Lab 1: Plan Synthesis

Michael Cashmore
12/10/2020

ICAPS 2020 Online Summer School
Lab 1: Plan Synthesis

- PS-1: Introductory Lecture.
- PS-2: Hands-on session.
- PS-3: Hands-on Session and Solution Walk-through.
Goals of this lab:

- Learn what domain-independent automated planning is.
- Gain hands-on experience modelling planning problems.

Outline:

- Domain-independent Automated Planning.
- Introduction to the Planning Domain Definition Language.
- Lab materials (inc. Online editor).
- PDDL2.1 Time and Numbers
- Lab materials (Exercise 4).
Domain-independent Automated Planning
What is Planning?

**Short Definition:** Planning is the act of thinking before acting.
What is Planning?

Short Definition: Planning is the act of thinking before acting.

Longer Definition: Planning is the process of choosing and organising actions that lead towards a goal, based on a high-level description of the world.
Domain specific planning uses representation or methods that are adapted to solving a specific problem.

- *many important domains*: path and motion planning, manipulation planning, communication planning, etc.
Domain Specific vs. Domain-Independent Planning

Domain specific planning uses representation or methods that are adapted to solving a specific problem.

• *many important domains:* path and motion planning, manipulation planning, communication planning, etc.

Domain-independent planning uses a general representation and technique that is applicable across different domains.

• *still many kinds of general planning:* online and offline; discrete and continuous; deterministic and non-deterministic; fully- and partially observable; sequential and temporal.
What is automated planning used for?
Introduction to Planning Domain Definition Language (PDDL)
The main components of a planning problem are:

- a description of how the world behaves and the capabilities of the agent (e.g. the action library).
- a description of the initial situation (the initial state).
- a description of the desired situation (the goal).
The main components of a planning problem are:

- a description of how the world behaves and the capabilities of the agent (e.g. the action library).
- a description of the initial situation (the initial state).
- a description of the desired situation (the goal).

A basic planning formalism represents the state of the world and actions using propositional variables. Such a (classical) planning problem is a tuple: \( < F, A, I, G > \), where:

- \( F \) is a set of (Boolean) propositions.
- \( A \) is a set of deterministic actions.
- The set of states \( S \) is the power set of \( F \), \( S = 2^F \).
- \( s_o \in S \) is the initial state.
- \( G : S \rightarrow \{ \top, \bot \} \) is the goal function.
A (classical) planning problem is a tuple: \( < F, A, I, G > \), where:

- \( F \) is a set of (Boolean) propositions.
- \( A \) is a set of deterministic actions.
- The set of states \( S \) is the power set of \( F \), \( S = 2^F \).
- \( s_o \in S \) is the initial state.
- \( G : S \rightarrow \{ \top, \bot \} \) is the goal function.

Each action \( a \in A \) consists of:

- \( pre(a) \subseteq F \) (simple preconditions)
- \( add(a) \subseteq F \) (add effects)
- \( del(a) \subseteq F \) (delete effects)
PDDL is a language for encoding classical planning tasks. Tasks are separated into two files:

1. **Domain File**, which contains:
   - **Predicates** that describe the properties of the world.
   - **Operators** that describe the way in which the state can change.

2. **Problem File**, which contains:
   - **Objects**: the things in the world.
   - The **initial state** of the world.
   - The **goal** specification.

\( F \) and \( A \) are found by applying the object terms to the predicates and operators.
(define (domain simple_switches)
  (:requirements :typing)
  (:types switch)
  (:predicates
    (off ?s - switch) (on ?s - switch))
  (:action switch_on
    :parameters (?s - switch)
    :precondition (off ?s)
    :effect (and (not (off ?s)) (on ?s)))
)
(define (problem more_switches)
  (:domain simple_switches)
  (:objects s1 s2 s3 - switch)
  (:init (off s1) (off s2) (off s3))
  (:goal (and (on s1) (on s2) (on s3)))
)
A **plan** for a classical planning problem is a sequence of actions that are applicable from the initial state and lead to a state that satisfies the goal:

\[ \langle a_0, \ldots, a_n \rangle \]

```
(switch_on s1)
(switch_on s2)
(switch_on s3)
```
Online Editor: planning.domains
http://editor.planning.domains/
http://editor.planning.domains/
http://editor.planning.domains/
http://editor.planning.domains/

- Simple switches:
  http://editor.planning.domains/#read_session=jfespcjFc3
- More switches:
  http://editor.planning.domains/#read_session=iseLBtK6jo
- Tricky Switches:
  http://editor.planning.domains/#read_session=ob1iWAQRp
PDDL2.1: Temporal Planning
Temporal Planning

- Up until now we have used classical planning: time is a sequence of states and actions are instantaneous.
Temporal Planning

• Up until now we have used classical planning: time is a sequence of states and actions are instantaneous.
• When multiple things can be happening at a time, it is necessary to model the duration and concurrency of actions and events.
• Actions and events may have complex inter-dependencies which determine which combinations are possible.
Temporal Planning

