6.S078 Planning Algorithms, Fall 2013

October 21, 2013

Assignment 5 (due Wed. Nov 13 before class)

Implement a planner for STRIPS-like domains. I've given you a simple skeleton for such a planner in the file `planner.py`; feel free to use it or not.

- Finish the planner inmplementation. Implement a domain analogous to the PDDL domain for Assignment 4. Try it on a very simple example.
- Experiment with best-first for your search. Use a very simple heuristic, such as the count of the number of literals not in the goal. Compare the plan lengths found by the search to the optimal length plan.
- Implement the FF heuristic and document the performance (time and plan length) on some more challenging examples.
- Implement Helpful Actions as described in the FF papers and report the change in performance.
- You should expect that your planner will be substantially slower than the highly optimized planners used in the planning competitions (Blackbox, FF), but you should try to compare them on some relatively simple problems.
- Please send email to ask for any clarifications.

The file `planner.py` provides a skeleton for a very simple symbolic planner. The key data structures are:

- **states** are represented by set of propositions (assertions) each of which is a tuple of the form (type, arg1, arg2, ...), for example, `('free', 'partA')`. The Python class `State` represents a state and supports a couple of methods: `check` to test if some assertions are part of the state and `addDelete` to create a new state which deletes and then adds the specified assertions. Note that the `addDelete` method takes an optional argument that allows you to skip doing the deletes, which may be useful in implementing the 'ignore delete list' heuristic. Note also that the class is defined so equality and hashing work properly.
- **actions** are represented as functions that are given a state and construct a new state by adding and deleting assertions from the input state, assuming that a set of preconditions are satisfied, otherwise, they return the input state. The Python class `Action` encodes an action.

Note that actions take a list of args that specify the action instance, e.g. `Drill(['partA', vertical'])`. The key method of an action instance is `resultStateAndCost` that takes a state and returns a pair of state and cost of the action. In our world, we have a set of 'entities', such as PARTS, MACHINES, etc., the set of all possible action instances are created from these items (see the example in the file).

- **planning problems** are represented by an initial state, a list of goal assertions (which must be present in the final state) and a list of action instaces. The Python class `PlanProblem` collects the specification of the problem and defines the `findPlan` method, which provides the interface to the search function (which you provide).