Dynamic Systems

Vladimir Lifschitz, University of Texas

Answer set programming has important applications to the study of *dynamic systems*—systems whose states can be changed by performing actions. It can be used, for instance, to *predict* and to *plan*. In a prediction problem, the task is to determine how the current state of a dynamic system will change after executing a given sequence of actions. In a planning problem, the task is to find a sequence of actions that leads a dynamic system from a given initial state to a goal state.

1 Example: The Blocks World

The *blocks world* is the dynamic system that consists of several blocks stacked one on top of another, forming towers. Each tower is based on the table. If, for example, the blocks are a and b then the blocks world can be in 3 states, shown in Figure 1. If the blocks are a, b, and c, then the blocks world can be in 13 states, shown in Figure 2. The actions available in the blocks world are $move(x, y)$—"move x onto y"—where x is a block and y is either a block other than x or $table$. Such an action can be executed only if there are no blocks on top of x. (We

```
 a      b
 b      a       a b
---    ---     -----
```

S_1 S_2 S_3

Figure 1: States of the blocks world with 2 blocks.

imagine that blocks are moved by a robot's arm, which grasps them from above.) For example, the only actions executable in state S_7 (Figure 2) are

$$move(a, c),\ move(a, table),\ move(c, a);$$

block b can't be moved. Action $move(c, b)$ is not executable in that state because there is another block on top of b. Action $move(c, table)$ is not executable because c is already on the table.

Here are examples of prediction and planning problems in the blocks world with the blocks a, b, c.

```
 a      b      a      c      b      c
 b      a      c      a      c      b
 c      c      b      b      a      a
---    ---    ---    ---    ---    ---
```

S_1 S_2 S_3 S_4 S_5 S_6

```
  a      b      a      c      b      c
  b c    a c    c b    a b    c a    b a    a b c
-----  -----  -----  -----  -----  -----  -------
```

S_7 S_8 S_9 S_{10} S_{11} S_{12} S_{13}

Figure 2: States of the blocks world with 3 blocks.

(i) Assume that the blocks world is initially in state S_7. What is going to be the effect of executing the action $move(a, c)$ and then $move(b, a)$? The answer to this prediction question is that the blocks world will be in state S_2.

(ii) For the same initial state, what about executing the action $move(a, c)$ and then $move(b, c)$? Answer: this sequence of actions is not executable.

(iii) For the same initial state, if we would like b to be on top of c, what sequence of actions would accomplish this? One possible plan is

$$move(a, table);\ move(b, c).$$

To represent a prediction problem, or a planning problem, in the blocks world as a CLINGO program, we'll need to specify, among other things, the set of available blocks, for instance:

```
block(a; b; c).                                                    (1)
```

Exercise D.1.1. (a) In the initial state of the blocks world with n blocks, all blocks are on the table. How many actions can be performed in this state? (b) In the initial state of the blocks world, blocks form n towers, and every tower contains more than one block. How many actions can be performed? (c) Blocks are arranged in n towers, and m out of them contain more than one block. How many actions can be performed?

Exercise D.1.2. Give an example of a planning problem in the blocks world with 3 blocks that can't be solved in fewer than 4 steps.

$$\begin{tikzcd}
S_1 \arrow[r, shift right, "\mathit{move(a,table)}"'] & S_3 \arrow[l, shift right, "\mathit{move(a,b)}"'] \arrow[r, shift left, "\mathit{move(b,a)}"] & S_2 \arrow[l, shift left, "\mathit{move(b,table)}"]
\end{tikzcd}$$

Figure 3: Transition diagram of the blocks world with 2 blocks.

2 Transition Diagrams

A dynamic system can be graphically represented by the directed graph called the *transition diagram*. The vertices of this graph are the states of the system, and its edges are labeled by actions. The tail of an edge is the state in which the action started, and the head is the resulting state.

Figure 3 shows the transition diagram of the blocks world with 2 blocks. The transition diagram of the blocks world with blocks a, b, c has 3 edges starting at S_7:

$$\begin{aligned}
&\text{edge labeled } \mathit{move}(a,c) \text{ leading to } S_9,\\
&\text{edge labeled } \mathit{move}(a,\mathit{table}) \text{ leading to } S_{13},\\
&\text{edge labeled } \mathit{move}(c,a) \text{ leading to } S_4.
\end{aligned} \tag{2}$$

Exercise D.2.1. In the transition diagram of the blocks world with 3 blocks, (a) what is the total number of edges? (b) how many of them lead to S_7? (c) how many of them are labeled $\mathit{move}(a,\mathit{table})$?

A prediction problem can be visualized as the problem of finding the end point of the path in the transition diagram that starts at the given vertex and consists of edges labeled by the given actions. For instance, the answer to question (i) from Section 1 tells us that the path beginning at vertex S_7 with the edges labeled $\mathit{move}(a,c)$ and $\mathit{move}(b,a)$ leads to vertex S_2. The answer to question (ii) tells us that no path starting at S_7 consists of edges labeled $\mathit{move}(a,c)$, $\mathit{move}(b,c)$.

A planning problem can be thought of as the problem of finding a path in the transition diagram that leads from the given initial state to the given set of goal states. For instance, question (iii) from Section 1 calls for finding a path from S_7 to any of the vertices S_1, S_5, S_{11}.

