Table of Contents I

Reading

- Read Chapter 9, *Planning Agents*, in the KRR book.

Back to Agents

Now that we have a way of representing knowledge about the world and how actions affect it, we want our agent to use that knowledge to plan its actions.

Recall from the intro that the agent:

1. observes the world, checks that its observations are consistent with its expectations, and updates its knowledge base;
2. selects an appropriate goal G;
3. searches for a plan (a sequence of actions) to achieve G;
4. executes some initial part of the plan, updates the knowledge base, and goes back to step (1).

The Classical Planning Problem

- A **goal** is a set of fluent literals which the agent wants to become true.
- A **plan** for achieving a goal is a sequence of agent actions which takes the system from the current state to one which 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 α 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 α.

A Declarative Approach to Planning

1. Use $\mathcal{AL}$ to represent information about an agent and its domain.
2. Translate to ASP.
3. Add the initial state to the program.
4. Add the goal to the program.
5. Add the **simple planning module** — a small, domain-independent ASP program.
6. Use a solver to compute the answer sets of the resulting program.
7. A plan is a collection of facts formed by relation *occurs* which belong to such an answer set.

Representing the Domain Description

- We already know how to represent information about an agent's domain in $\mathcal{AL}$ and translate it to ASP.
- Representing the initial state is the same as representing σ_0 in the transition diagram. (We use relation $holds(Fluent, 0)$.)
- Check off 1–3.

Representing the Goal

Relation *goal* holds iff all fluent literals from the problem's goal G are satisfied at step I of the system's trajectory:

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

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

Simple Planning Module — Achieving the Goal

```
% There must be a step in the system
% that satisfies the goal.

success :- goal(I).

% Failure is unacceptable;
% if a plan doesn't exist, there is no answer set.

:- not success.
```

Simple Planning Module — Action Generator

```
% An action either occurs at I or it doesn't.
occurs(A,I) | -occurs(A,I) :- not goal(I).

%% Do not allow concurrent actions:
:- 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).

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

Alternative Simple Planning Module with Choice Rule

Currently, Clingo runs much faster than DLV if you use a choice rule:

```
% There must be a step in the system
% that satisfies the goal.
success :- goal(I).

% Failure is unacceptable;
% if a plan doesn't exist, there is no answer set.
:- not success.

% Select exacty one action to occur per step:
1{occurs(A,I): action(A)}1 :- step(I),
                              not goal(I),
                              I < n.
```

That's It!

- Since ASP solvers already exist,
- and we have a proof that answer sets of such a program correspond to plans,
- we now have all the parts (1–7) we need to write a planner.

The Horizon

- Our particular planning module requires a **horizon** — a limit on the length of allowed plans.
- Constant `n`, which represents the limit on the number of steps in our trajectory, is set to the horizon.
- In our code, we do not have to specify that `I` $<$ `n` because it is constrained by SPARC declarations, but it is there.
- If the plan is shorter than the horizon, the planner may generate plans that contain unnecessary actions.
- There is no *known* way of finding minimal plans with straight ASP unless you call the program with n $= 1, 2, \ldots$ until you get an answer set.
- However, there are extensions that we'll talk about later that can be used to find minimal plans.

Example: Blocks World I

1. Use the $\mathcal{AL}$ description from before.
2. Translate to ASP (as before).
3. Add the initial state to the program; e.g.,:

```
holds(on(b0,t),0).
holds(on(b3,b0),0).
holds(on(b2,b3),0).
holds(on(b1,t),0).
holds(on(b4,b1),0).
holds(on(b5,t),0).
holds(on(b6,b5),0).
holds(on(b7,b6),0).
-holds(on(B,L),0) :- not holds(on(B,L),0).
```

Example: Blocks World II

4. Add the goal to the program:

```
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).
```

5. Add the **simple planning module**. (Cut and paste.)
6. Use a solver to compute the answer sets of the resulting program. (Not new.)
7. A plan is a collection of facts formed by relation *occurs* which belong to such an answer set. (Display the occurs statements.)

Advantages

- Problem description is separate from the reasoning part so we can change the initial state, the goal, and the horizon at will.
- We can write domain-specific rules describing actions that can be ignored in the search.
- If a solver is improved, the planner is improved.

Multiple Goal States

Suppose our goal is to have $b3$ on the table, but we don't care what happens to the other blocks. We write:

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

Naturally, multiple states will satisfy this condition, and plans will vary accordingly.

Using Defined Fluents in the Goal I

Suppose we had defined fluent *occupied*(*Block*) defined by

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

We can add the translation of this rule to the blocks-world program:

```
holds(occupied(B),I) :- #block(B),
                        holds(on(B1,B),I).
```

(We don't need to add the CWA for *occupied* because we already have the general CWA for defined fluents.)

Using Defined Fluents in the Goal II

Now we can use this fluent to specify that we want some blocks to be unoccupied:

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

You can see how this could be useful.

Defining Complex Goals

- Suppose we now wanted to describe a blocks-world domain in which we cared about colors.
- We could add a new sort, *color* and a new fluent, $is\_colored(B, C)$.
- Each block only has one color:

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

- New goal: all towers must have a red block on top.

Red Blocks on Top

- We need a way to describe what we want.
- Let's define a new defined fluent, *wrong_config*, which is true if we have towers that don't have a red block on top:

$$\mathit{wrong\_config} \textbf{ if } \neg \mathit{occupied}(B), \neg \mathit{is\_colored}(B, \mathit{red}).$$

- Notice that it is often easier to define what we don't want than what we do.
- Now the goal can be written as:

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

Igniting the Burner

- Here's a completely new 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. 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.

- The goal is to turn on the burner.

Signature

- sort section = s1, s2, s3.
- sort valve = v1, v2.
- statics: connected_to_tank(S), connected_to_burner(S), and connected(S1, V, S2).
- inertial fluents: opened(V) and burner_on.
- defined fluent: pressurized(S).
- actions: open(V), close(V), and ignite.

System Description

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

State, Goal, Plan

- Example initial state:

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

- Example goal:

$$burner\_on.$$

- Translate into SPARC:
  `http://pages.suddenlink.net/ykahl/s_ignite.txt`
- Example plan:

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

Missionaries and Cannibals

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?

What Are Our Objects?

- 3 missionaries
- 3 cannibals
- 1 boat
- 2 banks

Since we are interested in numbers of people and not specific individuals, we will represent them with numbers 0–3.

It is also convenient to represent the boat with numbers 0 or 1.

What Are the Fluents?

- Specifically, we are interested in the number of missionaries/cannibals on a bank.
- We can represent this with inertial fluents:

```
m(#location,#num)          % num missionaries at loc
c(#location,#num)          % num cannibals at loc
b(#location,#num_boats)    % num_boats at loc

% Examples:
%  m(bank1, 3) -- 3 missionaries on bank1.
%  b(bank1, 0) -- no boats on bank 1.
```

- We are also interested in there being no casualties, so we introduce inertial fluent

```
casualties  % True if the cannibals outnumber
            % the missionaries.
```

What Are the Actions?

- Movement changes the location of the people and the boat.
- Specifically, we want to know how many missionaries and cannibals we are moving and where:

  `move(#num_cannibals, #num_missionaries, #loc).`

Here Are the Sorts in SPARC

```
#step = 0..n.
#numM = 0..3.              % number of missionaries
#numC = 0..3.              % number of cannibals
#num_boats = 0..1.         % number of boats
#location = {bank1, bank2}.
#inertial_fluent =
    m(#location,#numM) +          % num missionaries at loc
    c(#location,#numC) +          % num cannibals at loc
    b(#location,#num_boats) +     % num_boats at location
    {casualties}.                 % true if cannibals
                                  %   outnumber missionaries
                                  %   on the same bank:
#fluent = #inertial_fluent.

#action = move(#numC, #numM, #location).
```

Practice: Write an $\mathcal{AL}$ system description

1. Moving objects increases the number of objects at the destination by the amount moved. (3 laws)
2. 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. (3 laws)
3. There cannot be different numbers of the same type of person at the same location. (2 laws)
4. A boat cannot be in and not in a location.
5. A boat cannot be in two places at once.
6. There will be casualties if cannibals outnumber missionaries.
7. It is impossible to move more than two people at the same time; it is also impossible to move less than 1 person. (2 laws)
8. It is impossible to move people without a boat at the source.
9. It is impossible to move N people from a source if there are not at least N people at the source in the first place. (2 laws)

The Program

Here's the SPARC program:
`http://pages.suddenlink.net/ykahl/s_crossing.txt`
Here's a plan:

```
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)
```

Using Domain-Specific Knowledge

- *The efficiency of ASP planners can be substantially improved by expanding a planning module by domain dependent heuristics represented by ASP rules.*
- Example from blocks-world: Why pick up a block just to put it back in the same location?
- We'll ban such actions:

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

Heuristics Based on Subgoals I

- Suppose we have a heuristic that is based on knowledge of subgoals.
- We need a way of separating this knowledge from the general definition of a goal.
- Here's a blocks-world example that can be *easily generalized.*

Heuristics Based on Subgoals II

```
% This is our original goal:
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).

% This is how we add subgoal information:
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).
```

Heuristics Based on Subgoals III

- Example heuristic: Only consider moving blocks that are out of place:

```
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).
```

Quality of Plans

- The last heuristic greatly improved the speed of the planner.
- However, if we don't have the perfect horizon, we can get a lot of non-optimal plans.
- (In many domains, even if we have plans of the same length, some may be better than others.)

Make Good Moves

Here's a heuristic that eliminates a large number of nonoptimal plans by considering only those moves which increase the number of blocks placed in the right position:

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

Using the Heuristic: Prohibit Bad Moves

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

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

Why can't we just say:

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

Concurrent Planning

- For the module using the choice rule:

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

  Here `m` is the maximum number of actions which can be performed simultaneously.
- If using the planning module with disjunction, simply remove the rule prohibiting simultaneous actions.

Minimal Plans with CR-Prolog (SPARC Version)

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

% Consider occurrences of actions if necessary to resolve
% a contradiction. Use cardinality preferences.
occurs(A,I) :+.

% Don't procrastinate:
something_happened(I) :- occurs(A,I).
:- not something_happened(I),
   something_happened(I+1).

% No concurrency:
-occurs(A2,I) :- occurs(A1,I), A1 != A2.
```

Minimal Plans with Clingo

- We can also compute minimal plans by using a special form of the minimize statement in Clingo.
- Syntactically, the statement has the form

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

  where $s_1, \ldots, s_n$ are sorts of parameters of q.
- Instructs the solver to compute only those answer sets of the program which contain the smallest number of occurrences of atoms formed by predicate symbol q.
- For planning, add statement:

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