LISV Laboratoire d'ingénierie des systèmes de Versailles | UVSQ université PARIS-SACLAY

AI Planning Lab 2

Halim Djerroud

revision 1.0

Lab 1: Getting Started with PDDL using `editor.planning.domains`

Learning Objectives

By the end of this lab, you will be able to:

- Understand the structure of a PDDL **domain** file (types, predicates, actions)
- Create a PDDL **problem** file (objects, initial state, goal)
- Use the online planner at editor.planning.domains to generate and **execute** a plan

Scenario: The *Gripper* Domain

Imagine a robot in room `rooma` with several balls scattered on the floor. The robot has two grippers (`left` and `right`) that can each hold one ball. The robot's task is to transport all balls to another room called `roomb`.

Available actions:

- `move`: The robot can move from one room to another
- `pick`: The robot can pick up a ball with a free gripper
- `drop`: The robot can drop a ball it's currently holding

Exercise 1: Write and Test Your First PDDL Domain/Problem

Goal: Write the minimum necessary PDDL code to generate a plan that moves two balls from `rooma` to `roomb`.

Step-by-Step Instructions

Part A: Setup (5 minutes)

1. Open your web browser and go to `editor.planning.domains`
2. In the left panel, you'll see a file explorer. Create two new files:
   - Click the "+" button and name the first file: `domain.pddl`
   - Click the "+" button again and name the second file: `problem.pddl`
3. You should now see both files listed in the left panel

Part B: Writing the Domain File (20 minutes)

The domain file describes the "rules of the world" – what types of objects exist and what actions are possible.

Step 1: Define the domain header Click on `domain.pddl` and start with the basic structure:

```
(define (domain gripper-simple)
  (:requirements :strips :typing)

  ; We'll add more here in the next steps
)
```

Explanation:

- `(define (domain gripper-simple))` – Names our domain
- `(:requirements :strips :typing)` – Declares we're using basic PDDL with types

Step 2: Define the types Inside the domain (after the requirements line), add:

```
  (:types
    room      ; locations where the robot can be
    ball      ; objects to transport
    gripper   ; robot's hands/pincers
  )
```

Explanation: Types categorize the objects in our world. Think of them as categories or classes. Every object we create later must belong to one of these types.

Step 3: Define the predicates Predicates describe the state of the world – what can be true or false at any moment. Add:

```
  (:predicates
    (at-robot ?r - room)              ; robot is in room ?r
    (at ?b - ball ?r - room)          ; ball ?b is in room ?r
    (free ?g - gripper)               ; gripper ?g is empty
    (carry ?b - ball ?g - gripper)    ; gripper ?g holds ball ?b
  )
```

Explanation:

- Variables start with `?` (e.g., `?r`, `?b`, `?g`)
- The `- type` notation specifies what type each variable must be
- These predicates can be either true or false in any given state

Step 4: Define the `move` action Now we define what actions the robot can perform. Start with movement:

```
  (:action move
    :parameters (?from - room ?to - room)
    :precondition (at-robot ?from)
    :effect (and
      (not (at-robot ?from))
      (at-robot ?to)))
```

Explanation:

- `:parameters` – What information does this action need? (where to move from and to)
- `:precondition` – What must be true BEFORE we can execute this action? (robot must be in the starting room)
- `:effect` – What changes AFTER executing this action? (robot is no longer in `?from`, now in `?to`)
- `(not ...)` removes a fact from the world state

Step 5: Define the `pick` action

```
(:action pick
  :parameters (?b - ball ?r - room ?g - gripper)
  :precondition (and
    (at-robot ?r)      ; robot must be in the room
    (at ?b ?r)         ; ball must be in the same room
    (free ?g))         ; gripper must be empty
  :effect (and
    (not (at ?b ?r))   ; ball no longer on the floor
    (not (free ?g))    ; gripper no longer free
    (carry ?b ?g)))    ; gripper now holds the ball
```

Explanation:

- All three preconditions must be true to pick up a ball
- The `(and ...)` combines multiple conditions or effects
- Notice how we remove the old facts and add new ones to reflect the state change

Step 6: Define the `drop` action

```
(:action drop
  :parameters (?b - ball ?r - room ?g - gripper)
  :precondition (and
    (at-robot ?r)      ; robot must be in the room
    (carry ?b ?g))     ; gripper must be holding the ball
  :effect (and
    (at ?b ?r)         ; ball is now in the room
    (free ?g)          ; gripper is now free
    (not (carry ?b ?g)))) ; gripper no longer holds the ball
```

Your complete `domain.pddl` should now look like the code in the correction section.

Part C: Writing the Problem File (15 minutes)

The problem file describes a specific instance: which objects exist, how the world starts, and what we want to achieve.

Step 1: Define the problem header Click on `problem.pddl` and write:

```
(define (problem move-two-balls)
  (:domain gripper-simple)

  ; We'll add more here in the next steps
)
```

Explanation:

- `(define (problem move-two-balls))` – Names our specific problem
- `(:domain gripper-simple)` – Links to the domain file (names must match!)

Step 2: Declare the objects

```
(:objects
  rooma roomb - room          ; two rooms
  ball1 ball2 - ball          ; two balls
  left right - gripper        ; two grippers
)
```

Explanation: These are the concrete objects in our world. Each must have a type that was defined in the domain file.

