Assignment 5: PDDL Planner

Due: July 13, 2023 at 11:59PM

1 Assignment 5: PDDL Planner Specification

1.1 Overview

For this assignment, you will write a state-space progression planner for sequential PDDL domains. Given a domain description and a problem instance (an initial state and a goal specification), your program will ground all actions and produce applicable actions for a domain description.

In the next assignment, you will extend your code to return a sequence of actions that will achieve the goal. You will use a variant of A^* to search the state space, with several heuristics.

Find the starter files and repository at `https://classroom.github.com/a/PXOKtlvr`.

1.2 Goals

1. Practice designing data structures to store planning domain information
2. Implement a grounding algorithm to generate concrete actions
3. Implement an action search which can determine which actions are applicable from a state

1.3 Input Descriptions

Your program should read both the domain and problem instance from standard input, in that order.

The domain description specifies in order: the predicates of the domain, a set of constants that apply to any problem instance, the number of actions, and the action definitions. Each part of an action definition is on its own line in the following order: positive preconditions, negative preconditions, negative effects, positive effects. Predicates in preconditions and effects can take constants (upper case) or variables (lower case) as arguments. Comment lines start with #. For example:

```
# switch world

predicates: On(x) Off(x)
constants: A B
2 actions

TurnOn switch
pre: Off(switch)
preneg:
del: Off(switch)
add: On(switch)

TurnOff switch
pre: On(switch)
preneg:
del: On(switch)
add: Off(switch)

initial: On(A) Off(B)
goal: Off(A) On(B)
```

The problem description on the last two lines consists of the initial state on one line, and the goal specification on the last line.

This is not the formal PDDL language, but it is clear to read and parse. See the `inputs` directory for examples that follow the domain/instance description. Assume that all inputs are correctly formatted.

1.4 Output

Your program should output as follows:

Printed to standard output: a list of all applicable actions from the start state, one per line.

Example output for `switches.in` (your action order may not match exactly):

```
TurnOn B
TurnOff A
```

If there are no applicable actions, print `No actions` instead of an action list.

1.5 Execution

Write your code in one of two languages: Java or Python. You should name your source file that has the program's main method `PDDL`. One of the following options should invoke your program:

```
> python PDDL.py < inputs/switches.in
```

(Assuming the code has been compiled)

```
> java PDDL < inputs/switches.in
```

2 Design Ideas

There are three important parts in this assignment: parsing, grounding, and searching. You can test each part before moving onto the next part, to confirm that it is working.

2.1 Parsing

There are several important entities required to hold input information. A domain description holds all of the possible predicates and actions. A predicate holds a list of terms. An action holds the preconditions (both positive and negative) and postconditions (both positive and negative). A state consists of all predicates that are true at the moment.

You may find it helpful for this and future sections to create or modify planning domains to test parts of your code.

2.2 Grounding

After parsing the input, all actions and constants are known. Some actions might have variables, meaning that any constant can be substituted in. Rather than waiting until a state-space search is underway to perform substitutions, it is efficient to make a grounded action list ahead of time.

In order to generate this list, we can try to substitute each constant for each variable:

```
GroundActions(actions)
1:  Q ← all actions with a variable
2:  groundActions ← ∅
3:  while Q ≠ ∅ do
4:      a ← Q.pop
5:      v ← first variable of a
6:      for all constants c in domain do
7:          g ← a.substitute(v, c)
8:          if g still has variables then
9:              Q.push(g)
10:         else
11:             groundActions.add(g)
12: return groundActions
```

In some domains this may produce impossible actions: actions in which the same predicate shows up in the preconditions and negated preconditions, or a predicate is both added and deleted by the action. It is more efficient to filter these impossible actions out of your action list before searching, but not necessary to get correct results.

2.3 Searching

In order to find applicable actions, you will have to search through all the grounded actions for actions where the preconditions are true and the prenegations are not true.

3 Submission

Submit the code for your solution along with a writeup that answers the following questions

1. Describe choices you made in your code that you feel are important. Mention any specific aspects of your implementation that might be interesting as I evaluate your program.
2. How many grounded actions do you expect, as a function of the input size?
3. What suggestions do you have for improving this assignment?

For submission, commit/push your updates to your github repository. I automatically have access to the repository, so if you see your updated files on github, I have access to them too.

4 Evaluation

- 40%: Reads the input and partially tokenizes the input lines
- 70%: Fully parse the input and store actions and predicates in data structures
- 85%: Generate all grounded actions
- 95%: Search over grounded actions to find applicable ones from the start state
- 100%: Print the applicable actions