Exercise D.2.2. *Turkey shoot.* Consider the dynamic system consisting of a gun, which can be loaded or unloaded, and a turkey, which can be alive or dead. This system can be in 4 states:

$$(\mathit{loaded}, \mathit{alive}),\ (\mathit{loaded}, \mathit{dead}),\ (\mathit{unloaded}, \mathit{alive}),\ (\mathit{unloaded}, \mathit{dead}).$$

There are 2 actions: *load*, which can be executed whenever the gun is unloaded, and *shoot*, which can be executed whenever the gun is loaded. After shooting, the turkey is dead, and the gun is unloaded. Draw the transition diagram for this dynamic system.

Exercise D.2.3. *3-way bulbs.* Consider the dynamic system consisting of several 3-way bulbs controlled by pull-chain switches. A switch can be in 4 positions: off (0), low (1), medium (2), and high (3). The system can be in 4^n states, represented by lists of numbers of length n, where n is the number of bulbs. The actions are $\mathit{pull}(i)$, "pull the chain controlling bulb i." (a) Draw the transition diagram describing this system for $n = 1$. (b) What is the total number of edges in the transition diagram for an arbitrary n? (c) Describe the edges starting at the vertex (0,0,3).

Exercise D.2.4. *Crossing the river.* You are on a river bank, with a boat and several items that you can take with you to the other bank. The boat, and each of the items can be in one of two locations—on the left bank or on the right bank. These locations determine the state of the system. The items are bulky, so that the boat can't carry more than one at a time. The available actions are $\mathit{cross}(x)$, where x is one of the items (cross the river taking x with you) or the symbol *empty*. (a) Assume that in state S the boat and all n items are on the left bank. What is the length of the shortest path in the transition diagram that leads from S to the state in which the boat and all items are on the right bank? (b) How many paths of length 2 in the transition diagram start at vertex S?

3 Time

To describe sequentially executed actions by a set of atoms, we need to represent time. Time instants will be denoted by integers between 0 and a positive integer h ("horizon"), and the execution of an action will be assumed to take one unit of time. For example, we will think of scenario (i) from Section 1 as executing the action $\mathit{move}(a, c)$ between times 0 and 1, and executing $\mathit{move}(b, a)$ between times 1 and 2.

In an ASP program, this scenario can be represented by the directive

```
#const h=2.                                    (3)
```

and by the facts

```
occurs(move(a,c),0).                           (4)
```

and

```
occurs(move(b,a),1).                           (5)
```

The predicate *occurs* serves for specifying the time when the execution of an action begins. Note that in each of the atoms (4), (5), the first argument is formed from constants using the binary operation symbol `move`, which is not an arithmetic operation. We didn't see terms like these in *Programming with CLINGO*, but they are allowed in programs, and we will have many occasions to use them.

States of a dynamic system can be described in terms of parameters called *fluents*. In the case of the blocks world, locations of blocks can be used as fluents. For instance, state S_7 can be characterized by saying that the location of a is (the top of) b, the location of b is the table, and the location of c is the table also; symbolically,

$$loc(a, b),\ \ loc(b, table),\ \ loc(c, table).$$

The use of fluents to describe the state of a dynamic system is similar to the use of coordinates to describe the position of an object in space.

To specify the value of a fluent at a given time instant, we will use the predicate *holds*. For instance, the assumption that at time 0 the blocks world with 3 blocks is in state S_7 can be expressed by the facts

```
holds(loc(a,b),0).
holds(loc(b,table),0).                    (6)
holds(loc(c,table),0).
```

Exercise D.3.1. States of the turkey shoot domain (Exercise D.2.2) can be described by two fluents, *state_of_gun* with the values *loaded* and *unloaded*, and *state_of_turkey*, with the values *alive* and *dead*. For instance, the condition "the gun is loaded" can be represented by the term *state_of_gun*(*loaded*). The turkey is initially alive, and the gun is unloaded. Then the actions *load* and *shoot* are executed. Describe this scenario by a directive and a set of atoms.

Exercise D.3.2. States of the 3-way bulb domain (Exercise D.2.3) with can be described by n fluents with the values $0, \dots, 3$. For instance, the condition "bulb I is off" can be represented by the term *brightness*(*bulb*(I), 0). The system consisting of 3 bulbs is initially in the state (0,0,3), and then the action *pull*(3) is executed. Describe this scenario by a directive and a set of atoms.

Exercise D.3.3. States of the crossing the river domain (Exercise D.2.4) can be described by the locations of the boat and of all the items. The possible values of these fluents are *left* and *right*. There are two items, a and b. Initially they are on the left bank, and the boat is on the right bank. You cross the river to bring the items to the right bank, first a and then b. Describe this scenario by a directive and a set of atoms.

The requirement that at each time instant, each block must have a unique location can be expressed by the constraint

```
:- #count{L : holds(loc(B,L),T)} != 1, block(B), T=0..h.    (7)
```

Constraints requiring the existence and uniqueness of value of every fluent at every time instant, as in this example, will be included in all our encodings of dynamic domains.

Exercise D.3.4. State the existence and uniqueness of value constraints

(a) for the turkey shoot domain,

(b) for the 3-way bulb domain, using the placeholder `n` for the number of bulbs,

(c) for the crossing the river domain, using the predicate symbol `item/1` to represent the set of items.

