facts are of course included in the solution too.

role(white)	base(p)	base(q)	base(r)
base(step(1))	base(step(2))	base(step(3))	base(step(4))
base(step(5))	base(step(6))	base(step(7))	input(white,a)
input(white,b))	input(white,c)	successor(1,2)	successor(2,3)
successor(3,4)	successor(4,5)	successor(5,6)	successor(6,7)

15.3 Solving Single-Player Games with Answer Set Programming

We have seen how a time-expanded GDL description allows you to compute the evolution of the game state for a given sequence of actions. The goal, however, is to find some such sequence that leads to a winning terminal state. In other words, we are looking for a model in which the player takes one legal move at every time step and eventually is awarded 100 points. This is achieved with just a few more clauses.

First, we need to stipulate that the player does one action per time point until the game has terminated. In analogy to the program in Figure 15.1, we could write the following for our example game with the three actions `a`, `b`, and `c`.

```
does(white,a,T) :- time(T) & ~does(white,b,T) & ~does(white,c,T)
does(white,b,T) :- time(T) & ~does(white,a,T) & ~does(white,c,T)
does(white,c,T) :- time(T) & ~does(white,a,T) & ~does(white,b,T)
```

These clauses together requiree that for every instance of `T`, exactly one element from the set {`does(white,a,T), does(white,b,T), does(white,c,T)`} is true in a model. Fortunately, most ASP systems support a more convenient way of specifying exactly this.

```
1 { does(white,a,T), does(white,b,T), does(white,c,T) } 1 :- time(T)
```

The numbers before and after the curly brackets respectively indicate the minimum and maximum number of set elements that must be contained in a solution in order for the head of this clause to be true, for any instance of `T`. This reduces the three clauses from above to a single one. But it still requires us to enumerate all possible moves. An even more compact encoding is obtained by implicitly, rather than explicitly, specifying the elements of a set. There is a simple way to do this by referring to another GDL keyword: Because the set contains all moves `M` that satisfy `input(white,M)`, we can use the following to require our player to make one move at every time step.

```
1 { does(white,M,T) : input(white,M) } 1 :- time(T)
```

In fact, we do not even need to mention explicitly our player's name, `white`, and instead refer to the keyword `role`. This leads to the following rule, which is applicable in any single-player game.

```
1 { does(R,M,T) : input(R,M) } 1 :- role(R) & time(T)
```

A valid solution to a single-player game requires that all chosen moves be legal at the time when they are performed. This is best expressed as a constraint by which any illegal moves are ruled out.

```
:- does(R,M,T) & ~legal(R,M,T)
```

The last and of course most important step is to look for models in which the goal of the game is achieved. To this end, the first constraint below rule out any candidate set in which the game never reaches a terminal state. Aiming high, the second constraint discards any model in which the game reaches a terminal state where the player is not awarded 100 points.

```
:- 0 { terminal(T) : time(T) } 0

:- terminal(T), role(R), ~goal(R,100,T)
```

This completes the answer set program for solving single-player games. Coming back to our example game, Buttons and Lights, suppose the time horizon is set to 7, that is,

```
time(1..7)
```

The program then has a minimal and supported model that includes the following atoms.

does(white,a,1)	does(white,b,2)	does(white,c,3)
does(white,a,4)	does(white,b,5)	does(white,a,6)

Another model exists with moves as shown below.

`does(white,a,1)`	`does(white,b,2)`	`does(white,a,3)`
`does(white,c,4)`	`does(white,b,5)`	`does(white,a,6)`

In fact, both solutions come in three variants, one for each available actions at time 7. This last move is obviously irrelevant since the game has terminated at that time. But it is required according to the rule that the player has to do one action at every time point.

No solution can be found with a time horizon shorter than 7. A longer time horizon, on the other hand, will generate more models with redundant actions before or after the game has terminated. Normally, of course, the required solution length is unknown when you get a new game description. But you can start with setting it to 1 and then increment it until a model is found. This guarantees that the first computed solution is a shortest one.

15.4 Systems for Answer Set Programming

Several ASP systems are available for free download. They are powerful enough to compute solutions to answer set programs for single-player games of medium size. Two such systems are Clasp and DLV.

Exercises

1. **Minimal and supported models**

   Consider the answer set program in Figure 15.1 but without rule 1. Show that this program has two minimal models. Show that only one of them is supported.

2. **Answer set programming**

   Recall the rules that requires every role to make a legal move at every time step.

   ```
   1 { does(R,M,T) : input(R,M) } 1 :- role(R) & time(T)

   :- does(R,M,T), not legal(R,M,T)
   ```

   This requirement may be too strong as it includes one or more time points after the termination of a game. Extend this specification so as to require a (legal) move only in case the game has not terminated at or before time point `T`.

   Hint: Use the GDL keyword `terminal` but be aware that this atom may only be derivable at the time of termination but not at later points in time.

3. **Solving a single-player game with logic**

   Recall the mysterious single-player game dealt with in Exercise 1 of Chapter 13. The goal of this exercise is to solve it with an ASP system.

   a. Extend the given rules by time points as needed.
   b. Add general rules to search for a sequence of legal moves that achieves the goal.
   c. Choose a suitable time horizon and feed the resulting answer set program into an existing ASP solver.

   Hints:

   - Remember that this game terminates when the player is left without a legal move. Hence you need a solution to Exercise 2.

- Familiarize yourself with the exact input syntax for the system of your choice, as this may be slightly different from the one used throughout this chapter. Specifically, the delimiter "`.`" should be added to mark the end of a clause. And most likely, "`&`" and "`~`" need to be substituted by "`,`" and "`not`", respectively.

4. **An automatic solver for single-player games**

   Write a module that automatically translates a single-player GDL into an answer set program and that uses an existing ASP system to try to solve them.