• Up until now we have used classical planning: time is a sequence of states and actions are instantaneous.
• When multiple things can be happening at a time, it is necessary to model the duration and concurrency of actions and events.
• Actions and events may have complex inter-dependencies which determine which combinations are possible.
Temporal Planning

• Up until now we have used classical planning: time is a sequence of states and actions are instantaneous.
• When multiple things can be happening at a time, it is necessary to model the **duration** and **concurrency** of actions and events.
• Actions and events may have complex inter-dependencies which determine which combinations are possible.
• Resource: Planning wiki (https://planning.wiki/ref)
• Resource: Planning editor (planning.domains)
Fox and Long introduced PDDL2.1 (and PDDL+) to increase the expressiveness of PDDL to more realistic problems:

- Level 1: STRIPS and ADL.
- Level 2: **Numeric variables and optimisation metrics.**
- Level 3: **Durative Actions.**
- Level 4: Continuous Change.
- Level 5: Processes and Events.

Formally a **Temporal Planning Problem** is a tuple:
\[ \Pi = \langle F, V, A, I, G \rangle, \]
where:

- \( F \) is a set of (Boolean) propositions.
- \( V \) is a set of (Real) *primitive numeric expressions* (PNEs/functions).
- \( A \) is a set of deterministic actions.
- \( I \) is the initial state.
- \( G : S \rightarrow \{\top, \bot\} \) is the goal function.
A state is now a combination of time and both Boolean and numeric variables: $S$ is a tuple $(time, s_{\text{logical}}, s_{\text{numeric}})$.

- $time \in \mathbb{R}$ is the time of the state.
- $s_{\text{logical}} \subseteq F$ is the logical state.
- $s_{\text{numeric}} : V \rightarrow \mathbb{R}_{\perp}$ the assignment to the numeric expressions, where $\perp$ denotes an undefined value.

For example: $(0, l_{\text{logical}}, x)$ is the initial state, where $x$ assigns each numeric function $v \in V$ to a value in $\mathbb{R}_{\perp}$ (the initial numeric assignments).
Below is an example initial state (with $time = 0$)

(:init
   (at truck Rome)
   (at car Paris)
   (= (fuel-level truck) 100)
   (= (fuel-level car) 100)
)
An action consists of:

- an action name,
- (typed) parameters,
- a duration constraint,
- \textit{at start}, \textit{over all}, and \textit{at end} conditions,
- \textit{at start}, \textit{over all}, and \textit{at end} effects.
• **Comparisons** between **numeric expressions** can be used as logical axioms:

\[
\geq (\text{fuel}) \times (\text{distance from to}) \times (\text{fuel consumption})
\]
• **Comparisons** between **numeric expressions** can be used as logical axioms:

\[
> = (fuel) \times (distance \ ?from \ ?to) (fuel\_consumption))
\]

• **Effects can modify functions** by numeric expressions:

\[
decrease (fuel) \times (distance \ ?from \ ?to) (fuel\_consumption))
\]
• **Comparisons** between numeric expressions can be used as logical axioms:
  \[ (>=(fuel) (* (distance \textit{from} \textit{to}) (fuel\_consumption))) \]

• **Effects can modify functions** by numeric expressions:
  \[ (decrease (fuel) (* (distance \textit{from} \textit{to}) (fuel\_consumption))) \]

• **Actions take an amount of time** given by the value of the numeric expression \textit{?duration}, which is constrained by a comparison:
  \[ (= ?duration (/ (distance \textit{from} \textit{to}) (?speed))) \]
• **Comparisons** between numeric expressions can be used as logical axioms:
  \((\geq (\text{fuel}) (\ast (\text{distance} \ ?\text{from} \ ?\text{to}) (\text{fuel}\_\text{consumption}))))\)

• **Effects can modify functions** by numeric expressions:
  \((\text{decrease} (\text{fuel}) (\ast (\text{distance} \ ?\text{from} \ ?\text{to}) (\text{fuel}\_\text{consumption}))))\)

• **Actions take an amount of time** given by the value of the numeric expression \(?\text{duration}\), which is constrained by a comparison:
  \((= ?\text{duration} (/ (\text{distance} \ ?\text{from} \ ?\text{to}) (?\text{speed}))))\)

Let’s look at an example of the syntax...
(:durative-action attend_lecture
 :parameters (?s - student ?l - lecturer ?r - room)
 :duration (<= ?duration 120)
 :condition (and
    (at start (awake ?s))
    (at start (in ?l ?r))
    (at start (in ?s ?r))
    (over all (awake ?l)))
 :effect (and
    (at end (not (awake ?s))))
(:durative-action attend_lecture)

A durative action is defined differently from an instantaneous action (use *durative-action* instead). You can include both types of action in the domain.
The parameters of an action can be typed.
The parameters of an action can be typed. Types can be compiled away using unary type predicates. For example:

(:objects student01)
(:init (is_a_student student01))
:duration (<= ?duration 120)

Duration constraints are expressed as a comparison with the special numeric expression ?duration.

