Artificial Intelligence CPSC 470/570
PS 4 – Planning in the Blocks World
20 points
Due Friday, March. 12Th, 10:30 AM

Some reminders:

- **Special grading note:** This problem set will be graded by an autograder. Your grade from the autograder will be on a scale of 0-100 points. To make this comparable to the point totals from other problem sets, we will divide this total by 5, for an overall problem set 4 score between 0 and 20 points.
- **Grading contact:** Debasmita Ghose (debasmita.ghose@yale.edu) is the point of contact for initial questions about grading for this problem set.
- **Late assignments** are not accepted without a Dean's excuse.
- **Collaboration policy:** You are encouraged to discuss assignments with the course staff and with other students. However, you are required to implement and write any assignment on your own. This includes both pencil-and-paper and coding exercises. You are not permitted to copy, in whole or in part, any written assignment or program as part of this course. You are not to take code from any online repository or web source. You will not allow your own work to be copied. Homework assignments are your individual responsibility, and plagiarism will not be tolerated.
- **Students taking CPSC570:** There is no extra section for this assignment. Your assignment is the same as CPSC470.

Introduction

In this problem, you will construct some key components of a partial-order planner that can solve problems in the blocks world. The blocks world domain consists of a finite set of blocks and a table large enough to accommodate several stacks of blocks. Each block is either on another block or on the table. No block can be on two places at the same time. You control a robotic arm that can reach the top block of any stack of blocks on the table. The arm can hold only one block at a time. You may also assume for this problem that the robot is completely reliable, that is, any valid command that you give to the robot will be carried out without error.

You will use the following representation for operators and knowledge predicates (*please note that those predicates and the operators are slightly differently from what is covered in lecture*):

Two Predicates:

$CLEAR(y)$: indicates that y has space on top of it for a block
$ON(x, y)$: indicates that block x is on y (y is directly below block x)

One Operator:

$MOVE(b, x, y)$: move block b from a position on top of x to on top of y

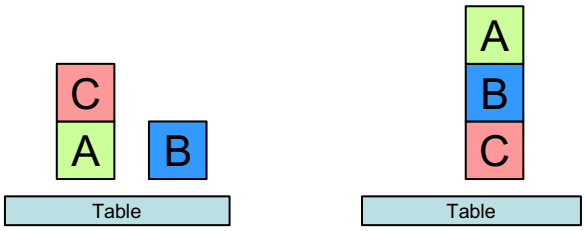
You will have to deal with the following objects :

Blocks - `a, b, c, d,` etc.

Table slots – `table0, table1, table2,` etc.

The state of the world and the goal state will both be described using these representations. We will represent a state within the world as a list of clauses that are implicitly connected by conjunction. Pictured below are three example scenarios. For each scenario, we show the initial state of the world (at left), the goal state (at right), and the descriptions of those states.

Example #1:

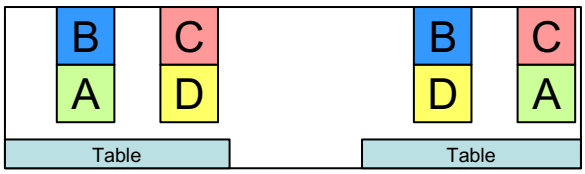


Representation:

Initial State: $on(c, a) \wedge on(b, table1) \wedge on(a, table0) \wedge clear(c) \wedge clear(b)$

Goal State: $on(a, b) \wedge on(b, c)$

Example #2:

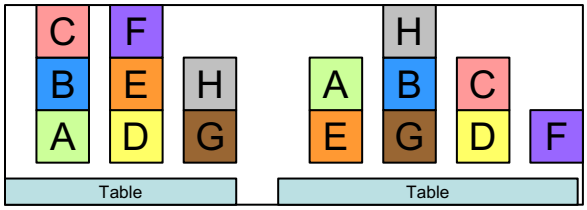


Representation:

Initial State: $on(b, a) \wedge on(c, d) \wedge on(a, table0) \wedge on(d, table1) \wedge clear(b) \wedge clear(c)$

Goal State: $on(b, d) \wedge on(c, a)$

Example #3:



Representation:

Initial State: $on(c,b) \wedge on(b,a) \wedge on(a,table0) \wedge on(f,e) \wedge on(e,d) \wedge on(d,table1) \wedge on(h,g) \wedge on(g,table2) \wedge clear(c) \wedge clear(f) \wedge clear(h)$

Goal State: $on(a,e) \wedge on(h,b) \wedge on(b,g) \wedge on(c,d) \wedge on(e,table0) \wedge on(g,table1) \wedge on(d,table2) \wedge on(f,table3)$

Starter Code Introduction

You will see the following files in the starter code:

- __init__.py
- condition.py
- link.py
- ordered_set.py: a data structure that removed the randomized in set
- plan.py
- **planner.py**
- step.py
- **student_test_case.py**
- student_tests/: test cases provided for you

The highlighted files are the ones you will modify and summit (planner.py and student_test_case.py). Below is a brief introduction to the rest of the files, which also provides a brief introduction of how the block world problem is represented.

Step (step.py):

A single uniquely identified step (action by the robot) in the plan. Each lists the operator that is performed as well as a list of the preconditions and effects.

```
class Step:
    def __init__(self, …):
        …
        self.identity = identity
        self.operator = operator
        self.preconditions = []
        self.effects = []
        …
```

with components defined as follows:

identity (or id):

an arbitrary unique integer representing the step
(e.g. 0 for the starting step and 1 for the last step). The id does not have to be in any particular order; it is intended only to serve as an aid in bookkeeping.

operator:

the operation that will be preformed
For example "`move a b c`" means "take block `a`, which is currently on top of block `b`, and put it on top of block `c`". The step-operator for the start state is "`start`" and the operator for the last state is "`finish`".

precondition:

A list prerequisites for applying the step's operator. It is a list of Condition objects.

effects:

A list of effects caused by applying the step's operator. It is a list of Condition objects.

