Lecture Ten: Planning Agents

301315 Knowledge Representation and Reasoning

Intelligent Agents

The Classical Planning Problem

The Declarative Approach to Planning
- Representing the Domain Description
- Representing the Goal
- Achieving the Goal
- Generating Actions
- The Horizon

Example: The Blocks World
- Advantages
- Multiple Goal States
- Using Defined Fluents in the Goal

Complex Goals

Example: Igniting the Burner

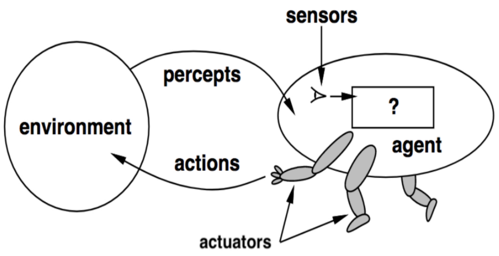
Missionaries and Cannibals

Heuristics - The Blocks World Revisited

Tutorial and Lab Exercises

What an Intelligent Agent Does

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.



What an Intelligent Agent Does

What an agent does:

(a) to observe the world, checks that its observations are consistent with its expectations, and updates its knowledge base;

(b) to select an appropriate goal G;

(c) to search for a plan (a sequence of actions) to achieve G;

(d) to execute some initial part of the plan, updates the knowledge base, and goes back to step (a).

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.
- The **problem**: Given a description of a deterministic dynamic system, its current state, and a goal, we need to find a plan to achieve this goal.

A sequence 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 action sequence α.

The Classical Planning Problem

We can use a declarative approach to solve the planning problem. The procedure is as follows:

- Use $\mathcal{AL}$ to represent information about an agent and its domain, e.g., system/domain description.
- Translate it into an ASP program.
- Add the initial state to the program.
- Add the goal to the program.

The Classical Planning Problem

- Add the **simple planning module** - a small, domain-independent ASP program.
- Use a solver to compute the answer sets of the resulting program.
- A plan is a collection of facts formed by relation occurs which belong to such an answer set.

The Declarative Approach to Planning - Representing the Domain Description

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

The Declarative Approach to Planning - Representing the Goal

- Relation goal holds if and only if 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 $G = \{f_1, \cdots, f_m\} \cup \{\neg g_1, \cdots, \neg g_n\}$.

The Declarative Approach to Planning - Achieving the Goal

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

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

The Declarative Approach to Planning - Generating Actions

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

%% Note: we can also use choice rule to replace
%% the above disjunction:
1{occurs(A,I): action(A)}1 :- step(I),
                              not goal(I),
                              I < n.

%% Do not allow concurrent actions - two actions
%% cannot occur at the same time:
:- occurs(A1,I),
   occurs(A2,I),
   A1 != A2.
```

The Declarative Approach to Planning - Generating Actions

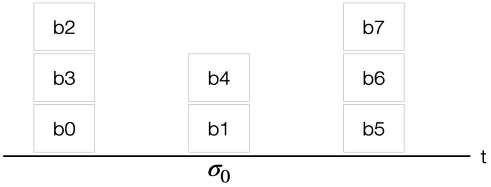
```
%% An action occurs at each step before
%% the goal is achieved:
something_happened(I) :- occurs(A,I).

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

%% Note: all rules above should be safe, revise them
%% if necessary.
```

The Declarative Approach to Planning - The Horizon

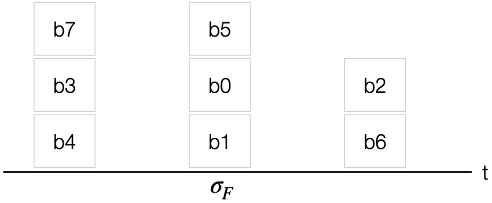
- Our planning module requires a **horizon** - a limit on the length of allowed plans.
- Constant n, which represents the limit on the number of steps in our trajectory, is set to the horizon.
- There is no known way of finding minimal plans with straight ASP unless you call the program with $n = 1, 2, \cdots$ until you get an answer set.

Example: The Blocks World

Example (The Blocks World)



Action: `put(B,L).`



Example: The Blocks World

Example (The Blocks World (continued))

- Use the $\mathcal{AL}$ description from before.
- Translate to ASP (as before).
- Setting horizon and add the initial state to the program:

```
step(0..N) :- N=10. % maximally we allow 10 steps.

