Problem Set 4 - Task Planning Heuristics

STRIPS Planners and PDDL will be covered in lecture on Thursday 11/1. The slides will be posted on Piazza shortly after lecture.

STRIPS (PDDL) Planner Search Code Overview

The planner code defines a simple planner. The key data structures are:

- **states** are represented by a Python set of propositions (assertions) each of which is a tuple of the form (type, arg1, arg2, ...), for example, `('free', 'partA')`.
- **actions** are represented as instances of the `Operator` class (in `strips.py`). An action is specified by a name, a list of preconditions (facts that must be present in the state for the action to be applicable), add effects (facts that are added to the current state), and delete effects (facts that are deleted from the current state). An action `op` (an instance of `Operator`) has two key methods: `op.applicable(state)` checks whther the action is applicable in a state and `op.apply(state)` produces the new state specified by the action.
- **tasks** are represented by an instance of the `Task` class (in `strips.py`). A task specifies the initial state, a list of all the possible facts, a list of goal facts (which must be present in the final state), and a list of action instances. The Python class `Task` has several essential methods for defining the search problem; you should look at the definition.

The file `pddl_main.py` is meant to be called as follows:

```
python pddl_main.py directoryName fileName
```

where `directoryName` indicates where the `domain.pddl` and problem files can be found. `filename` is the name of a problem file (without any extension). For example:

```
python main.py prodigy-bw bw-simple
```

The planning code we gave you should be functional.

The implementation we have given you uses `astar_search` with the trivial heuristic that always returns 0. Try a couple of problems problems in the `prodigy-bw` directory. Use an `InstrumentedProblem` to keep track of how many expansions are done.

Your Job: Heuristics Implementation

These are the tasks you must complete for pset 4.

1. Implement `h_FF(node)`

   The slides from Thursday's lecture describe these heuristics. Note that the `h_max` and `h_add` heuristics only need the "forward" pass described in the slide "Computing hFF: Relaxed Planning Graphs (RPG)" (slide 47/78); to compute `h_FF` you also need the backward pass described in "Computing hFF: Extracting a Relaxed Plan" (slide 48/78). Below is an abstract numerical example and examples of computing these heuristics for the logistics domain in the lecture slides.

2. Make sure that you debug the heuristic computation separately, before you try to run it on a full planning problem. In the `pddl_main.py` file, after you create an instance of the planning problem (a subclass of `search.Problem`), you can call the heuristic function defined there:

   ```
   prob = PlanProblem(task)
   # In search.py, heuristics takes an instance of search.Node


   ffv = prob.h(search.Node(task.initial_state))
   print 'initial h_FF', ffv
   ```

Here are the values we obtained for `h_FF` in the initial state (and the final plan length) of the following problems (from the `prodigy-bw` domain):

```
problem    h_ff plan
bw-simple   2   2
bw-sussman  5   6
bw-large-a 12  12
bw-large-b 16  18
bw-large-c 26  28
bw-large-d 34  36
```

You should also have your implementation print the facts and actions at each of the levels in the forward pass as well as the actions selected in the backward pass. Check these results by hand on simple cases, such as `bw-simple.pddl`. Make sure that your results match what you expect.

3. Compare the performance of the planner using the simple heuristic `h_G` (counts unsatisfied goals) versus using the FF heuristic on some examples from the `prodigy-bw` and some examples from `logistics-strips`.

```
    def h_G(self, node):
        return len(self.task.goals - node.state)
```

4. (Optional) Implement Helpful Actions as described in class and in the slides and report the impact in performance.
5. Submit your file `pddl_main.py` with your heuristic implementation. **Include a summary of your testing in your file as comments**.

Numerical Example

Consider the following STRIPS actions.

Name	Pre	Add	Del
A	m	n, o	
B	m, o	p	m
C	p	m	p
D	n, o	p	o

Assume the current state is $s = \{m\}$ and the goal is $g = \{m, n, o, p\}$.

1. What is the value of $h^*(s)$ (the actual shortest plan length)?
   Show/hide answer 3
2. What is the value of $h_{max}(s)$?
   Show/hide answer 2
3. What is the value of $h_{add}(s)$?
   Show/hide answer 4
4. What is the value of $h_{FF}(s)$?
   Show/hide answer 2

   Make sure that you understand why the values are as given.

Heuristics in Logistics domain

The **relaxed** Logistics domain is described in the slide "A Relaxed Plan for "Logistics" (slide 24/78).

