Assignment 6: PDDL Planner
Due: July 24, 2023 at 11:59PM

1 Assignment 6: 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 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. This assignment extends the previous assignment.

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

1.2 Weighted A*

The weighted A* state-space search algorithm is just A* with one modification: the heuristic function is given a constant weight w in the calculation of each node's cost. wA* uses $f(x) = g(x) + w \cdot h(x)$, with weights typically being ≥ 1. This may lead to suboptimal solutions, but often reduces the number of generated nodes in the search. There are more details in the textbook on page 91.

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 is on the last line. The problem description is also included above.

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.

In addition to the domain input, your program should accept two command-line arguments:

weight: an *integer* used as the weight to use for wA*.

heuristic: one of four heuristic functions:

- `h0`: $h(n) = 0$
- `hlits`: number of goal literals that are false
- `hmax`: the h^{max} heuristic
- `hsum`: the h^{sum} heuristic

1.4 Output

Your program should output as follows:

Printed to standard output: a list of instantiated actions with each action labeled with the time it is to be executed (starting at time 0), followed by the number of search nodes expanded and generated.

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

```
0 TurnOff A
1 TurnOn B
9 nodes generated
4 nodes expanded
```

If a goal is impossible (for example a goal of `Off(A)` and `On(A)`), print `Impossible goal` instead of a solution.

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 2 hsum < inputs/switches.in
```

(Assuming the code has been compiled)

```
> java PDDL 2 hsum < inputs/switches.in
```

2 Design Ideas

There are four important parts in this assignment: parsing, grounding, searching, and heuristics. You should have completed parsing and grounding already.

2.1 Searching

The A* algorithm from Assignment 1 should work the same for this assignment, with the addition of the w constant in the $f(n)$ calculation. States should hold a set of all predicates that are true, and nodes should hold a state, f-value, and parent state (and the action necessary to reach that state).

In order to generate children of a node, loop through all actions, decide which ones are applicable in that node's state, and generate child nodes with the updated state. The updated state should have all of the appropriate predicates added or deleted from its parent.

2.2 Heuristics

The two complicated heuristics, h^{sum} and h^{max} should follow the h^+ algorithm from the lecture slides. The idea with these heuristics is to perform a relaxed sort of search that is much faster to compute than the full plan. You should call the heuristic function every time you generate a node.

You might include a t field in predicates that is only used for these heuristic functions.

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. Which of the four heuristic functions are admissible? Why?
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%: Populate nodes for search algorithm
- +20%: Generate and expand nodes in wA*
- +10%: Produce correct plans with `h0` heuristic
- +10%: Produce correct plans with `hlits` heuristic
- +5%: Print plans and statistics correctly
- +15%: Produce correct plans with h^{sum} and h^{max} heuristics