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),
                     block(B),block(L).
-holds(on(B,t),0) :- not holds(on(B,t),0),
                     block(B).
```

Example: The Blocks World

Example (The Blocks World (continued))

- 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).
  ```

- Add the planning module (see earlier slides).
- Using ASP solver *clingo* to compute the answer set of the program.
- A plan is a collection of facts formed by relation occurs which belong to such an answer set, i.e., display the occurs statements.

The Blocks World - 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.

The Blocks World - 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.

The Blocks World - Using Defined Fluents in the Goal

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

$$occupied(B) \textbf{ if } on(B1, B)$$

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

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

Note that we don't need to add extra CWA for *occupied* because we already have the general CWA for defined fluents.

The Blocks World - Using Defined Fluents in the Goal

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.

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:

$$\neg is\_colored(B, C1) \textbf{ if } is\_colored(B, C2),\ C1 \neq C2.$$

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

Complex Goals

- 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:

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

- 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).
```

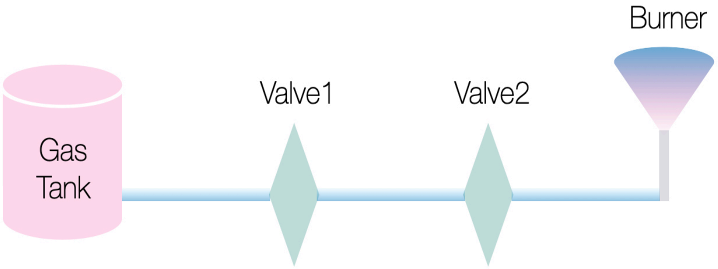
Example: Igniting the Burner

The scenario:

> *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.*

Our goal: To turn on the burner.

Example: Igniting the Burner



Example: Igniting the Burner

Defining signature

- sort section = s1, s2, s3.
- sort valve = v1, v2.
- statics: $connected\_to\_tank(s1)$, $connected\_to\_burner(s3)$, $connected(s1, v1, s2)$, and $connected(s2, v1, s3)$,
- inertial fluents: $opened(V)$ and $burner\_on$.
- defined fluent: $pressurized(S)$.
- actions: $open(V)$, $close(V)$, and $ignite$.

Example: Igniting the Burner

System Description:

State constraints:

$$
\begin{array}{l}
\mathit{pressurized}(S)\ \textbf{if}\ \mathit{connected\_to\_tank}(S). \\
\mathit{pressurized}(S2)\ \textbf{if}\ \mathit{connected}(S1, V, S2), \\
\qquad\qquad\qquad\qquad \mathit{opened}(V), \\
\qquad\qquad\qquad\qquad \mathit{pressurized}(S1).
\end{array}
$$

Example: Igniting the Burner

Causal laws:

open(V) **causes** *opened*(V).
close(V) **caused** $\neg$*opened*(V).
ignite **causes** *burner_on*.

Executability conditions:

impossible *open*(V) **if** *opened*(V).
impossible *open*$(V1)$ **if** *connected*$(S1, V1, S2)$,
connected$(S2, V2, S3)$,
opened$(V2)$.
impossible *close*(V) **if** $\neg$*opened*(V).
impossible *ignite* **if** *conneced_to_burner*(S),
$\neg$*pressurized*(S).

Example: Igniting the Burner

Now suppose initial state is:

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

Our goal is: *burner_on*.

How can we generate a plan to achieve this goal?

A possible plan can be discovered as follows:

```
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?

Missionaries and Cannibals

Objects:

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

Statics: opposite(#loc, #loc)
Fluents:

- inertia fluents:

m(#loc, #num)	num missionaries at loc (bank1 or bank2)
c(#loc, #num)	num cannibals at loc
b(#loc, #num_boat)	num boat (0 or 1) at loc

- We introduce a new defined fluent *casualties*, which becomes true if the cannibals outnumber the missionaries.

