1 Integrity Agreement

This must be your own individual work.

By submitting a solution to this assignment, you attest that you (1) did all of the work, (2) that you did it alone, and (3) that you did it without using resources outside of those provided by class (CS2710/ISSP 2160 Foundations of Artificial Intelligence, Fall 2015), unless you explicitly state otherwise (in which case exactly what must be clearly indicated).

2 Grading

Your assignment will be graded on:

- Correctness (75%)
- Completeness and clarity of report (25%)

3 Introduction

Planning has been a widely studied topic in Artificial Intelligence. The basic planning problem is as follows.

Given a description of an initial state, a goal state, and a set of possible actions, compute a sequence of actions that will lead to the goal state if performed in the initial state. Over the years, there have been many variations on and elaborations of this basic problem, along with many approaches to solving them.

You will build a planning system for a basic problem, using the technique of partial-order planning in plan space. Then, you will apply your system to a new planning problem that you develop.

To keep things simple, we will only deal with a propositional representation. Note that, if there are a finite number of objects and no function symbols, as in common planning languages, a first-order domain may be represented propositionally by defining multiple actions for each action schema. For example, in place of “puton(X,Y)”, we could introduce the following actions (assuming there are three blocks: a, b, and c): “puton-a-b”, “puton-a-c”, “puton-b-a”, “puton-b-c”, “puton-c-a”, “puton-c-b”.

States of the world will be described in terms of sets of propositions, and actions will be described as operators with three lists of propositions: a precondition list (what must be true for the action to be performed), an add list (propositions that are made true when the
action is performed), and a delete list (propositions that are made false when the action is performed).

For example, we might describe a planning problem for a housecleaning robot as follows:

Initial state: \{floor-dusty, floor-dirty, furniture-dusty\}

Goal state: \{floor-clean, furniture-clean\}

Operators:

Sweep: Preconds: \{floor-dusty\},
          Add: \{floor-not-dusty\},
          Delete: \{floor-dusty\}

Vaccuum: Preconds: \{floor-dusty, vaccuum-works\},
          Add: \{\},
          Delete: \{floor-dusty\}

Wash-floor: Preconds: \{floor-not-dusty, floor-dirty\},
              Add: \{floor-clean\},
              Delete: \{floor-dirty\}

Dust: Preconds: \{furniture-dusty\},
      Add: \{floor-dusty, furniture-clean\},
      Delete: \{floor-not-dusty, furniture-dusty\}

We will model a plan as consisting of five components:

1. a set of steps, which are instances of actions
2. a set of temporal ordering constraints on those steps
3. a set of causal links, which represent causal relationships between steps. Formally, a causal link is a triple \(si, p, sj\) where \(si\) is a step in the plan that makes \(p\) true and \(sj\) is a step in the plan that has \(p\) on its precondition list.
4. a set of open conditions, which are preconditions of steps that still need to be made true;
5. a set of threats, where a threat is a \(\langle\text{causal link, threatening step}\rangle\). An effect of the threatening step clobbers the middle element of the causal link.

Here is a solution plan for the problem just described. “\(si: \text{action-type}\)” means that “\(si\)” is an action of type \(\text{action-type}\).
steps: ['init', 'goal', '0*wash-floor', '1*dust', '2*sweep']
causal links:
(0*wash-floor < floor-clean < goal)
(1*dust < furniture-clean < goal)
(2*sweep < floor-not-dusty < 0*wash-floor)
(init < floor-dirty < 0*wash-floor)
(init < furniture-dusty < 1*dust)
(init < floor-dusty < 2*sweep)
ordering constraints (other than those with goal or init):
(2*sweep < 0*wash-floor)
(1*dust < 2*sweep)
no openconditions or threats

Note that there are two reasons for the ordering constraints the plan includes. The first one, (2*sweep < 0*wash-floor), is included because of the causal link:

(2*sweep < floor-not-dusty < 0*wash-floor).

That is, a sweep action achieves floor-not-dusty, which is a precondition of the wash-floor action.

The second one, (1*dust < 2*sweep), is included to resolve the following threat:

((’2*sweep’, ’floor-not-dusty’, ’0*wash-floor’), ’1*dust’)

A "dust" action "clobbers" the condition "floor-not-dusty". The problem is resolved by ordering the "dust" action before the "sweep" action.

4 Constructing a Partial-Order Planner

In this assignment, you will construct a system that generates plans by searching through the space of partial plans. Specifically, you should complete “assign4code.py”, available on the course web page.