4 Effects of Actions

The effect of moving a block to a new location can be expressed by the rule

```
holds(loc(B,L),T+1) :- occurs(move(B,L),T).            (8)
```

(the location of B at time $T+1$ is L if B was moved to L between times T and $T+1$).

The effects of actions in the turkey shoot domain can be expressed by three rules:

```
holds(state_of_gun(loaded),T+1) :- occurs(load,T).
holds(state_of_gun(unloaded),T+1) :- occurs(shoot,T).
holds(state_of_turkey(dead),T+1) :- occurs(shoot,T).
```

In these examples, the values of fluents at time $T+1$ are completely determined by the action performed between the times T and $T+1$; they do not depend on the state in which that action was executed. Crossing the river is different: the location of the boat after crossing depends on where the boat was previously. The effects of actions in that domain can be described using the auxiliary rule

```
opposite(left,right; right,left).
```

as follows:

```
holds(loc(boat,L1),T+1) :- occurs(cross(X),T),
                           holds(loc(boat,L),T),
                           opposite(L,L1).
holds(loc(I,L1),T+1) :- occurs(cross(I),T), item(I),
                        holds(loc(I,L),T),
                        opposite(L,L1).
```

Exercise D.4.1. Describe the effects of actions in the 3-state bulb domain by one rule.

Program (1), (3)–(8) is close to being an adequate description of scenario (i) from Section 1, but it is not completely satisfactory. In fact, that program has no stable models! To understand why, consider the program obtained from it by removing constraint (7). That smaller program has one stable model, consisting of the atoms

```
block(a)  block(b)  block(c)
holds(loc(a,b),0)  holds(loc(b,table),0)  holds(loc(c,table),0)
occurs(move(a,c),0)
holds(loc(a,c),1)
occurs(move(b,a),1)
holds(loc(b,a),2)
```

In this model, blocks b and c are not assigned a location at time 1, and blocks a and c are not assigned a location at time 2. This is why the program becomes inconsistent after adding a constraint that requires a unique location for each block.

A satisfactory encoding of scenario (i) would imply that at time 1 both b and c are on the table, because that's where they were at time 0, and they were not moved anywhere between times 0 and 1. It would also imply that at time 2 block a is on top of c, and c is on the table, because that's where they were at time 1, and they were not moved anywhere between times 1 and 2. Program (1), (3)–(8) specifies the location of a block after moving it, but it does not say that a block "inherits" its location from the previous time instant if it is not moved. This is why program (1), (3)–(8) is unsatisfactory.

This difficulty, known as *the frame problem*, is quite general. If the turkey is alive then it will remain alive after loading the gun; an encoding of the turkey shoot domain will not be adequate unless it allows us to draw this conclusion. If a bulb is off then it will be off after pulling the chain controlling another bulb. If an item is on the left bank then it will be still there after crossing the river with any other item in the boat. Generally, a fluent can be presumed to have the same value that it had before if there is no evidence to the contrary. This principle plays an important role in the theory of dynamic systems, and it is called *the commonsense law of inertia*.

In the blocks world, inertia can be expressed by the rule

```
holds(loc(B,L),T+1) :- holds(loc(B,L),T),
                       not not holds(loc(B,L),T+1),        (9)
                       T=0..h-1.
```

(the location of a block at time $T + 1$ is L if it was L at time T and there is no evidence that it is not L at time $T+1$). Note the use of two negations, next to each other, in the body. The first of them translates the English expression "there is no evidence" into the language of logic programming. It is followed by the negation of the head of the rule; this is the translation of the English expression "to the contrary."

Program (1), (3)–(9) has one stable model—the one that we expected:

```
block(a)  block(b)  block(c)
holds(loc(a,b),0)  holds(loc(b,table),0)  holds(loc(c,table),0)
occurs(move(a,c),0)
holds(loc(a,c),1)  holds(loc(b,table),1)  holds(loc(c,table),1)
occurs(move(b,a),1)
holds(loc(a,c),2)  holds(loc(b,a),2)  holds(loc(c,table),2)
```

Exercise D.4.2. Express inertia by rules (a) for the turkey shoot domain, (b) for the 3-way bulb domain, (c) for the crossing the river domain.

If we replace rule (9) by the choice rule

```
{holds(loc(B,L),T+1)} :- holds(loc(B,L),T), T=0..h-1.     (10)
```

then the stable models of the program will not change. This is not surprising in view of the fact that the propositional rules $p \leftarrow \neg\neg p$ and $p \vee \neg p$ are strongly equivalent to each other (*Stable Models*, Exercise S.11.5); these two formulas are the propositional images of the rules

```
p :- not not p.
```

and

```
{p}.
```

respectively (*Stable Models*, Section 12).

Exercise D.4.3. Rewrite the inertia rule from Exercise D.4.2(b) as a choice rule.

5 Nonexecutable Actions

As discussed above, scenario (i) from Section 1 can be represented by program (1), (3)–(9). But an attempt to turn that program into an encoding of scenario (ii) by replacing rule (5) with

```
occurs(move(b,c),1).     (11)
```

will be unsuccessful. The modified program has a unique stable model that includes the atoms `holds(loc(a,c),2)` and `holds(loc(b,c),2)`, among others. In other words, instead of predicting that moving a and then b onto the same block c is impossible, the program predicts a state in which a and b are both on top of c. There is no such state, because blocks are assumed to be stacked one on top of another forming towers.