- Comparison operators: <, >, <=, >=, =.
Numeric expressions are either:

- A constant value, e.g. 120,
- a PDDL function, e.g. \((\text{distance} \ ?\text{road})\),
- the unary operator \((- \ expression)\)
- or a binary operation with operators: +, −, *, /.
PDDL2.1 Durative Action Syntax

:condition (and
  (at start (awake ?s))
  ...
)

- *at start* conditions must be true in the state that the action is applied.
- *at end* conditions must be true in the state that the actions is completed.
- *over all* conditions must be true throughout the duration of the action.

Note that the value of a function must be made true at least a little time (\(\epsilon\)) before it is used to satisfy a condition. This is called *epsilon separation*.
:effect (and
(at end (not (awake ?s))))

- Effects can be at start or at end.
- Numeric Effects can increase, decrease, or assign the values of primitive numeric assignments. Example: (assign (?fuel) (?max_fuel_capacity))
(define (domain matchcellar)
 (:requirements :typing :durative-actions)
 (:types match fuse)
 (:predicates
  (light ?m - match)
  (handfree)
  (unused ?m - match)
  (mended ?f - fuse)
 )

(:durative-action light_match
 :parameters (?m - match)
 :duration (= ?duration 8)
 :condition (and
  (at start (unused ?m)))
 :effect (and
  (at start (not (unused ?m)))
  (at start (light ?m))
  (at end (not (light ?m))))

(:durative-action mend_fuse
 :parameters ( ?f - fuse 
  ?m - match)
 :duration (= ?duration 5)
 :condition (and
  (at start (handfree))
  (over all (light ?m)))
 :effect (and
  (at start (not (handfree)))
  (at end (mended ?f))
  (at end (handfree)))))
(define (domain matchcellar)
  (:requirements :typing :durative-actions)
  (:types match fuse)
  (:predicates
    (light ?m)
    (handfree)
    (unused ?m - match)
    (mended ?f - fuse)
  )
  (:durative-action light_match
    :parameters (?m - match)
    :duration (= ?duration 8)
    :condition (and
      (at start (unused ?m)))
    :effect (and
      (at start (not (unused ?m)))
      (at start (light ?m)))
      (at end (not (light ?m))))

  (:durative-action mend_fuse
    :parameters (?f - fuse
      ?m - match)
    :duration (= ?duration 5)
    :condition (and
      (at start (handfree))
      (over all (light ?m)))
    :effect (and
      (at start (not (handfree)))
      (at end (mended ?f))
      (at end (handfree))))
)
(define (problem fixfuse)
  (:domain matchcellar)
  (:objects
    match1 match2 - match
    fuse1 fuse2 - fuse)
  (:init
    (unused match1)
    (unused match2)
    (handfree))
  (:goal (and
    (mended fuse1))
  )
Plan Timeline

0.00: light_match match1 [8.00]
0.01: fix_fuse fuse1 match1 [5.00]
Plan Timeline

- Light Match
- Mend Fuse

Timepoints: $t_0$, $t_1$, $t_2$, $t_3$
Plan Timeline

- Conditions are above the action and red.
- *Over all* conditions are below the middle of the action in red.
- Effects are below the action and in blue.
Timed initial literals are defined in the initial state:

\[
(:\text{init} \\
\hspace{1em} (\text{at} \ 20 \ (\text{available \ match1})) \\
\hspace{1em} (\text{at} \ 40 \ (\text{not} \ (\text{available \ match1}))))
\]

leads to a time window in which \textit{match1} can be used.
40.01: light_match match1 [8.00]
40.02: fix_fuse fuse1 match1 [5.00]
Online Editor: Temporal Planning
Use the following plugins to enable a temporal solver and timeline view:

The following link already has these enabled:
http://editor.planning.domains/#read_session=EWjbgnhuUd
• Packages can be loaded into and unloaded from trucks (10 time units).
• Drivers can walk between connected waypoints at a speed of 0.5.
• Drivers can get into and out of trucks (10 time units).
• Trucks with drivers can drive between connected waypoints at a speed of 1.
Temporal Logistics
• The plane starts in the sky waypoint, the boat starts in the lighthouse waypoint.
• The boat and the plane don’t need drivers to move.
• They can only travel over the blue and yellow edges (connected to the lighthouse and the sky).
• The boat travels at a speed of 1.5.
• The plane travels at a speed of 2.
Goal:
Extra Challenges

- Each truck can only make a maximum of 7 trips.
- The plane must wait a minimum of 20 time units between trips.
- Each driver must return to waypoint 1 and disembark at least once within each 400 time unit interval. This can be at the start or end of the time interval, for example at: 399, 401, 850, ....
- Trucks can make any number of trips, but consume 1 unit of fuel for each time unit they are travelling. Trucks can be refuelled at stations in waypoints 3 and 9.
- Trucks also consume 0.1 fuel per time unit when they are not driving.

Note: these extra challenges may create a problem that is too difficult to solve within the time limit given to the online solver.