CS 2710 Fall 2015 (Foundations of Artificial Intelligence)

Assignment 4: A Planning System

Mohammad Hasanzadeh Mofrad (hasanzadeh@cs.pitt.edu)

1 Introduction

A Partial Order Planning (POP) algorithms tries to automate planning that leaves decisions about the ordering of actions as open as possible. A Partial order plan consists of the following components:

Steps: A set of steps or actions that we plan to take.

Ordering constraints: A set of ordering constraints that say which steps have to be before which other ones.

Variable binding constraints: A set of variable ordering constraints that say which variables are equal to which other variables or constants.

Casual links: A set of links that say which effect of actions is going to satisfy which preconditions of other actions.

In this report, we are going to build a planning system for two simple problems in the context of propositional representation using the above description.

2 Proposed Planning System

In this section, the changes to the assign4code.py will be introduced. Since the assignment description asks for relevant changes to the code, each section comes with its corresponding code snippet with an example showing that the code is capable of doing a specific task.

2.1 Goal Test

The goal function checks threats and open conditions. If there exists a threat or open condition, the goal function returns false. If the is no threat and open condition, it return true.

```
def goalp(node):
  if not node.state.threats and not node.state.openconds:
     return True
  else:
     return False
```

2.2 Heuristic Function

The heuristic function checks the number of threats and open conditions, and return them for further use.

```
def h(node):
  return (len(node.state.threats) + len(node.state.openconds))
```

2.3 Successor Function

As stated in the project description, the successor function receives a plan with some flaws and tries to resolve these flaws. Plan may include two types of flaws: open conditions and/or threats. In the following, we introduce the ways of eliminating these flaws and present their corresponding code snippets and traces.

2.3.1 Resolving an open condition by using an existing step

Let (C, X) be an open condition. The first way to solve an open condition (C, X) is to find some other step, S, already in the plan that makes the condition true i.e. S has C on its add list. The following code snippet belongs to this way of resolving open condition.

```
# Select the first open condition
selectedOC = plan.openconds[0]
for S in plan.steps:
   if (selectedOC[0] in adds[S]) and (selectedOC[1] != S):

      # Create successor node
      successorNode = Plan(steps = list(plan.steps),\
                           ordercons = list(plan.ordercons),\
                           causallinks = list(plan.causallinks),\
                           openconds = list(plan.openconds),\
                           threats = list(plan.threats))

      # Create additional causal link
      if not (S, selectedOC[0], selectedOC[1]) in successorNode.causallinks:
         addCL = (S, selectedOC[0], selectedOC[1])
         successorNode.causallinks.append(addCL)

      # Create additional ordering constraint
      addOrC = (S,selectedOC[1])
      if orderConsistency(addOrC,successorNode.ordercons):
         if not (S, selectedOC[1]) in successorNode.ordercons:
            successorNode.ordercons.append(addOrC)

      #Check for new threats
      for tS in successorNode.steps:
         if (selectedOC[0] in deletes[tS]) and (tS != selectedOC[1]) and \
               (not (selectedOC[1], tS) in successorNode.ordercons) and \
               (not (tS, S) in successorNode.ordercons):
            if not ((S, selectedOC[0], selectedOC[1]), tS) in successorNode.threats:
               successorNode.threats.append(((S, selectedOC[0], selectedOC[1]), tS))

      #remove open condition
      successorNode.openconds.remove(selectedOC)
      successorNode.comment = "Adding causal link with existing step\n" +\
                              "     The new causal link is: " + str(addCL) +\
                              "\n     Removing the open condition: " + str(selectedOC) +\
                              "\n"
      successor.append(successorNode)
```

Moreover, the following is a part of trace file showing how to resolve an open condition for plan 6 using an existing steps:

```
open conditions:
    floor-dirty wash-floor
    furniture-dusty dust
    floor-dusty sweep
threats:
    (('sweep', 'floor-not-dusty', 'goal'), 'dust')
    (('sweep', 'floor-not-dusty', 'wash-floor'), 'dust')
plan6 -----
Adding causal link with existing step
    The new causal link is: ('init', 'floor-dirty', 'wash-floor')
    Removing the open condition: ('floor-dirty', 'wash-floor')

steps: ['init', 'goal', 'wash-floor', 'dust', 'sweep']
causal links:
    wash-floor < floor-clean < goal
    dust < furniture-clean < goal
    sweep < floor-not-dusty < goal
    sweep < floor-not-dusty < wash-floor
    init < floor-dirty < wash-floor
ordering constraints (other than those with goal or init):
    sweep < wash-floor
open conditions:
    furniture-dusty dust
    floor-dusty sweep
threats:
    (('sweep', 'floor-not-dusty', 'goal'), 'dust')
    (('sweep', 'floor-not-dusty', 'wash-floor'), 'dust')
```