Initial state I is:

```
(truck A), (pack C)
```

But, also includes some "static" facts that don't change:

```
(road A B), (road B A), (road B C), (road C B), (road C D), (road D C)
```

The goal G is:

```
(truck A), (pack D)
```

The actions are:

```
    (:action drive
        :parameters (?x ?y)

        :precondition (and (road ?x ?y) (truck ?x))
        :effects (truck ?y))
    (:action load
        :parameters (?x)
        :precondition (and (truck ?x) (pack ?x))
        :effects (pack in-truck))
    (:action unload
        :parameters (?x)
        :precondition (and (truck ?x) (pack in-truck))
        :effects (pack ?x))
```

Let's compute `h_FF` at the initial state I. We're going to define alternating sets of facts (`Fj`) and actions (`Ai`). The fact have indices `j` from 0 to M, while the actions have indices `i` from 0 to $M-1$. Below we do not show the "static" facts which would also go in `F0`

Note that the "reverse" actions (for example, `(drive B A)` in `A1`) don't add new facts in this example (recall that `Fj` are sets).

- `F0={(truck A), (pack C)}` : facts in the initial state
- `A0={(drive A B)}` : actions all of whose preconditions are in F0
- `F1= F0 + {(truck B)}` : F0 + facts added by actions in A0
- `A1={(drive B C), (drive B A)}` : actions all of whose preconditions are in F1
- `F2= F1 + {(truck C)}` : F1 + facts added by actions in A1
- `A2={(drive C D), (drive C B), load(C)}` : actions all of whose preconditions are in F2
- `F3= F2 + {(truck D), (pack in-truck)}` : F2 + facts added by actions in A2
- `A3={(drive D C), (unload A), (unload B), (unload D) }` : actions all of whose preconditions are in F
- `F4= F3 + {(pack A), (pack B), (pack D)}` : F3 + facts added by actions in A3

At this point, all of the goal facts are present in `F4`, so we can stop. The value of M in this example is 4.

Note that as part of this process, we want to build a dictionary that will tell us the `level` (the index) at which a fact first appears in `F`. Given these levels, we can easily compute `h_max` as the max level of any of the goals and `h_add` as the sum of the levels of the goals.

To compute `h_FF` we need a pass backwards in which we build a **set** of actions (call it `selected`) where the actions in this set achieve all the goals and all of whose pre-conditions are added by actions in the set (or are in the initial state).

We now define `Gt` where the index `t` goes from 0 to M. These sets will be initialized with the goals at the corresponding levels. In our examples:

```
G0={(truck A)}, G1={}, G2={}, G3={}, G4={(pack D)}
```

Now, we iterate over these sets starting at `G4`. For each goal g in set `Gt`, we find some action a in `A(t-1)` that adds g; we add it to the `selected` set. We then add each precondition p of action a to the `G` set corresponding to the level of p. When we're done with this pass, we return the length of the `selected` set. Below, we show the active `G` sets and the `selected` set as `t` goes from 4 to 1.

- at start of `t=4 loop`

```
G0={(truck A)}, G1={}, G2={}, G3={}, G4={(pack D)} selected={}
```

- at end of `t=4 loop`

```
G0={(truck A)}, G1={}, G2={}, G3={(truck D), (pack in-truck)}, G4={(pack D)} selected={(
```

- at end of `t=3 loop` (ignoring `G4`)

```
G0={(truck A), (pack C)}, G1={}, G2={(truck C)}, G3={(truck D), (pack in-truck)} selecte
```

- at end of `t=2 loop` (ignoring `G3,G4`)

```
G0={(truck A), (pack C)}, G1={(truck B)}, G2={(truck C)} selected={(drive B C) (drive C I
```

- at end of `t=1 loop` (ignoring `G2, G3,G4`)

```
G0={(truck A), (pack C)}, G1={(truck B)} selected={(drive A B) (drive B C) (drive C D) (
```

At this point, we are done and the value of `h_FF` for this state is the length of selected, that is, 5. Note that the actual optimal plan length is 8 (see slide 36/78).

It is much harder to see how `h_FF` can overestimate the plan length. Assume that there are cities A through J and the truck starts in city E. Let's assume that there are two types of packages (red and green) and there is a red package at city A and another red package at city J; there is a green package at city H. The goal is to load a red and a green package into the truck. The optimal plan in to keep going right and pick up the packages at H and J (5 drives and 2 loads). But, `h_FF` would instead select actions that go right to H and that go left to A. We would reach A and H by level 4 of the RPG, so we would go no farther (so we would never reach J). We would select actions to reach A and (different) actions to reach H; in the optimal plan, the actions to reach H also help us reach J.

Testing

To run the Gradescope tests, run `test_pset_4.py` in a docker bash container. For example, if you are on Mac, run

```
docker/docker_run_bash_mac.sh drake-20181030 pset_4/
python test_pset_4.py results.json
```

[Back to Pset 4 Main (../pset_4.html)](../pset_4.html)