Planning with PDDL

1 Introduction

The search algorithms we discussed in the first week are powerful algorithms capable of solving a wide variety of problems. However, to apply these algorithms you must first formulate the associated search problem (i.e., define states, transitions, start states, and goal states). Every time you want to solve a new problem, you are confronted with the task of formulating and programming that search problem from scratch.

In this set of notes, we will look at a general language for defining planning problems called PDDL (Planning Domain Definition Language). During the first week of class, we focused on how we solve search problems. Now, we focus on how we can represent knowledge about the world and reason about actions that change that world—a core problem in Knowledge Representation and Reasoning (KRR).

PDDL is the successor of STRIPS (Stanford Research Institute Problem Solver) from 1971. STRIPS was the primary planning system aboard Shakey the robot, one of the first autonomous robots capable of reasoning about its actions. A STRIPS instance consists of an initial state and goal state (much like our search problems) and a set of operators. Operators in this context can be thought of as actions an agent can complete. Operators are defined by preconditions (things that must be true before the action is executed) and effects (what changes the action causes in the world). PDDL generalizes STRIPS and includes additional features.[2]

2 Blocks-World

Blocks-World is a problem setting where labeled blocks are sitting on a table (or stacked on other blocks) and a robot arm is tasked with stacking the blocks into some goal configuration. The robot arm may only pick up one block at a time and can either place a block on another block or on the table. It is a relatively simple problem (i.e., it's a game that young children play), but serves an introduction to PDDL.

[1] These notes were compiled in conjunction with Professor Greenwald.

[2] PDDL allows negative predicates in preconditions (i.e., things that cannot be true before an action is executed), conditional effects, durative actions, and various other extensions.

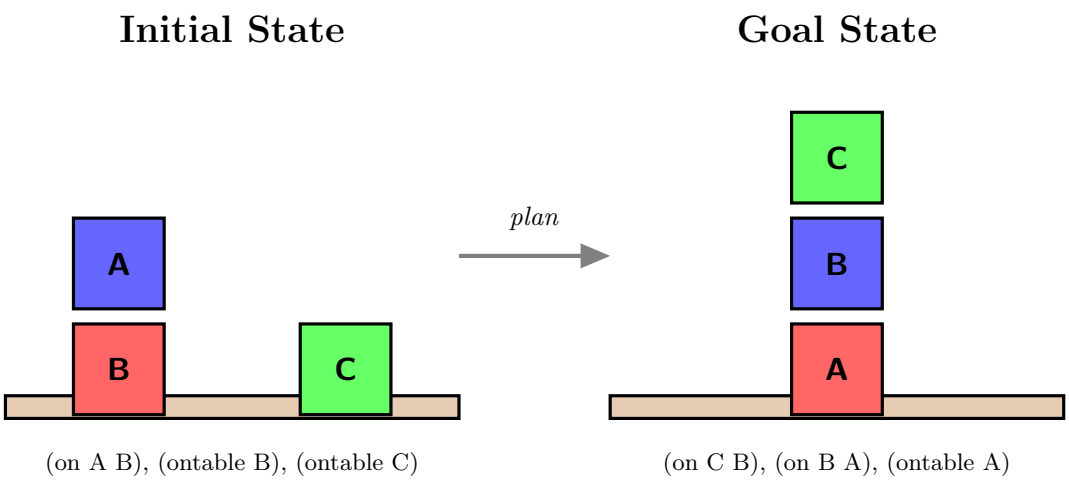

Figure 1: An example start and goal state for Blocks-World. The planning objective is to find a sequence of actions that transition from the start state to the goal state.

Figure 1 shows an example problem. In this case, a solution plan would be to pick up A and place it on the table, pick up B and place it on A, and finally pick up C and place it on B.

3 From First Order Logic to PDDL

In our previous lecture, we explored First Order Logic (FOL) as a powerful language for representing knowledge about the world. PDDL builds directly on FOL's foundation but specializes it for planning problems. Let's examine this connection more closely.

3.1 Shared Foundations

Both FOL and PDDL use the same fundamental building blocks:

- **Objects**: Entities in the world (blocks, robots, locations)
- **Predicates**: Properties and relationships (e.g., $\text{Clear}(x)$, $\text{On}(x, y)$)
- **Logical Formulas**: Combinations of predicates using $\wedge$, $\vee$, $\neg$
- **Quantifiers**: Universal ($\forall$) and existential ($\exists$) quantification

However, PDDL adapts these concepts specifically for planning:

1. **Closed World Assumption**: In FOL, if we don't know whether $P(x)$ is true, it could be either true or false. In PDDL, anything not explicitly stated as true is assumed false. This makes defining the initial starting conditions much easier, as not every false predicate needs to be explicitly stated.
2. **State-Based Representation**: PDDL views the world as a sequence of states, where each state is a complete assignment of truth values to all ground predicates (predicates with all variables replaced by objects).

3.2 From FOL Rules to PDDL Actions

Consider how we might express the ability to pick up a block in both systems:

In FOL:

$$\forall x\,(\text{Block}(x) \wedge \text{Clear}(x) \wedge \text{HandEmpty}() \Rightarrow \text{CanPickUp}(x))$$

$$\forall x\,(\text{PickUp}(x) \wedge \text{OnTable}(x) \Rightarrow \neg\text{OnTable}(x) \wedge \text{Holding}(x))$$

In PDDL:

```
(:action pick-up
  :parameters (?x - block)
  :precondition (and (clear ?x) (handempty) (ontable ?x))
  :effect (and (holding ?x) (not (ontable ?x)) (not (handempty)))
)
```

In PDDL, actions have specific preconditions and effects. The precondition of an action must be true before the action can be executed. The effect is what changes after the action is executed. Most conditions and effects and specified as a conjunction of predicates. Disjunctions are actually not allowed in effects in standard PDDL (the effect of an action can't have one effect OR another effect, it has to fully specify the effect of that action).

