Cha Li – chavli@cs.pitt.edu
11.21.2011
CS2710 – Assignment 4: Partial Order Planner

Summary

1. Backwards Search

   I chose to do backwards search for this assignment, starting with the goal propositions (initial open conditions) and searching through all possible steps for actions that satisfy the open conditions. The following sections will outline how certain tasks are accomplished and include code snippets that illustrate the functionality. For the sake of brevity, the code will not contain any comments and will only be lines that directly address the task (all variables assumed to be initialized). Please refer to the source file (assign4code.py) for more details and comments. If necessary, pieces of raw output will also be provided.

   - Detecting Steps, and New Open Conditions

     The first step is to chose an open-condition to address, my code chooses the open-condition randomly. Once that is chosen, Co , the code searches through `adds` for steps that have the chosen open-condition in their add list. For each step found, a new Plan is created that holds the results of adding he new step (every plan that is created for a given open-condition is returned by successor). Additionally, a new ordering condition is added that links the new step, with the step the open-condition is associated with. A new causal link is also added that includes the new step, the open-condition, and the open-condition's step.

     Once the new step is included, all of the new steps preconditions are added to the new Plan's open-conditions (which might also include unresolved open-conditions from the parent Plan).

```
for step, props  in adds.iteritems():
  if (open_prop in props):
    child = Plan();

    child.steps = list(plan.steps);
    if not child.stepExists(step):
      child.steps.append(str(node.depth) + DIVIDER + step);
      print "[STATUS] Added Step: ", step;

    new_ordercon = (step, end_step);
    child.ordercons = list(plan.ordercons);
    if new_ordercon not in child.ordercons:
      child.ordercons.append(new_ordercon)
      print "[STATUS] Added New Order Condition: ", new_ordercon;

    new_causal= (step, open_prop, end_step);
    child.causallinks = list(plan.causallinks);
    if new_causal not in child.causallinks:
      child.causallinks.append(new_causal);
      print "[STATUS] Added New Causal Link: ", new_causal;
```

   - Detecting Threats

     Two methods are used to detect any new threats created by adding a new step:
     - Check if any of the existing steps in the plan have delete lists that delete propositions in the new steps preconditions list
     - Check if any of the elements in the new steps delete list delete propositions in the current set of clausal links

     Both of these conditions are checked by just using for loops iterating over their respective lists. The following code pieces illustrate the about two methods respectively. Any threats that are detected are added to the new Plan's threat list. Unresolved threats from the parent plan are also carried over to the new plan. Duplicate threats are ignored.

```
for previous_step in plan.steps:
  previous_step = previous_step[string.find(previous_step, DIVIDER) + 1  :];
  delete_list = deletes[previous_step]
  if open_prop in delete_list:
    new_threat = (new_causal, previous_step);
    if new_threat not in child.threats:
      child.threats.append(new_threat);
      num_threats += 1;

for current_causal in plan.causallinks:
  if current_causal[PROP_CL] in deletes[step]:
    new_threat = (current_causal, step);
    if new_threat not in child.threats:
      child.threats.append(new_threat);
      num_threats += 1;
```

2. Resolving Flaws

- Open Conditions

  Resolving open-conditions was covered briefly in the first section “Detecting Steps and new Open Conditions” but this section will go into a little more detail. As stated earlier, the first step is to choose a random open-condition in the current plan and find any steps that contain the open-condition as an add condition. New plans are created for each new step found and the original open-condition removed from them, however, more can be done. Each new plan has a new step which has its own set of add conditions, these conditions can be used to resolve additional open-conditions in the new plan. For example, if the original plan has open-conditions A, B, C and D and a step S is found that resolves A a new plan is created with S and A removed. However, if S has C in its add list, then C can also be removed from the new plan. The following code is just what follows the code presented in “Detecting Steps and new Open Conditions”:

  ```
  #1. remove any resolved open conditions
  #2. add new open conditions (the preconditions of added step)
  child.openconds = list(plan.openconds);
  child.openconds.remove((open_prop, end_step));
  print "[STATUS] Removing Open Condition: ", (open_prop, end_step);

  #add pre conditions of added step to open conditions
  for precond in preconds[step]:
    child.openconds.append((precond, step));

  #see if the adds of the added step remove any other open conditions
  for opencond in child.openconds:
    if opencond[PROP_OC] in adds[step]:
      child.openconds.remove(opencond);
  ```