2.3.2 *Resolving an open condition by inserting a new step*

Let (C, X) be an open condition. The second way to resolve achieve an open condition (C, X) is to find an operator that makes it true, and then insert a new step S representing that operator into the plan. The following code snippet belongs to this way of resolving open condition.

```
for newS in adds:
   if (selectedOC[0] in adds[newS]) and (newS != selectedOC[1]):
      if not newS in plan.steps:
         successorNode = Plan(steps = list(plan.steps),\
                              ordercons = list(plan.ordercons),\
                              causallinks = list(plan.causallinks),\
                              openconds = list(plan.openconds),\
                              threats = list(plan.threats))
         # Inserting a new step
         successorNode.steps.append(newS)

         # Create additional causal link
         addCL = (newS, selectedOC[0], selectedOC[1])
         successorNode.causallinks.append(addCL)

         # Eliminate the open condition
         successorNode.openconds.remove(selectedOC)

         # Create additional ordering constraint
         addOrC = (newS, selectedOC[1])
#          if (selectedOC[1]!='goal' and selectedOC[1]!='init') and \
         if orderConsistency(addOrC, successorNode.ordercons):
            successorNode.ordercons.append(addOrC)

        # add preconditions of the new step to the existing open conditions
         for precondition in preconds[newS]:
            successorNode.openconds.append((precondition, newS))

         #Check for new threats
         for tS in successorNode.steps:
            if (selectedOC[0] in deletes[tS]) and (tS != selectedOC[1]) and \
                  (not (tS, newS) in successorNode.ordercons) and \
                  (not (selectedOC[1], tS) in successorNode.ordercons):
               if selectedOC[0] in deletes[tS]:
                  successorNode.threats.append(((newS, selectedOC[0], selectedOC[1]), tS))

         for proposition in deletes[newS]:
            for causalLink in successorNode.causallinks:
               if (proposition in causalLink) and (not newS in causalLink):
                  successorNode.threats.append((causalLink, newS))

         successorNode.comment = "Adding new step " + newS +\
                                 "\n   The new causal link is: " + str(addCL) +\
                                 "\n   Removing the open condition: " + str(selectedOC) +\
                                 "\n"

         successor.append(successorNode)
```

Moreover, the following is a part of trace file showing how to resolve an open condition for plan 4 by adding a new step:

```
plan4 -----
Adding new step sweep
   The new causal link is: ('sweep', 'floor-not-dusty', 'goal')
   Removing the open condition: ('floor-not-dusty', 'goal')

steps: ['init', 'goal', 'wash-floor', 'dust', 'sweep']
causal links:
    wash-floor < floor-clean < goal
    dust < furniture-clean < goal
    sweep < floor-not-dusty < goal
open conditions:
    floor-not-dusty wash-floor
    floor-dirty wash-floor
    furniture-dusty dust
    floor-dusty sweep
threats:
    (('sweep', 'floor-not-dusty', 'goal'), 'dust')
```

2.3.3 *Resolving a threat by demotion*

A threat can be viewed as having four components: a producer step, a consumer step, a condition, and a threatening step. While solving a threat by *demotion*, we add a new ordering constraint that requires *the threatening step to precede the producer*. The following is the code snippet for resolving threats by demotion.

```
selectedT = plan.threats[0]
 successorNode = Plan(steps = list(plan.steps),\
                 ordercons = list(plan.ordercons),\
                 causallinks = list(plan.causallinks),\
                 openconds = list(plan.openconds),\
                 threats = list(plan.threats))

 # Demotion by adding a new ordering constraint
 addOrC = (selectedT[1], selectedT[0][0])
 if orderConsistency(addOrC, successorNode.ordercons) and (len(plan.threats)>1):
    if (not addOrC in successorNode.ordercons) and (addOrC[0] != addOrC[1]):
       successorNode.ordercons.append(addOrC)
    # Eliminate the resolved threat
    successorNode.threats.remove(selectedT)
    successorNode.comment = "Resolving threat by demotion: order constraints " +\
                            str(addOrC) + "\n    Threat eliminated " +\
                            str(selectedT) + "\n"
```