4 PDDL Syntax

PDDL uses two types of files: the *domain* file, which defines the predicates and actions available, and the *problem* file, which defines the initial and goal states for a specific instance.

4.1 The Domain File

To demonstrate the syntactical structure of PDDL, we will examine an example domain file. The domain file specifies things that are common to all instances of a given problem: the types of objects we might encounter, predicates that operate on these types, and the actions we may perform. We'll look at the classic "blocks-world" problem where a robot arm can pick up blocks and stack them.

```
(define (domain simple-blocks-world)            ;; Domain name
  (:requirements :strips :typing)               ;; STRIPS-style + typing
  (:types block)                                ;; Object types
  (:predicates                                  ;; State predicates
    (clear ?b - block)                          ;; No block on top of b
    (on ?b1 - block ?b2 - block)                ;; b1 is on b2
    (ontable ?b - block)                        ;; b is on the table
    (handempty)                                 ;; Robot hand is empty
    (holding ?b - block)                        ;; Robot holds block b
  )
  (:action pick-up                              ;; Pick up from table
    :parameters (?b - block)
    :precondition (and (clear ?b) (ontable ?b) (handempty))
    :effect (and (holding ?b) (not (ontable ?b)) (not (handempty)))
  )
  (:action put-down                             ;; Place on table
    :parameters (?b - block)
    :precondition (holding ?b)
    :effect (and (ontable ?b) (not (holding ?b)) (clear ?b) (handempty))
  )
```

```
  (:action stack                                ;; Stack b1 onto b2
    :parameters (?b1 - block ?b2 - block)
    :precondition (and (holding ?b1) (clear ?b2))
    :effect (and (not (holding ?b1)) (not (clear ?b2))
                 (clear ?b1) (on ?b1 ?b2) (handempty))
  )
  (:action unstack                              ;; Pick up b1 from b2
    :parameters (?b1 - block ?b2 - block)
    :precondition (and (on ?b1 ?b2) (clear ?b1) (handempty))
    :effect (and (holding ?b1) (clear ?b2)
                 (not (on ?b1 ?b2)) (not (handempty)))
  )
)
```

The syntax of PDDL is inspired by Lisp, using parentheses for scoping and grouping. Key syntactic elements:

- Expressions beginning with a colon are keywords (e.g., :action, :predicates)
- Expressions beginning with a question mark (e.g., ?b) are variables
- Expressions in parentheses without special prefixes are predicates or compound expressions
- Atoms (simple identifiers) represent constants or object names

4.2 The Problem File

The problem file defines a single instance by specifying the starting state and goal state. Here's an example with three blocks:

```
(define (problem block-problem-1)
  (:domain simple-blocks-world)
  (:objects A B C - block)
  (:init
    (handempty)
    (clear A) (clear C)
    (on A B)
    (ontable B) (ontable C))
  (:goal (and (on C B) (on B A)))
)
```

This represents: - **Initial state**: A is on B, B and C are on the table - **Goal state**: Tower with C on B on A

5 Quantifiers and Advanced Features in PDDL

While basic PDDL uses ground predicates, advanced PDDL supports quantifiers similar to FOL, enabling more expressive problem definitions.

5.1 Universal Quantification in Preconditions

We can require that all objects of a certain type satisfy a condition:

```
(:action clear-table
  :parameters ()
  :precondition (forall (?b - block) (not (ontable ?b)))
  :effect (table-cleared)
)
```

This action can only execute when no blocks are on the table.

5.2 Existential Quantification in Preconditions

We can check if at least one object satisfies a condition:

```
(:action start-stacking
  :parameters ()
  :precondition (exists (?b - block) (and (clear ?b) (ontable ?b)))
  :effect (ready-to-stack)
)
```

5.3 Conditional Effects

PDDL can express effects that only occur when certain conditions are met:

```
(:action careful-stack
  :parameters (?b1 - block ?b2 - block)
  :precondition (holding ?b1)
  :effect (and
    (when (fragile ?b2) (careful-mode))
    (on ?b1 ?b2)
    (not (holding ?b1))))
)
```

6 From PDDL to Search Problems

PDDL descriptions can be automatically converted to search problems:

- **States**: Complete assignments of truth values to all ground predicates
- **Initial State**: Specified in the problem file's :init section
- **Goal Test**: Check if all predicates in :goal are satisfied
- **Actions**: Applicable when preconditions are satisfied in current state
- **Transition Function**: Apply effects to generate successor states

This connection allows us to apply all our search algorithms (BFS, DFS, A*, etc.) to PDDL problems.

7 PDDL Solvers and Heuristics

One major advantage of PDDL is standardization—any PDDL solver can work with any valid PDDL problem. Modern solvers like Fast Downward use sophisticated techniques:

7.1 Domain-Independent Heuristics

Unlike domain-specific heuristics (like Manhattan distance for sliding puzzles), PDDL solvers must work across all domains. Common approaches include:

- **Delete Relaxation**: Ignore negative effects of actions. If achieving the goal requires making P false but later making it true again, the relaxed problem ignores the first requirement.
- **Landmark Detection**: Identify predicates that must be true at some point in any valid plan. For example, in blocks-world, to place A on B, we must at some point be holding A.
- **Pattern Databases**: Precompute distances to goal for abstract versions of the problem.

7.2 The Fast-Forward Heuristic

The FF heuristic estimates cost-to-goal by solving a relaxed problem: 1. Remove all negative effects from actions 2. Find a plan in this easier problem (often using a graph-based approach) 3. Use the length of this relaxed plan as the heuristic value

While the relaxed plan isn't executable in the real problem, its length provides a lower bound on the actual distance to the goal.