The assumption that two blocks can't be located on top of the same block can be expressed by the constraint

```
B1=B2 :- holds(loc(B1,B),T), holds(loc(B2,B),T), block(B).    (12)
```

If we add this constraint to program (1), (3)–(9), and after that replace (5) by (11), then we'll get an adequate representation of scenario (ii): the resulting program has no stable models.

Besides the requirement that blocks be stacked one on top of another, there are two other reasons why moving a block may be impossible. First, a block can be moved only if there are no blocks on top of it (Section 1):

```
:- occurs(move(B,L),T), holds(loc(B1,B),T).    (13)
```

Second, a block can't be moved to its current location:

```
:- occurs(move(B,L),T), holds(loc(B,L),T).    (14)
```

Constraints (12)–(14) describe all possible reasons why the action of moving a block onto another block or onto the table can be nonexecutable. Constraints (13) and (14) are stated in terms of *preconditions*—conditions on a state that must be satisfied before executing the action. According to the former, the block that we want to move should be clear. According to the latter, the location to which we want to move it should be different from its current location. Constraint (12), on the other hand, restricts the executability of actions implicitly. It is stated in terms of fluents, and doesn't even mention actions. But in combination with rule (8), which describes the effect of moving a block, constraint (12) entails that a block can't be moved to a place that is currently occupied.

The assumption that a block can be placed on top of another block only if the latter is clear can be thought of as a precondition, and a constraint expressing that precondition can be used to encode scenario (ii) instead of constraint (12).

Exercise D.5.1. Write a constraint expressing that precondition.

In Section 8 we will say more about the relationship between this precondition and constraint (12).

Exercise D.5.2. In the turkey shoot domain, the action *shoot* can be executed only when the gun is loaded, and the action *load* only when it is not. Express these preconditions by constraints.

Exercise D.5.3. In the crossing the river domain, an item can be taken to the other bank only when it is on the same bank as the boat. Express this precondition by a constraint.

6 Prediction

Listings 1 and 2 reproduce the prediction program presented in Sections 4 and 5, slightly modified. The first listing comprises the general rules, which are applicable to any blocks world prediction problem; the second is the input describing scenario (i) from Section 1. The modification involves introducing the auxiliary predicates `init/1` and `final/1`. The former allows us to describe the initial state a little more concisely—compare facts (6) with Line 3 of Listing 2. The latter is used in the `show` directive at the end of Listing 1.

Exercise D.6.1. (a) Write a prediction program for the turkey shoot domain. (b) Initially, the gun is unloaded and the turkey is alive. Then we load the gun, shoot the turkey, and load the gun again. Describe this scenario as input for your prediction program. (c) Run CLINGO on your program with that input.

Exercise D.6.2. (a) Write a prediction program for the 3-way bulb domain. (b) There are 10 bulbs, and all switches are initially in the high position. We pull all chains controlling the bulbs, one after the other. Describe this scenario as input for your prediction program. (c) Run CLINGO on your program with that input.

Exercise D.6.3. (a) Write a prediction program for the crossing the river domain. (b) There are two items, a and b. Initially a and the boat are on the left bank, and b is on the right bank. We take a to the opposite bank. Describe this scenario as input for your prediction program. (c) Run CLINGO on your program with that input.

7 Planning

The most straightforward application of answer set programming to planning involves looking for a plan of a given length. Consider, for instance, the problem of constructing a plan of length 2 for example (iii) from Section 1. This problem can be encoded by modifying the prediction program shown in Listings 1 and 2 as follows.

First, we replace Line 4 in Listing 2, which describes a specific sequence of actions, by rules allowing an aribitrary sequence of actions of length h:

```
action(move(B1,B2)) :-  block(B1), block(B2), B1!=B2.
action(move(B,table)) :- block(B).
1 {occurs(A,T) : action(A) } 1 :- T=0..h-1
```

These rules allow two kinds of actions in a plan: moving a block onto another block and moving a block onto the table. After this modification, the program will have many stable models: one per each sequence of 2 actions that can be executed starting in state S_7.

Listing 1: Prediction in the blocks world

```
% Predict the effect of executing a sequence of actions
% in the blocks world.

% input: set block/1 of blocks; length h of the sequence
%        of actions; set init/1 of atoms representing
%        the initial state; set occurs/2 of pairs (a,t)
%        such that action a is executed between times t
%        and t+1.

% effects of actions
holds(loc(B,L),T+1) :- occurs(move(B,L),T).

% two blocks can't be located on top of the same block
B1=B2 :- holds(loc(B1,B),T), holds(loc(B2,B),T),
         block(B).

% a block can't be moved unless it is clear
:- occurs(move(B,L),T), holds(loc(B1,B),T).

% a block can't be moved to its current location
:- occurs(move(B,L),T), holds(loc(B,L),T).

% existence and uniqueness of value
:- #count{L : holds(loc(B,L),T)} != 1, block(B), T=0..h.

% inertia
{holds(loc(B,L),T+1)} :- holds(loc(B,L),T), T=0..h-1.

% relationship between holds/2, init/1, and final/1
holds(A,0) :- init(A).
final(A) :- holds(A,h).

#show final/1.
```

Listing 2: Input for the blocks world prediction program

```
block(a; b; c).
#const h=2.
init(loc(a,b); loc(b,table); loc(c,table)).
occurs(move(a,c),0; move(b,a),1).
```

Exercise D.7.1. How many stable models do you think the modified program has?