To make grading the project more feasible, please produce the same type of output as in the output examples on the class schedule.
4.1 States

In a plan-space planner, you begin with a dummy initial plan that includes two steps. One represents the initial state, and it has no preconditions, but has an add list containing everything that is true initially. Essentially, the initial step makes true the initial conditions.

The other step of the initial plan represents the goal state. It has no effects (i.e., its add and delete lists are empty), but its preconditions are equal to the goal propositions.

During the planning process, you will also store information in each plan about its flaws, i.e., its open conditions and threats.

The dummy initial plan has as open conditions all the preconditions of the goal step. It does not contain any threats, because it does not yet contain any causal links.

4.2 Goal Test

The goalp test takes a plan as an argument and checks whether the plan is complete. A complete plan is one that has no open conditions and no threats.

4.3 Heuristic Function

There is much literature on how to control the search during planning. In this project, we will simply rank a plan by the number of open conditions and threats it contains, on the assumption that the fewer the number of flaws, the better. Your heuristic evaluation function should take a plan as an argument, and return the sum of the number of its open conditions and threats.

4.4 Successor Function

The interesting part is the successor function. Remember that the nodes in the search space represent plans, and, until a goal node is reached, each of those plans will have some flaws: open conditions and/or threats. To produce the successors of a node, what you will do is to select some flaw, and then compute all the ways of correcting (“resolving”) the flaw.

Let’s consider the two types of flaws, starting with open conditions. Recall that an open condition is a precondition of some step that needs to be made true.

Let (C,X) be an open condition. There are two ways to resolve an (C,X). The simplest 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).

For every step S already in the plan that can be re-used to achieve the selected open condition, you will create a successor node that has an additional causal link (S,C,X), along with an additional ordering constraint that requires S to precede X. We then need to check if there is a new threat: it is possible that another step, Y, which is already in the plan, threatens the new causal link. This happens if C is on the delete list of Y, and Y is not already ordered before S or after X. (Be careful with operators such as sweep, with dusty as a precondition and dusty as a delete. You don’t want your program to think that an action threatens itself!)
The other way to 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. In this case, in the successor plan we again need to add a causal link and an ordering constraint between \(S\) and \(X\). We’ve also got to add ordering constraints to ensure that \(S\) occurs between steps init and goal. We’ve got to add all of the preconditions of \(S\) to the set of open conditions. And, we’ve got to update the list of threats associated with the plan, to include any new threats that were introduced by \(S\). That means that we need to look at the steps in the plan, to determine whether any of them delete the newly satisfied open conditions, and we need to look at all the causal links, to see if the newly inserted step deletes the propositions of any of the causal links. Remember that if there are multiple operators that can achieve the selected open condition, we will create multiple successor nodes.

Now let’s consider how to resolve threats. A threat can be viewed as having four components: a producer step, a consumer step, a condition (that the producer makes true for the consumer), and a threatening step. There are two ways of resolving threats:

1. demotion: adding an ordering constraint that requires the threatening step to precede the producer
2. promotion: adding an ordering constraint that requires the consumer to precede the threatening step.

The last interesting complication to our planning problem: whenever we introduce a new temporal ordering constraint, we need to make sure that it is consistent. 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 \(s_i\) precedes itself in the transitive closure of the ordering constraints. For example, you can’t have \(s_5\) before \(s_7\) and \(s_7\) before \(s_5\).

5 Testing

Submit your completed version of assign4code.py. You should include the cleaning problem above, and a new problem you develop.

Submit a report that demonstrates that your code works by using appropriate output and adding appropriate comments. For each case, give an example showing that your code is correct (e.g., a threat resolved by promotion, a threat resolved by demotion, resolve an open condition using an action that is already in the plan, etc.). If your report does not do this, you will receive a severe penalty.
6 Submission

Please submit your solution to the incoming folder in AFS:
/afs/cs.pitt.edu/public/incoming/CS2710/assignment4

If you do not have a cs account, then please email your solution to the TA.

Please submit one zipfile containing: your report, your code, any required input files, and a README file explaining how to run your program.

Use the following naming convention:

lastnameFirstname_(Version).zip (everything lowercase)
examples:
akkaya_cem.zip
akkaya_cem_v2.zip

If you wish to submit a correction, please append a version number to the new archive to indicate this version should be graded instead. For example, if the original file was ‘akkaya_cem.zip’ then please submit ‘akkaya_cem_v2.zip’ as the updated version.

Please submit by 11:59pm on the due date. Late assignments will not be accepted without documented extreme circumstances.