Moreover, the following is a part of trace file showing how to resolve a threat for plan 10 by using *demotion*:

```
ordering constraints (other than those with goal or init):
    sweep < wash-floor
threats:
    (('sweep', 'floor-not-dusty', 'goal'), 'dust')
    (('sweep', 'floor-not-dusty', 'wash-floor'), 'dust')

plan10 -----
Resolving threat by demotion: order constraints ('dust', 'sweep')
    Threat eliminated (('sweep', 'floor-not-dusty', 'goal'), 'dust')

steps: ['init', 'goal', 'wash-floor', 'dust', 'sweep']
causal links:
    wash-floor < floor-clean < goal
    dust < furniture-clean < goal
    sweep < floor-not-dusty < goal
    sweep < floor-not-dusty < wash-floor
    init < floor-dirty < wash-floor
    init < furniture-dusty < dust
    init < floor-dusty < sweep
ordering constraints (other than those with goal or init):
    sweep < wash-floor
    dust < sweep
threats:
    (('sweep', 'floor-not-dusty', 'wash-floor'), 'dust')
```

2.3.4 Resolving a threat by promotion

While solving a threat by *promotion*, we add a new ordering constraint that requires *the consumer to precede the threatening step*. The following is the code snippet for resolving threats.

```
else:
   # Promotion by adding a new ordering constraint
   addOrC = (selectedT[0][1], selectedT[1])
   if orderConsistency(addOrC, successorNode.ordercons):
      if not addOrC in successorNode.ordercons and (addOrC[0] != addOrC[1]):
         successorNode.ordercons.append(addOrC)
      # Eliminate the resolved threat
      successorNode.threats.remove(selectedT)
      successorNode.comment = "Resolving threat by promotion: order constraints "+\
                              str(addOrC) + "\n    Threat eliminated " +\
                              str(selectedT)+ "\n"
successor.append(successorNode)
```

Also, the following is a part of trace file showing how to resolve a threat for plan 12 by using *promotion*:

```
ordering constraints (other than those with goal or init):
    sweep < wash-floor
    dust < sweep
threats:
    (('sweep', 'floor-not-dusty', 'wash-floor'), 'dust')
plan12 -----
Resolving threat by promotion: order constraints ('floor-not-dusty', 'dust')
    Threat eliminated (('sweep', 'floor-not-dusty', 'wash-floor'), 'dust')

steps: ['init', 'goal', 'wash-floor', 'dust', 'sweep']
causal links:
    wash-floor < floor-clean < goal
    dust < furniture-clean < goal
    sweep < floor-not-dusty < goal
    sweep < floor-not-dusty < wash-floor
    init < floor-dirty < wash-floor
    init < furniture-dusty < dust
    init < floor-dusty < sweep
ordering constraints (other than those with goal or init):
    sweep < wash-floor
    dust < sweep
    floor-not-dusty < dust
```

Note that, I added a late condition to enforce a specific circumstance for applying a promotion based on the length of the threat list (len(plan.threats)>1). Therefore, by removing this condition, the algorithm only uses demotion which is comes first in the code. Thus, in this case the final ordering constraints will be as follows:

```
    sweep < wash-floor
    dust < sweep
```

2.4 Temporal Ordering Constraint Consistency

A set of ordering constraints is consistent if three conditions hold:

1. No step precedes step *init.*
2. Step *goal* does not precede any step.
3. No step S_i precede itself in the transitive closure of the ordering constraints.

The following is the code snippet regarding these constraints checks:

```python
def orderConsistency(addOrC,constraints):
   # Verify the new  ordering constraint position
   # No step preceds 'init'
   # Step 'goal' does not preceds any step
   steps=["init","goal"]
   if addOrC[0] in steps or addOrC[1] in steps:
      return False
   else:
      # Transitivity check
      for closure in constraints:
         if addOrC[0] == closure[1] and addOrC[1] == closure[0]:
            return False
      return True
```

3 Developing a new problem

In this section we test the planning algorithm with a new problem which is hardcoded in assign4code.py. To run this test case use the following command:

```
python3 assign4code.py 3
```

The proposed problem tries to plan a banking solution for transferring money:

1. A user want to login to the online banking system
2. There is a step for checking the user name.
3. There is another step for checking the password.
4. After successful authentication, the user is logged in to the online banking system.
5. The user could check its current balance or withdraw some money.
6. Finally, the user will logout from the system.
7. The goal is to login to system and withdraw 100 bucks and then logout.

