Advanced Topics on Artificial Intelligence

Alban Grastien

The Australian National University

Second Semester, 2020

How to Represent MDPs

- MDPs can include a huge number of states.
- It is impossible to enumerate all these states, all the transitions, without making a mistake, even for smaller problems.
- Yet, the problem can often be simply described.
- Think game of go:
  - The rules can be written on a small page.
  - The number of states is astronomical.

What is a Good Representation

- One goal is to come up with general-purpose planners
  - Cf. Atari games
- A good representation should be easy to write
  - Compact
  - Intuitive (in particular for non-experts)
  - No pit falls!
- A good representation should make the planning task easy. That includes:
  - Expose constants and mutexes
  - Make it easy to compute heuristics

Probabilistic Planning Domain Description Language PPDDL

H. Younes and M. Littman. *PPDDL: The probabilistic planning domain definition language*. Technical Report. 2004.

Probabilistic Planning Domain Description Language

```
(define (domain test-domain)
  (:requirements :typing :equality
     :conditional-effects :fluents)
  (:types car box)
  (:constants goldie - car)
  (:predicates (parked ?x - car) (holding ?x - box)
     (in ?x - box ?y - car))
  (:functions (fuel-level ?x - car))
  (:action load :parameters (?x - box ?y - car)
   :precondition (and (holding ?x) (parked ?y))
   :effect (and (in ?x ?y)(forall (?z - car)
     (when (not (= ?z ?y))(not (in ?x ?z))))))
  (:action refuel :parameters (?x - car)
   :precondition (< (fuel-level ?x) 10)
   :effect (increase (fuel-level ?x) 1))
)
```

Types and Objects

- *Types* form a hierarchy of objects, the most abstract of which is `object`.
  - for instance `truck` and `plane` are subtypes of `vehicle`.
  - `location` is also a type.
  - both `vehicle` and `location` are subtypes of `object`.
- *Objects* refer to real-world instances of the types.
  - `truck_1` and `truck_2` are instances of `truck`, and therefore of `vehicle`.
  - `airport_1` and `warehouse_1` are instances of `location`.

Predicates and Facts

- *Predicates* are properties that link objects of given types. They have a signature that indicates what type of objects they link:
  - `at_loc( ?t - vehicle ?l - location)` is a predicate with signature `vehicle,location`.
- *Facts* are instantiation of predicates with objects:
  - `at_loc(truck_1, warehouse_1)` models the fact that the truck 1 is in the first warehouse.

Action Schemas and Actions

- *Action Schemas* are general definitions of actions that can be instantiated.
- An action schemas includes a name, a signature, a precondition, and a list of effects (more about effects later).

```
(:action drive
  :parameters (?t - truck ?lstart ?lend - location)
  :precondition (and (at_loc ?t ?lstart) (road ?lstart ?lend))
  :effect (and (not (at_loc ?t ?lstart)) (at_loc ?t ?lend))
)
```

- Action `drive(truck_1,warehouse_1,airport_1)` is an instantiation of the `drive` schema.

Functions and Rewards

- *Functions* are used to represent numeric variables.
  Warning: numeric variables make state space infinite, and problems undecidable in general.
- Functions are defined by schemas:
  `(:functions (fuel-level ?t - truck))`
- Functions can be called in conditions: `(> (fuel-level ?t) 100)`
- Functions can be updated via action effects `assign`, `scale-up`, `scale-down`, `increase`, `decrease`:
  `(increase (fuel-level ?t) 100)`
- The *reward* function is accessed by the function `reward`.

Conditional Effects

- Action preconditions and *conditional effects* are different:
  - An action precondition indicates when an action is applicable.
  - A conditional effect indicates when some effect applies.

  The distinction is important when the current state is only partially known.
  Example:
  - Can you perform “move forward” if there is a wall in front of you?

- A conditional effect is represented by the keyword `when`:

```
(when (<= (fuel-level ?t) 0) (stuck ?t))
```

Uncertainty

- *Probabilistic effects* are represented as

  $\boxed{\texttt{(probabilistic } p_1 \ \ e_1 \ \ \ldots \ \ p_k \ \ e_k\texttt{)}}$

  where each p1 is a probability so that $\Sigma_i \ p_i \leq 1$.

- When the effect is supposed to trigger, then effect e_i triggers (and e_j does not if $j \neq i$) with probability p_i.

- With probability $1 - \Sigma_i \ p_i$, no effect triggers.

Initial State and Goal

- *Initial state* are defined as a conjunction of probabilistic facts.

```
(:init (and  (at_loc truck_1 warehouse_1)
             (probabilistic  .5 (at_loc plane_1 airport_1)
                             .5 (at_loc plane_1 airport_2))
             ...
))
```

- The *goal* is a formula on the facts and fluent values.