Now it remains to add a constraint eliminating "bad" choices of actions—those for which the goal of having b on top of c at the end is not satisfied:

```
:- not holds(loc(b,c),h).
```

Stable models of the new program represent all plans of length 2 that solve the problem in question.

Exercise D.7.2. How many stable models do you think are eliminated by adding this constraint?

A general planning program for the blocks world and an input file for it that represents problem (iii) from Section 1 are shown in Listings 3 and 4. The auxiliary predicate `goal/1` is used to encode the goal of the planning problem.

Exercise D.7.3. In the blocks world with 9 blocks, the initial state is

```
1
2 5
3 6 8
4 7 9
```
(15)

and the goal is to have 9 on top of 6, 6 on top of 3, and 3 on the table. We would like to achieve this goal in 7 steps. Encode this problem as an input file for the blocks world planning program in Listing 3.

If the blocks world planning program doesn't have stable models for a given input then we can conclude that there is no plan of required length h. It doesn't follow, generally, that there is no plan of length less than h, so that we can't conclude that the given value of h is too small. But it is easy to find out whether it is indeed necessary to increase h. This can be accomplished by introducing a new action, which has no effect on the values of fluents.

```
action(wait).
```
(16)

Allowing this trivial action to appear in the plan is essentially equivalent to allowing h to be replaced by a smaller number.

Listing 3: Planning in the blocks world

```
% For a planning problem in the blocks world, find
% solutions of a given length.

% input: set block/1 of blocks; set init/1 of atoms
%        representing the initial state; set goal/1 of
%        atoms representing the goal; length h of
%        solutions.

% choice of actions
action(move(B1,B2)) :-  block(B1), block(B2), B1!=B2.
action(move(B,table)) :- block(B).
1 {occurs(A,T) : action(A) } 1 :- T=0..h-1.

% effects of actions
holds(loc(B,L),T+1) :- occurs(move(B,L),T).

% two blocks can't be located on top of the same block
B1=B2 :- holds(loc(B1,B),T), holds(loc(B2,B),T),
         block(B).

% a block can't be moved unless it is clear
:- occurs(move(B,L),T), holds(loc(B1,B),T).

% a block can't be moved to its current location
:- occurs(move(B,L),T), holds(loc(B,L),T).

% existence and uniqueness of value
:- #count{L : holds(loc(B,L),T)} != 1, block(B), T=0..h.

% inertia
{holds(loc(B,L),T+1)} :- holds(loc(B,L),T), T=0..h-1.

% relationship between holds/2, init/1, and goal/1.
holds(A,0) :- init(A).
:- goal(A), not holds(A,h).

#show occurs/2.
```

Listing 4: Input for the blocks world planning program

```
block(a; b; c).
init(loc(a,b); loc(b,table); loc(c,table)).
goal(loc(b,c)).
#const h=2.
```

Furthermore, if we know a value of h such that the planning program with line (16) added to the input has a stable model then we can instruct CLINGO to look for the shortest possible solution to the given problem. To do that, add a directive that calls for using the trivial action as often as possible:

```
#maximize{T : occurs(wait,T)}.
```
(17)

Take, for example, the input shown in the solution to Exercise, D.7.3, replace the value 7 of the placeholder `h` by a larger number—say, 10—and add lines (16) and (17) to it. For the modified input, the last part of the output produced by CLINGO may look like this:

```
Answer: 11
occurs(move(1,table),0) occurs(move(2,1),1) occurs(move(5,table),2)
occurs(move(3,table),3) occurs(move(8,table),4) occurs(move(6,3),5)
occurs(move(9,6),6) occurs(wait,7) occurs(wait,8) occurs(wait,9)
Optimization: -24
OPTIMUM FOUND
```

The fact that 3 out of the 10 `occurs` atoms in the optimal stable model contain the trivial action shows that the length of the shortest solution to the given planning problem is 7. One of these solutions is given by the `occurs` atoms that do not contain `wait`.

Exercise D.7.4. In the output above, the trivial action is always executed at the very end, between times 7 and 10. Can you explain why?

Exercise D.7.5. (a) Write a planning program for the turkey shoot domain. (b) Initially, the gun is loaded and the turkey is alive. We'd like the gun to be loaded while the turkey is dead. Use CLINGO to solve this planning problem.

Exercise D.7.6. (a) Write a planning program for the 3-way bulb domain. (b) There are 3 bulbs, and initially all switches are in the medium position. We'd like all of them to be off. Use CLINGO to solve this planning problem.

In some cases, we want to find a solution to a planning problem that satisfies some additional conditions, beyond those expressed by the definition of `goal/1`. This can be often accomplished by adding constraints to the input of the planning program. As an example, consider the planning problem from Exercise D.7.3 with the additional requirement: there should never be more than 3 towers on the table while the plan is executed. (The table is small, and there is no room on it for more than 3 blocks side by side.) This requirement can be expressed by the constraint

```
  :- #count{B : holds(loc(B,table),T)}>3, T=0..h.
```

Exercise D.7.7. Use CLINGO to find the shortest plan satisfying this condition.

In the fox, goose and bag of beans puzzle (`https://en.wikipedia.org/wiki/Fox,_goose_and_bag_of_beans_puzzle`) a farmer must transport a fox, goose and bag of beans from one side of a river to another using a boat which can only hold one item in addition to the farmer, subject to the constraints that the fox cannot be left alone with the goose, and the goose cannot be left alone with the beans. This puzzle can be viewed as a planning problem in the crossing the river domain with additional conditions: the fox and the goose cannot be on the same bank unless the boat is there as well, and similarly for the goose and the beans.