3.1 Resolving an open condition by using an existing step

The following is an example of solving an open conditions by using an existing step:

```
plan8 -----
Adding causal link with existing step
    The new causal link is: ('init', 'unknown-balance', 'current-balance')
    Removing the open condition: ('unknown-balance', 'current-balance')

steps: ['init', 'goal', 'login', 'current-balance', 'withdraw', 'logout', 'enter-username',
'enter-password']
causal links:
    login < successful-login < goal
    current-balance < sufficient-balance < goal
    withdraw < withdraw-100-bucks < goal
    logout < not-login < goal
    enter-username < valid-username < login
    enter-password < valid-password < login
    init < unknown-balance < current-balance
ordering constraints (other than those with goal or init):
    enter-username < login
    enter-password < login
open conditions:
    successful-login withdraw
    unknown-balance withdraw
    successful-login logout
    blank-username enter-username
    valid-username enter-password
    blank-password enter-password
threats:
    (('current-balance', 'sufficient-balance', 'goal'), 'withdraw')
    (('logout', 'not-login', 'goal'), 'login')
```

3.2 *Resolving an open condition by inserting a new step*

The following is an example of solving an open conditions by adding a new step:

```
plan7 -----
Adding new step enter-password
   The new causal link is: ('enter-password', 'valid-password', 'login')
   Removing the open condition: ('valid-password', 'login')

steps: ['init', 'goal', 'login', 'current-balance', 'withdraw', 'logout', 'enter-username',
'enter-password']
causal links:
    login < successful-login < goal
    current-balance < sufficient-balance < goal
    withdraw < withdraw-100-bucks < goal
    logout < not-login < goal
    enter-username < valid-username < login
    enter-password < valid-password < login
ordering constraints (other than those with goal or init):
    enter-username < login
    enter-password < login
open conditions:
    unknown-balance current-balance
    successful-login withdraw
    unknown-balance withdraw
    successful-login logout
    blank-username enter-username
    valid-username enter-password
```

3.3 *Resolving a threat by demotion*

Since asking for the current balance is independent of withdrawing money, there is a flaw between withdrawing money and query about current balance. The following shows that how the algorithm resolve this threat.

```
threats:
    (('current-balance', 'sufficient-balance', 'goal'), 'withdraw')
    (('logout', 'not-login', 'goal'), 'login')

plan15 -----
Resolving threat by demotion: order constraints ('withdraw', 'current-balance')
    Threat eliminated (('current-balance', 'sufficient-balance', 'goal'), 'withdraw')

steps: ['init', 'goal', 'login', 'current-balance', 'withdraw', 'logout', 'enter-username',
'enter-password']
causal links:
    login < successful-login < goal
    current-balance < sufficient-balance < goal
    withdraw < withdraw-100-bucks < goal
    logout < not-login < goal
    enter-username < valid-username < login
    enter-password < valid-password < login
    init < unknown-balance < current-balance
    login < successful-login < withdraw
    init < unknown-balance < withdraw
    login < successful-login < logout
    init < blank-username < enter-username
    enter-username < valid-username < enter-password
    init < blank-password < enter-password
ordering constraints (other than those with goal or init):
    enter-username < login
    enter-password < login
    login < withdraw
    login < logout
    enter-username < enter-password
    withdraw < current-balance
threats:
    (('logout', 'not-login', 'goal'), 'login')
```

3.3 Resolving a threat by promotion

The logout action, adds the not-login to the plan, which is among the goals. Also, not-login is a deleted by login before. Thus, there is a threat here. The following shows that how the algorithm resolve this threat by promotion.

```
threats:
    (('logout', 'not-login', 'goal'), 'login')

plan16 -----
Resolving threat by promotion: order constraints ('not-login', 'login')
    Threat eliminated (('logout', 'not-login', 'goal'), 'login')

steps: ['init', 'goal', 'login', 'current-balance', 'withdraw', 'logout', 'enter-username',
'enter-password']
causal links:
    login < successful-login < goal
    current-balance < sufficient-balance < goal
    withdraw < withdraw-100-bucks < goal
    logout < not-login < goal
    enter-username < valid-username < login
    enter-password < valid-password < login
    init < unknown-balance < current-balance
    login < successful-login < withdraw
    init < unknown-balance < withdraw
    login < successful-login < logout
    init < blank-username < enter-username
    enter-username < valid-username < enter-password
    init < blank-password < enter-password
ordering constraints (other than those with goal or init):
    enter-username < login
    enter-password < login
    login < withdraw
    login < logout
    enter-username < enter-password
    withdraw < current-balance
    not-login < login
```