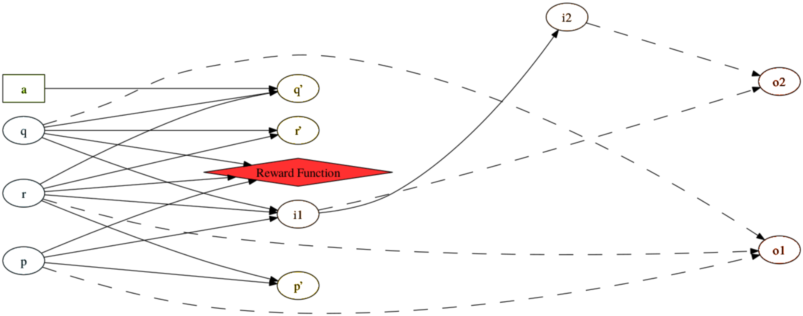
```
(:goal (or  (> (fuel-level truck_1) 20)
            (at_loc plane_1 warehouse_1))
)
```

Relational Dynamic Influence Diagram Language RDDL

S. Sanner, *Relational dynamic influence diagram language (RDDL): language description*, Technical Report, 2010.

Slides heavily inspired by Scott Sanner's Technical Report

Influence Diagram



Left:

- Circles: state fluents
- Square: action fluents

Middle and right:

- Circles: intermediate & next state fluents
- Diamond: reward

Define Variables

```
domain prop_dbn  {
  requirements = { reward-deterministic };

  pvariables {
  // State fluents
  p : { state-fluent , bool , default = false };
  q : { state-fluent , bool , default = false };
  r : { state-fluent , bool , default = false };
  // Action fluents
  a : { action-fluent , bool , default = false };
  // Intermediate fluents
  i1 : { interm - fluent , int , level = 1 };
  i2 : { interm - fluent , int , level = 2 };
  // Observable fluents
  o1 : { observ - fluent , bool };
  o2 : { observ - fluent , real };
  };
...
```

Define Values

```
...
  cpfs {
    p’ = if ( p ^ r ) then Bernoulli (.9) else Bernoulli (.3);

    ...

    i1 = KronDelta(p + r + q)

    ...

    o1 = Bernoulli ( ( p + q + r )/3.0 );
  }

  reward = p - r + i2
}
```

Define Problem Instance

```
instance inst_dbn {
  domain = prop_dbn2 ;
  init-state { p ; r ; };
  max-nondef-actions = 1;
  horizon = 20;
  discount = 0.9;
}
```

Lifted (Parametrised) Representation (1/2)

```
domain game_of_life {
...
  types {
    x_pos : object ;
    y_pos : object ;
  };

  pvariables {
    alive ( x_pos , y_pos ) :
      { state-fluent , bool , default = false };
  };

  cpfs {
    alive'(?x ,?y ) = ...
  }
}
```

Lifted (Parametrised) Representation (2/2)

```
non-fluents game2x2 {
  domain = game_of_life ;
  objects {
    x_pos : { x1 , x2 };
    y_pos : { y1 , y2 };  };
  non-fluents {
    // can define constants here
  };
}
instance is1 {
  domain = game_of_life ;
  non-fluents = game2x2 ;
  init-state {
    alive ( x1 , y1 );
    alive ( x2 , y2 );  };
  ...
}
```

Scikit-decide

https://airbus.github.io/scikit-decide/

Description

- Scikit-decide is a framework for decision making under uncertainty
- Compatible with AI gym

How to Define an MDP (1)

https://airbus.github.io/scikit-decide/guide/codegen.html

```
class D(MDPDomain):
    T_state = State
    T_observation = T_state
    T_event = Action
    T_value = float
    T_info = None

class MyDomain(D):
  def _get_transition_value(self, memory: D.T_state,
    action: D.T_event, next_state: Optional[D.T_state] = None)
  -> TransitionValue[D.T_value]: pass

  def _get_next_state_distribution(self, memory: D.T_state,
    action: D.T_event)
  -> DiscreteDistribution[D.T_state]: pass
```

How to Define an MDP (2/2)

https://airbus.github.io/scikit-decide/guide/codegen.html

```
def _is_terminal(self, state: D.T_state)
  -> bool: pass

def _get_applicable_actions_from(self, memory: D.T_state)
  -> Space[D.T_event]: pass

def _get_action_space_(self)
  -> Space[D.T_event]: pass

def _get_initial_state_(self)
  -> D.T_state: pass

def _get_observation_space_(self)
  -> Space[D.T_observation]: pass
```