Exercise D.7.8. (a) Write a planning program for the crossing the river domain. (b) Write an input file for this program that represents the fox, goose and bag of beans puzzle. (c) Use CLINGO to solve the puzzle.

8 Concurrency

A robot with several grippers may be able to move several objects simultaneously. Think, for instance, of a robot with 2 grippers facing blocks world configuration (15). Instead of choosing between the actions, say, *move*(1,*table*) and *move*(5,*table*), it can execute these actions concurrently. Or it can swap blocks 1 and 5 by executing the actions *move*(1,6) and *move*(5,2). These two actions can only be performed concurrently; any one of them is not executable in state (15) by itself.

To talk about the concurrent execution of actions in terms of transition diagrams, we need to modify the description of that concept given in Section 2 and label the edges of the graph by sets of actions, rather than individual actions. In case of the blocks world with 3 blocks (Figure 2) and a robot with 2 grippers, every edge will be labeled by a set consisting of 1 or 2 actions. Let's assume that the robot is unable to move a block onto a block that is being moved at the same time. Under this assumption, instead of the edges (2) starting at S_7, in the modified transition diagram we will see

edge labeled $\{move(a,c)\}$ leading to S_9,
edge labeled $\{move(a,table)\}$ leading to S_{13},
edge labeled $\{move(c,a)\}$ leading to S_4,
edge labeled $\{move(a,table), move(c,b)\}$ leading to S_{12}.

To encode the assumption that blocks are being moved by a robot with m grippers (or by m agents whose actions are synchronous), we replace Line 12 in Listing 3 by the rule

```
1 {occurs(A,T) : action(A) } m :- T=0..h-1.
```

The assumption that the destination of a move is not moving concurrently can be expressed by the constraint

```
:- occurs(move(B1,B2),T), occurs(move(B2,L),T).
```

It's interesting that constraint (12) and the precondition from Exercise D.5.1 are not interchangeable when concurrent actions are allowed. The latter would not allow `B1` to be moved onto `B2` even while the block covering `B2` is being moved away. For instance, it would prohibit swapping blocks 1 and 5 in state (15). This is much too strong. On the other hand, it would allow two blocks to be moved concurrently onto the same block that is currently clear. For instance, it would allow moving blocks 1 and 5 onto block 8 concurrently. In this sense, it is too weak. Constraint (12), which restricts the executability of actions indirectly, doesn't suffer from these problems.

When concurrent actions are allowed, the shortest solution to a planning problem is not necessarily the most economical in the sense of the total number of actions: counting actions is different from counting steps.

The use of rule (16) and directive (17) for generating the shortest solution to a planning problem of this kind requires an additional constraint: the trivial action is not executed concurrently with any other action. This can be written as

```
A=wait :- occurs(wait,T), occurs(A,T).
```

Exercise D.8.1. Use CLINGO to find out how many steps a robot with 2 grippers would need to solve the planning problem from Exercise D.7.3.

Exercise D.8.2. (a) Modify the planning program from Exercise D.7.6(a) to reflect the assumption that the person operating the switches uses both hands and can access any two switches simultaneousy. (b) How many steps do you think are needed to solve the planning problem from Exercise D.7.6(b) under this assumption? Use CLINGO to verify your conjecture.

Answers to Exercises

D.1.1. (a) $n(n-1)$. (b) n^2. (c) $n^2 - n + m$.

D.1.2. Initial state S_1; goal S_3.

D.2.1. (a) 30. (b) 3. (c) 4.

D.2.2.

$$\begin{tikzcd}
(\mathit{loaded}, \mathit{alive}) \arrow[dr, "\mathit{shoot}"] & (\mathit{loaded}, \mathit{dead}) \arrow[d, shift right, "\mathit{shoot}"'] \\
(\mathit{unloaded}, \mathit{alive}) \arrow[u, "\mathit{load}"] & (\mathit{unloaded}, \mathit{dead}) \arrow[u, shift right, "\mathit{load}"']
\end{tikzcd}$$

D.2.3. (a)

$$\begin{tikzcd}
0 \arrow[r, "\mathit{pull}(1)"] & 1 \arrow[d, "\mathit{pull}(1)"] \\
3 \arrow[u, "\mathit{pull}(1)"] & 2 \arrow[l, "\mathit{pull}(1)"]
\end{tikzcd}$$

(b) $n4^n$. (c) Edge labeled *pull*(1) leading to (1,0,3); edge labeled *pull*(2) leading to (0,1,3); edge labeled *pull*(3) leading to (0,0,0).

D.2.4. (a) $2n - 1$. (b) $2n + 1$.

