Optop Demo: Dealing with Autonomous Processes

Drew V. McDermott
Yale University
Computer Science Department
`drew.mcdermott@yale.edu`

Optop is an evolving planning system based on *estimated regression,* a technique for guiding best-first search through situation space using estimates of the difficulty of completing a plan with a given prefix. The estimates are obtained by searching backward from the goal using a relaxed search space in which literals are treated in isolation and deletion effects of actions are ignored.

In this demo, I will show Optop solving problems that involve *autonomous processes,* defined as processes that occur and have continuous effects whenever a condition is satisfied, independently of what actions are taken by the target agent of the planner. The changes to a classical planner to get it to reason about processes are described in my paper presented at ICAPS 03, "Reasoning about Autonomous Processes in an Estimated-Regression Planner." In the demo, I will show how processes are specified, and how well the planner works when it has to reason about processes.

I will use two sample domains: a very simple example involving filling a tub, and a more complex one involving convoys traveling over a road network. The tub domain is shown in figure 1. We use an extension of the PDDL notation, in which `:process` definitions are allowed by analogy with `:action` definitions. The `:process` construct differs from the familiar `:action` definition in two ways:

1. The `:precondition` field has become the `:condition` field. The process is active whenever the condition is true.
2. The `:effect` field has split into three fields, `:effect`, `:start-effect`, and `:stop-effect`. The `:effect` is true whenever the process is active. Such continuous effects can include `derivative` assertions, which state the rates of change of fluents. The `:start-effect` field states effects that become true whenever the process switches from inactive status to active. Similarly, the `:stop-effect` field states effects that become true when the process switches from active to inactive. Like traditional effects, these stay true until altered by subsequent actions or process switches.

Note that the agent's actions have no direct impact on the status of a process. Of course, the agent can make a process active or inactive by altering the status of its condition.

A problem in the "tub" domain might involve the goal of floating your boat. A solution would require one to turn the tub on, wait for it to fill, then put the boat in. If there are multiple tubs, the optimal plan is the one that uses the tub that fills up the fastest, assuming that one wants to minimize time. One might want to minimize some other metric, such as the weighted difference of time and tub surface area.

A solution to a one-boat problem in this domain will look like

(turn-on u); (wait); (floatit the-boat)

where u is the chosen tub, and `the-boat` is the given boat. If we want to require that no overflow occur, the plan will also have a `turn-off` step.

The demo will show the following points:

- How fast the planner runs as problems scale up.
- With various debugging switches turned on, exactly what the planner is doing at various points.
- How processes are simulated; how the system decides what point to advance time to.
- The role of the "plausible projector" in predicting what will happen when a plan is executed in the presence of autonomous processes, and, in particular, what the value of an objective function will be at the end of execution.
- Some open problems whose resolution will guide the evolution of the planner.

```
(define (domain tub)
   (:requirements :fluents :processes)

   (:types Boat Tub - Obj)

   (:functions (volume ?tub - Tub)
               (faucet-rate ?tub - Tub) - Float
               (water-in ?tub - Tub) - (Fluent Float))

   (:predicates (floating ?b - Boat ?tub - Tub)
                (faucet-on ?tub - Tub)
                (overflowing ?tub - Tub))

   (:action (floatit ?b - Boat ?tub - Tub)
      :precondition (=~ (water-in ?tub) (volume ?tub))
      :effect (floating ?b ?tub))

   (:action (turn-on ?tub - Tub)
      :effect (faucet-on ?tub))

   (:action (turn-off ?tub - Tub)
      :effect (not (faucet-on ?tub)))

   (:process (filling ?tub - Tub)
      :condition (faucet-on ?tub)
      :effect
         (and (when (< (water-in ?tub) (volume ?tub))
                (derivative (water-in ?tub)
                            (faucet-rate ?tub)))
              (when (>= (water-in ?tub) (volume ?tub))
                 (and (derivative (water-in ?tub) 0.0)
                      (overflowing ?tub))))))

(define (problem tub-prob-1)
   (:domain tub)
   (:objects tub1 - Tub my-boat - Boat)
   (:facts (= (faucet-rate tub1) 1.0)
           (= (volume tub1) 10.0))
   (:init (current-value (water-in tub1) 0.0))
   (:goal (exists (tub - Tub)
             (and (floating my-boat tub)
                  (not (overflowing tub)))))
   (:metric minimize (total-time)))
```

Figure 1: Simple domain involving processes