Step 3: Define the initial state

```
(:init
  (at-robot rooma)     ; robot starts in rooma
  (at ball1 rooma)     ; ball1 is in rooma
  (at ball2 rooma)     ; ball2 is in rooma
  (free left)          ; left gripper is empty
  (free right))        ; right gripper is empty
```

Explanation:

- We list all predicates that are TRUE at the start
- Anything not listed is assumed to be FALSE (closed world assumption)
- Notice we don't say the balls are in `roomb` – they're not!

Step 4: Define the goal

```
(:goal (and
  (at ball1 roomb)     ; ball1 must be in roomb
  (at ball2 roomb)))   ; ball2 must be in roomb
```

Explanation:

- The goal specifies what must be true for success
- Notice we don't specify WHERE the robot ends up – we only care about the balls!
- We also don't care about the gripper states in the final configuration

Your complete `problem.pddl` should now look like the code in the correction section.

Part D: Running the Planner (5 minutes)

1. Make sure both files are saved (they auto-save in the editor)
2. Click the green `Play` button (or `Solve` button) at the top
3. Wait a few seconds while the planner searches for a solution
4. You should see a plan appear in the output panel!

Part E: Understanding the Plan (10 minutes)

Read through the generated plan and verify:

1. Does the robot pick up balls only when it's in the same room?
2. Does the robot use grippers that are free?
3. Does the robot move to `roomb` before dropping the balls?
4. Are the balls in `roomb` at the end?

Discussion questions:

- Could the robot move back and forth multiple times? Would that be efficient?
- What's the advantage of having two grippers vs. one?
- What happens if you change the goal to only move `ball1`?

LISV Laboratoire d'ingénierie des systèmes de Versailles | UVSQ université PARIS-SACLAY

Lab 2: Extending the Gripper Domain

Scenario: The Warehouse Robot

In this exercise, we extend the original *Gripper* domain. The robot is now working in a small warehouse consisting of three rooms:

- `rooma`: the storage area, where all packages start
- `roomb`: a transit room, used as a corridor
- `roomc`: the delivery area, where packages must be delivered

The robot:

- has two grippers (`left`, `right`), each can carry one ball
- can move between rooms, but only along adjacency relations
- can pick up or drop balls, subject to constraints

New constraints:

1. The robot can only move between *adjacent* rooms: `rooma <-> roomb <-> roomc`
2. At least one ball must temporarily pass through `roomb` before reaching `roomc`.

Goal: Deliver all balls to `roomc`, respecting adjacency and transit constraints.

Exercise 1: Extending the Domain File

Step 1: Add adjacency predicate Modify the `domain.pddl` by introducing a new predicate:

```
(:predicates
  (at-robot ?r - room)
  (at ?b - ball ?r - room)
  (free ?g - gripper)
  (carry ?b - ball ?g - gripper)
  (adjacent ?r1 - room ?r2 - room) ; new predicate
)
```

Step 2: Update the move action Now ensure that the robot can only move between adjacent rooms:

```
(:action move
  :parameters (?from - room ?to - room)
  :precondition (and
    (at-robot ?from)
    (adjacent ?from ?to))
  :effect (and
    (not (at-robot ?from))
    (at-robot ?to)))
```

Question: Why do we need both directions of adjacency (e.g. `(adjacent rooma roomb)` and `(adjacent roomb rooma)`)?

Exercise 2: Defining the Problem File

Step 1: Declare the objects

```
(:objects
  rooma roomb roomc - room
  ball1 ball2 ball3 - ball
  left right - gripper)
```

Step 2: Define the initial state

```
(:init
  (at-robot rooma)
  (at ball1 rooma)
  (at ball2 rooma)
  (at ball3 rooma)
  (free left)
  (free right)
  (adjacent rooma roomb)
  (adjacent roomb rooma)
  (adjacent roomb roomc)
  (adjacent roomc roomb))
```

Step 3: Define the goal

```
(:goal (and
  (at ball1 roomc)
  (at ball2 roomc)
  (at ball3 roomc)))
```

Discussion: Even though the goal only mentions `roomc`, the adjacency constraints will force the plan to use `roomb` as an intermediate step.

Exercise 3: Running and Analyzing the Plan

1. Load your new domain and problem files in editor.planning.domains.
2. Run the planner and observe the generated plan.
3. Verify:
   - Does the robot respect adjacency?
   - Does the robot use both grippers efficiently?
   - Are all balls delivered to `roomc`?
4. Compare two strategies:
   (a) Transporting balls one by one
   (b) Transporting two balls at once

Question: Which plan is shorter? Why?

Exercise 4 (Bonus): Adding a Swap Action

As an optional challenge, define a new action `swap`, where the robot exchanges a ball it is carrying with another ball in the same room.

Hints:

- Preconditions: robot must be carrying one ball in a gripper and another ball must be in the room
- Effects: the carried ball is placed in the room, and the new ball is taken in hand

Discussion: How could this action help make plans shorter?

Discussion Questions

- Why is the adjacency predicate important for scalability in larger environments?
- How does introducing a third room change the planning complexity compared to Lab 1?
- Can you generalize this model to represent a delivery network with multiple robots and corridors?