D.3.1.

```
#const h=2.
holds(state_of_gun(unloaded),0).
holds(state_of_turkey(alive),0).
occurs(load,0).
occurs(shoot,1).
```

D.3.2.

```
#const h=1.
holds(brightness(bulb(1),0),0).
holds(brightness(bulb(2),0),0).
holds(brightness(bulb(3),3),0).
occurs(pull(3),0).
```

D.3.3.

```
#const h=4.
holds(loc(a,left),0).
holds(loc(b,left),0).
holds(loc(boat,right),0).
occurs(cross(empty),0).
occurs(cross(a),1).
occurs(cross(empty),2).
occurs(cross(b),3).
```

D.3.4. (a)

```
      :- #count{S : holds(state_of_gun(S),T)} != 1, T=0..h.
      :- #count{S : holds(state_of_turkey(S),T)} != 1, T=0..h.
```

(b)

```
:- #count{B : holds(brightness(bulb(I),B),T)} != 1, I=1..n, T=0..h.
```

(c)

```
      :- #count{L : holds(loc(I,L),T)} != 1, item(I), T=0..h.
      :- #count{L : holds(loc(boat,L),T)} != 1, T=0..h.
```

D.4.1.

```
    holds(brightness(bulb(I),(B+1)\4),T+1) :-
              occurs(pull(I),T), holds(brightness(bulb(I),B),T).
```

D.4.2. (a)

```
 holds(state_of_gun(X),T+1) :- holds(state_of_gun(X),T),
                               not not holds(state_of_gun(X),T+1),
                               T=0..h-1.
 holds(state_of_turkey(X),T+1) :-
                          holds(state_of_turkey(X),T),
                          not not holds(state_of_turkey(X),T+1),
                          T=0..h-1.
```

(b)

```
holds(brightness(bulb(I),B),T+1) :-
                         holds(brightness(bulb(I),B),T),
                         not not holds(brightness(bulb(I),B),T+1),
                         T=0..h-1.
```

(c) Rule (9) will do. (It would be natural to use the variable `X` in that rule in place of `B`.)

D.4.3.

```
 {holds(brightness(bulb(I),B),T+1)} :- holds(brightness(bulb(I),B),T),
                                       T=0..h-1.
```

D.5.1. `:- occurs(move(B1,B2),T), holds(loc(B,B2),T).`

D.5.2.

```
:- occurs(shoot,T), holds(state_of_gun(unloaded),T).
:- occurs(load,T), holds(state_of_gun(loaded),T).
```

D.5.3.

```
L1=L2 :- occurs(cross(I),T), item(I),
         holds(loc(I,L1),T), holds(loc(boat,L2),T).
```

D.6.1. (a)

```
% Predict the effect of executing a sequence of actions in the
% turkey shoot domain.

% input: length h of the sequence of actions; set init/1 of atoms
%        representing the initial state; set occurs/2 of pairs
%        (a,t) such that action a is executed between times t and t+1.

% effects of actions
holds(state_of_gun(loaded),T+1) :- occurs(load,T).
holds(state_of_gun(unloaded),T+1) :- occurs(shoot,T).
holds(state_of_turkey(dead),T+1) :- occurs(shoot,T).

% loading is impossible if the gun is loaded
:- occurs(load,T), holds(state_of_gun(loaded),T).

% shooting is impossible if the gun is unloaded
:- occurs(shoot,T), holds(state_of_gun(unloaded),T).

% existence and uniqueness of value
:- #count{X : holds(state_of_gun(X),T)} != 1, T=0..h.
:- #count{X : holds(state_of_turkey(X),T)} != 1, T=0..h.

% inertia
{holds(state_of_gun(X),T+1)} :- holds(state_of_gun(X),T), T=0..h-1.
{holds(state_of_turkey(X),T+1)} :- holds(state_of_turkey(X),T),
                                   T=0..h-1.

% Relationship between holds/2, init/1, and final/1
holds(A,0) :- init(A).
final(A) :- holds(A,h).

#show final/1.
```

(b)

```
#const h=3.
init(state_of_gun(unloaded); state_of_turkey(alive)).
occurs(load,0; shoot,1; load,2).
```

D.6.2. (a)

```
% Predict the effect of executing a sequence of actions in the
% 3-way bulb domain.

% input: number n of bulbs; length h of the sequence of actions;
%        set init/1 of atoms representing the initial state; set
%        occurs/2 of pairs (a,t) such that action a is executed
%        between times t and t+1.

% effects of actions
holds(brightness(bulb(I),(B+1)\4),T+1) :-
         occurs(pull(I),T), holds(brightness(bulb(I),B),T).

% existence and uniqueness of value
:- #count{B : holds(brightness(bulb(I),B),T)} != 1, I=1..n, T=0..h.

% inertia
{holds(brightness(bulb(I),B),T+1)} :- holds(brightness(bulb(I),B),T),
                                      T=0..h-1.

% relationship between holds/2, init/1, and final/1
holds(A,0) :- init(A).
final(A) :- holds(A,h).

#show final/1.
```

(b)

```
#const n=10.
#const h=10.
init(brightness(bulb(1..10),3)).
occurs(pull(I),I-1) :- I=1..10.
```

D.6.3. (a)

```
% Predict the effect of executing a sequence of actions in the
% crossing the river domain.

% input: set item/1 of items; length h of the sequence of actions;
%        set init/1 of atoms representing the initial state; set
%        occurs/2 of pairs (a,t) such that action a is executed
%        between times t and t+1.

% definition of opposite/2.
opposite(left,right; right,left).

% effects of actions
holds(loc(boat,L1),T+1) :- occurs(cross(X),T),
                           holds(loc(boat,L),T),
                           opposite(L,L1).
holds(loc(I,L1),T+1) :- occurs(cross(I),T), item(I),
                        holds(loc(I,L),T),
                        opposite(L,L1).

% existence and uniqueness of value
:- #count{L : holds(loc(I,L),T)} != 1, item(I), T=0..h.
:- #count{L : holds(loc(boat,L),T)} != 1, T=0..h.

% inertia
{holds(loc(X,L),T+1)} :- holds(loc(X,L),T), T=0..h-1.

% relationship between holds/2, init/1, and final/1
holds(A,0) :- init(A).
final(A) :- holds(A,h).

#show final/1.
```