- Threats

  A random threat is chosen out of the current plan's threats to be resolved. The resolution of a threat results in a new plan with that threat removed (and with any new order conditions included). Given a threat T = ((step1, prop, step2), threatening_step), it can be resolved using either demotion or promotion. The method chosen depends on which option creates a consistent ordering (see below), and if any of the steps are init or goal. Demotion states that threatening_step comes before step1, and promotion states that threatening_step comes after step2. Both are enforced by adding one more ordering condition to the new plan. Another constraint for the ordering, aside from consistency, is that nothing can come before init, and nothing can come after goal. The following code does what is described in this section:

```
if step != "goal" and pre_step != "init" and
      orderConsistent(order + [(step, pre_step)]):

      child_threats.remove(threat);
      new_ordercon = (step, clause[STEP1_CL])
      if new_ordercon not in child.ordercons:
        child.ordercons.append(new_ordercon);
      child.threats = child_threats;
      children.append(child);
      print "[STATUS] Threat resolved through demotion:", new_ordercon

elif step != "init" and post_step != "goal" and
      orderConsistent(order + [(post_step, step)]):

      child_threats.remove(threat);
      new_ordercon = (clause[STEP2_CL], step);
      if new_ordercon not in child.ordercons:
        child.ordercons.append(new_ordercon);
      child.threats = child_threats;
      children.append(child);
      print "[STATUS] Threat resolved through promotion:", new_ordercon
```

- Ordering Consistency

  The foremost condition that must hold when promoting or demoting steps to resolve threats is ordering consistency. Ordering consistency just means that for every ordering condition s1 > s2, there can't be another ordering condition that states s2 > s1 (there can't be any cycles). I enforce this with a method called orderConsistency which simply builds an ordered list based on the current ordering conditions. Elements of this list are allowed to be swapped if there isn't an ordering condition explicitly saying they can't be swapped.

  For example, given the initial ordering constraints: (1 > 3) (2 > 3)
  you get a list [1, 2, 3]
  if another constraint is added: (2 > 1)
  then 1 and 2 can be swapped since there is no condition explicitly saying (1 > 2)
  so the final ordering [2, 1, 3] still satisfies all constraints.

  However, if (1 > 2) were part of the initial ordering, (2 > 1) would cause the list to become inconsistent. I'm not going to include the code for this method here since it's kind of long, but if you want to see it it can be found starting on line 334 of assign4code.py.

3. Example Output

The following output is what one iteration of the for loops (presented by the code in “Detecting Steps and new Open Conditions”) might produce:

```
plan5 -----
RESOLVED THREAT: consumer of clause(wash-floor) ordered before wash-floor
steps: ['init', 'goal', '0.wash-floor', '1.sweep']
causal links:
  (wash-floor < floor-clean < goal)
  (sweep < floor-not-dusty < goal)
  (init < floor-dirty < wash-floor)
ordering constraints:
  (wash-floor < goal)
  (sweep < goal)
  (init < wash-floor)
open conditions:
  (furniture-clean, goal)
parent: plan plan4
-------------
[STATUS] Working on open condition:  ('furniture-clean', 'goal')
[STATUS] Added Step:  dust
[STATUS] Added New Order Condition:  ('dust', 'goal')
[STATUS] Added New Causal Link:  ('dust', 'furniture-clean', 'goal')
[STATUS] Removing Open Condition:  ('furniture-clean', 'goal')
[STATUS] Found 1 threats
=========
Len of fringe: 1
fringe[0].state: plan6 ---
```

4. Custom Scenario

The custom scenario can be executed with: `python assign4code.py 3`
I created a scenario that outlines the steps needed to complete this assignment. The following are the start and goal conditions. The bulk of the description (add lists, delete lists, preconditions) would take up too much space so I left them out of this report. (There are 10 steps and many propositions, excluding init and goal)

```
initprops3 = ["hungry", "tired", "not-at-home", "use-bathroom"]
goalprops3 = ["have-report", "code-compressed", "not-at-home"]
```

5. Planning Results

The following outputs are the final plans under what was displayed as the Result by the python script. The full print out of the results are really long so they aren't included in their entirety.