Missionaries and Cannibals

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).
  ```

Missionaries and Cannibals

Rules

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

$$
\begin{aligned}
&\mathit{move}(NC, NM, \mathit{Dest})\ \textbf{causes}\ m(\mathit{Dest}, N + NM)\\
&\qquad\qquad\qquad\qquad\qquad\ \ \textbf{if}\ m(\mathit{Dest}, N)\\
&\mathit{move}(NC, NM, \mathit{Dest})\ \textbf{causes}\ c(\mathit{Dest}, N + NC)\\
&\qquad\qquad\qquad\qquad\qquad\ \ \textbf{if}\ c(\mathit{Dest}, N)\\
&\mathit{move}(NC, NM, \mathit{Dest})\ \textbf{causes}\ b(\mathit{Dest}, 1)
\end{aligned}
$$

Missionaries and Cannibals

- The number of missionaries/cannibals at the opposite bank is: $(3 - \mathit{number\_on\_this\_bank})$. The number of boats at the opposite bank is $1 - \mathit{number\_of\_boats}$ on this bank.

  $m(\mathit{Source}, 3 - N)$ **if** $m(\mathit{Dest}, N), \mathit{opposite}(\mathit{Source}, \mathit{Dest})$
  $c(\mathit{Source}, 3 - N)$ **if** $c(\mathit{Dest}, N), \mathit{opposite}(\mathit{Source}, \mathit{Dest})$
  $b(\mathit{Source}, 1 - NB)$ **if** $b(\mathit{Dest}, NB), \mathit{opposite}(\mathit{Source}, \mathit{Dest})$

Missionaries and Cannibals

- There cannot be different numbers of the same type of person at the same location:
  $\neg m(Loc, N1)$ **if** $m(Loc, N2), N1 \neq N2$
  $\neg c(Loc, N1)$ **if** $c(Loc, N2), N1 \neq N2$
- A boat cannot be in and not in a location:
  $\neg b(Loc, NB1)$ **if** $b(Loc, NB2), NB1 \neq NB2$
- A boat cannot be in two places at once:
  $\neg b(Loc1, N)$ **if** $b(Loc2, N), Loc1 \neq Loc2$

Missionaries and Cannibals

- There will be casualties if cannibals outnumber missionaries:

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

- It is impossible to move more than two people at the same time; it is also impossible to move less than 1 person:

$$\textbf{impossible}\ \textit{move}(NC, NM, Dest)\ \textbf{if}\ (NC + NM) > 2$$
$$\textbf{impossible}\ \textit{move}(NC, NM, Dest)\ \textbf{if}\ (NM + NC) < 1$$

Missionaries and Cannibals

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

```
impossible move(NC, NM, Dest)
        if opposite(Source, Dest),
           b(Source, 0)
```

- 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:

```
impossible move(NC, NM, Dest)
        if opposite(Source, Dest),
           m(Source, NMSource),
           NMSource < NM
impossible move(NC, NM, Dest)
        if opposite(Source, Dest),
           c(Source, NCSource),
           NCSource < NC
```

Missionaries and Cannibals

A possible 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)
```

Heuristics - - The Blocks World Revisited

The efficiency of ASP planners can be substantially improved by expanding a planning module by domain dependent heuristics represented by ASP rules.

Heuristic 1:
In the previous Blocks World domain, our plan could contain an action like $put(B, L)$ even if block B is already on location L. This can be avoided by adding a rule:

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

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

Heuristic - - The Blocks World Revisited

Heuristic 2:
Another heuristic is to extend the planning module by adding a useful rule which tells the planner to only consider moving the blocks that are out of place.

```
%% 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).

%% 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).
```

Heuristic - - The Blocks World Revisited

Only 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).
```

Heuristic - - The Blocks World Revisited

Heuristic 3:
Making good moves - to eliminate a large number of non-optimal 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), step(I).
```

Heuristic - - The Blocks World Revisited

Heuristic 4:
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).
```

Note that we cannot just use a rule like:

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

because this will lead to inconsistency. Do you know why?

Tutorial and Lab Exercises

1. In our Blocks World domain, consider to replace the following rules:

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

   by this one rule:

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

   What will happen.

Tutorial and Lab Exercises

2. 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.*

Tutorial and Lab Exercises

3. Complete the ASP program for solving Missionaries and Cannibals, and generate a plan for this problem by running your program on *clingo*.
4. Revise your Blocks World ASP program by adding those four heuristics into the program, and run your new program on *clingo*.