(b)

```
item(a; b).
#const h=1.
init(loc(a,left); loc(boat,left); loc(b,right)).
occurs(cross(a),0).
```

D.7.3. (a)

```
block(1..9).
init(loc(1,2); loc(2,3); loc(3,4); loc(4,table);
     loc(5,6); loc(6,7); loc(7,table);
     loc(8,9); loc(9,table)).
goal(loc(9,6); loc(6,3); loc(3,table)).
#const h=7.
```

D.7.4. Maximizing the sum of the times T when the *wait* action is executed makes each T as large as possible.

D.7.5. (a)

```
% For a planning problem in the turkey shoot domain, find
% solutions of a given length.

% input: set init/1 of atoms representing the initial state;
%        set goal/1 of atoms representing the goal; length h
%        of solutions.

% choice of actions
action(load; shoot).
1 {occurs(A,T) : action(A) } 1 :- T=0..h-1.

% effects of actions
holds(state_of_gun(loaded),T+1) :- occurs(load,T).
holds(state_of_gun(unloaded),T+1) :- occurs(shoot,T).
holds(state_of_turkey(dead),T+1) :- occurs(shoot,T).

% loading is impossible if the gun is loaded
:- occurs(load,T), holds(state_of_gun(loaded),T).

% shooting is impossible if the gun is unloaded
:- occurs(shoot,T), holds(state_of_gun(unloaded),T).

% existence and uniqueness of value
:- #count{X : holds(state_of_gun(X),T)} != 1, T=0..h.
:- #count{X : holds(state_of_turkey(X),T)} != 1, T=0..h.

% inertia
{holds(state_of_gun(X),T+1)} :- holds(state_of_gun(X),T), T=0..h-1.
{holds(state_of_turkey(X),T+1)} :- holds(state_of_turkey(X),T),
                                   T=0..h-1.
```

```
% relationship between holds/2, init/1, and goal/1.
holds(A,0) :- init(A).
:- goal(A), not holds(A,h).

#show occurs/2.
```

D.7.6. (a)

```
% For a planning problem in the 3-way bulb domain, find solutions
% of a given length.

% input: number n of bulbs; set init/1 of atoms representing
%        the initial state; set goal/1 of atoms representing the
%        goal; length h of solutions.
% choice of actions
action(pull(1..n)).
1 {occurs(A,T) : action(A) } 1 :- T=0..h-1.

% effects of actions
holds(brightness(bulb(I),(B+1)\4),T+1) :-
         occurs(pull(I),T), holds(brightness(bulb(I),B),T).

% existence and uniqueness of value
:- #count{B : holds(brightness(bulb(I),B),T)} != 1, I=1..n, T=0..h.

% inertia
{holds(brightness(bulb(I),B),T+1)} :- holds(brightness(bulb(I),B),T),
                                      T=0..h-1.

% relationship between holds/2, init/1, and goal/1.
holds(A,0) :- init(A).
:- goal(A), not holds(A,h).

#show occurs/2.
```

D.7.8. (a)

```
% For a planning problem in the crossing the river domain, find
% solutions of a given length.

% input: set item/1 of items; set init/1 of atoms representing
%        the initial state; set goal/1 of atoms representing the
%        goal; length h of solutions.
% choice of actions
action(cross(I)) :- item(I).
action(cross(empty)).
1 {occurs(A,T) : action(A) } 1 :- T=0..h-1.

% definition of opposite/2.
  opposite(left,right; right,left).

% effects of actions
  holds(loc(boat,L1),T+1) :- occurs(cross(X),T),
                             holds(loc(boat,L),T),
                             opposite(L,L1).
  holds(loc(I,L1),T+1) :- occurs(cross(I),T), item(I),
                          holds(loc(I,L),T),
                          opposite(L,L1).

% existence and uniqueness of value
:- #count{L : holds(loc(I,L),T)} != 1, item(I), T=0..h.
:- #count{L : holds(loc(boat,L),T)} != 1, T=0..h.

% inertia
{holds(loc(X,L),T+1)} :- holds(loc(X,L),T), T=0..h-1.

% relationship between holds/2, init/1, and goal/1.
holds(A,0) :- init(A).
:- goal(A), not holds(A,h).

#show occurs/2.
```

(b)

```
item(fox; goose; beans).

init(loc(boat,left)).
init(loc(I,left)) :- item(I).

goal(loc(boat,right)).
goal(loc(I,right)) :- item(I).

:- holds(loc(fox,L),T), holds(loc(goose,L),T),
   not holds(loc(boat,L),T).
```

```
:- holds(loc(goose,L),T), holds(loc(beans,L),T),
   not holds(loc(boat,L),T).

#const h=10.

action(wait).
#maximize{T : occurs(wait,T)}.
```

D.8.2. (a) Replace the line

```
1 {occurs(A,T) : action(A) } 1 :- T=0..h-1.
```

by

```
1 {occurs(A,T) : action(A) } 2 :- T=0..h-1.
```