- Cleaning

```
plan9 -----
RESOLVED THREAT: consumer of clause(dust) ordered before dust
steps: ['init', 'goal', '0.wash-floor', '1.sweep', '4.dust']
causal links:
  (wash-floor < floor-clean < goal)
  (sweep < floor-not-dusty < goal)
  (init < floor-dirty < wash-floor)
  (dust < furniture-clean < goal)
  (init < furniture-dusty < dust)
ordering constraints:
  (wash-floor < goal)
  (sweep < goal)
  (init < wash-floor)
  (dust < goal)
  (dust < sweep)
  (init < dust)
```

- Flat Tire

```
plan10 -----
RESOLVED THREAT: consumer of clause(putonSpareAxle) ordered before
putonSpareAxle
steps: ['init', 'goal', '0.putonSpareAxle', '1.removeFlatAxle',
'4.removeSpareTrunk']
causal links:
  (putonSpareAxle < at(spare,axle) < goal)
  (removeFlatAxle < clear(axle) < putonSpareAxle)
  (init < at(flat,axle) < removeFlatAxle)
  (removeSpareTrunk < at(spare,ground) < putonSpareAxle)
  (init < at(spare,trunk) < removeSpareTrunk)
ordering constraints:
  (putonSpareAxle < goal)
  (removeFlatAxle < putonSpareAxle)
  (init < removeFlatAxle)
  (removeSpareTrunk < putonSpareAxle)
  (init < removeSpareTrunk)
```

- Finishing this Assignment (custom)

```
plan43 -----
RESOLVED THREAT: consumer of clause(useBathroom) ordered before useBathroom
steps: ['init', 'goal', '0.compressCode', '1.writeReport', '2.debugCode',
'3.writeCode', '4.drinkCoffee', '5.eatFood', '12.useBathroom']
causal links:
  (compressCode < code-compressed < goal)
  (writeReport < have-report < goal)
  (debugCode < not-buggy-code < compressCode)
  (writeCode < buggy-code < debugCode)
  (drinkCoffee < not-tired < writeCode)
  (eatFood < not-hungry < writeCode)
  (init < not-at-home < drinkCoffee)
  (init < tired < drinkCoffee)
  (useBathroom < not-use-bathroom < debugCode)
  (init < use-bathroom < useBathroom)
ordering constraints:
  (compressCode < goal)
  (writeReport < goal)
  (debugCode < compressCode)
  (writeCode < debugCode)
  (drinkCoffee < writeCode)
  (eatFood < writeCode)
  (init < drinkCoffee)
  (sleepInBed < eatFood)
  (playSkyrim < drinkCoffee)
  (useBathroom < debugCode)
  (init < useBathroom)
  (playSkyrim < eatFood)
  (playSkyrim < useBathroom)
```

Other

1. Additional Code Changes

   I changed `printstate()` to print all order conditions, even the ones with init and goal. This made it easier to debug the output.

   I added a method `stepExists()` to the `Plan` class to handle checking for steps in the step list that have a number appended to them. `for step in plan.steps` doesn't work since steps in plan.steps look like: 1.step_name

2. Notes

   I didn't include the block stacking exercise since it looked like it wasn't defined properly. The goals lack `clear()` and `on()` propositions and it makes them unsatisfiable.

3. Files

   Executing my custom scenario can be done by running the command: `python assign4code.py 3`
   Additional scenarios are:

```
1 – cleaning
2 – blocks (doesn't seem to be properly defined)
4 – tires
```

   Full plan dumps for the plans mentioned in the “Planning Results” section can be found in the output/ folder included in the submission.