Examples:

In the test cases, you will see a step in this format. For example, in test1.py:

```
Step(1,"finish",[
        Condition(True,"on b table2"),
        Condition(True,"on c b"),
        Condition(True,"on d c"),
        Condition(True,"on e a")],
            [])
```

Condition (condition.py):

A single clause stating a condition of the world. The clause can either be true, or false, depending on the state. It is identified by a single predicate.

```
class Condition:
    def __init__(self, …):
        …
        self.state = True/False
        self. predicate = []
        …
```

with components defined as follows:

state:

If this is true, the condition states that the predicate is true, if it is false the condition states that the predicate is false.

predicate:

A list of strings/identifiers that form a predicate.
The first identifier will be the name of the predicate, while the rest will be parameters. In our block world, a predicate can be in the form of [“on”, “x”, “y”] *or* [“clear”, “x”]*, where x and y can be a block or a table location.*

Example:

In the test cases, you will see a step in this format. For example, in test1.py:

```
Condition(True,"on b table2")
```

You will see some conditions with the state False in the test cases. Those are used primarily in part A to check if there are conflicts.

Link (link.py):

A causal link within the plan, indicating that step with id1 creates the condition effect, which in turn is needed by the step with id2.

```
Class Link:
    def __init__(self, …):
        …
        self.id1 = id1
        self.id2 = id2
        self.effect = Condition(…)
        …
```

with components defined as follows:

id1:

The id of the step causing the effect.

id2:

The id of the step requiring the effect.

effect:

The condition the link represents

Example:

In the test cases, you will see a step in this format. For example, in test1.py:

```
Link(0,2,Condition(True,"on a b"))
```

Plan (plan.py):

The full partial order plan. The plan will consist of a list of uniquely identified steps, a list of ordering constraints, and a list of causal links.

```
class Plan:
    def __init__(self, …):
        …
        self.steps = []
        self.ordering_constraints = []
        self.causal_links = []
        …
```

with components defined as follows:

steps:

The uniquely identified steps of the plan. For the most part they are in no particular order. However, "start" will be the first element, and "finish" will be the second element. It is a list Step objects.

ordering_constraints:

an ordering constraint in the form [id_a , id_1 , id_2 … id_n].
This is equivalent to the statement:

($id_a \prec id_1$) ^ ($id_a \prec id_2$) ^ … ^($id_a \prec id_n$)

For example [0, 2, 3, 1] says "step 0 occurs before 2, 3, and 1", but makes no claim about the ordering of 2, 3 and 1 in relation to each other. If you update your ordering constraints for a particular target a as the *transitive closure* of a, your life may be made considerably easier during the linearization phase. A *transitive closure* is full list of all successors for any given target. It is a list of int.

causal_links:

A list of causal links, in no particular order.

Examples:

You can find some examples of steps, ordering_constraints and causal_links in the test1.py files (and some other test files) in the folder student_tests.

In plan, we represent a problem as a graph of actions (the nodes in the graph) and two types of linkages between steps: ordering constraints and causal links.

Each **Step** is equivalent to the actions that covered in class. The world mains a list of Conditions that describe the current state. The initial world state is an empty set as the precondition of “start” is an empty set. It is initialized with the effect of “start” which represent the problem to be solved. The goal state is represented as the precondition of “finish”. The preconditions in general are the conditions to be deleted from the current world state if the step is performed, and they must present in the current world state. The effect are the conditions to be added after the step is performed.

Each **ordering constraint** is of the form "A before B" and means that action A must be executed sometime before action B, but not necessarily immediately before. A valid partial-order plan cannot contain cycles of ordering constraints. Such cycles (such as "A before B" and "B before A") represent a contradiction and cannot be allowed to be added to a partial-order plan. This is more of an explicit presentation between the ordering of steps.

Each **causal link** is of the form "A achieves p for B" and describes how the effect of action A causes condition p to be achieved (asserted as true) which satisfies a pre-condition of action B.

For example, we may have "*PutOnRightSock* achieves *RightSockOn* for *PutOnRightShoe*". That is, when the action *PutOnRightSock* is taken, it asserts the property *RightSockOn*, which is a pre-condition for the action of putting on your right shoe (*PutOnRightShoe*).

Causal links also necessarily imply an ordering. You cannot perform some action B that relies on a pre-condition p without first performing the action A that achieves that pre-condition.

You can think of each causal link as implying an implicit ordering constraint. This constraint may not be listed in the list of explicit ordering constraints.

Part A: Consistency, Completeness and Linearization (46 autograder points)

In this part, you will fill in the #TODO part in ***planner.py*** that implements the class ***Planner***. You must finish the following methods:

def isComplete(self, plan)

Returns true if the plan is complete, false otherwise.
A plan is complete if there are no open preconditions. A pre-condition is open if it is not achieved by some action with the causal links. For example, if *Condition(True,"on a b")* is one of the preconditions of step 2, and with the presence of the causal link *Link(0, 2, Condition(True,"on a b"))*, then the condition is closed since there is a step, which is 0, that can result in the condition for step 2.

def isConsistent(self, plan)

Returns true if the plan is consistent, false otherwise.
A plan is consistent if there are no cycles in the ordering constraints and no conflicts with the causal links.
For example,
1. the following ordering_constraints is not consistence:
    ordering_constraints.append([6,7,51])
    ordering_constraints.append([7,6])
Since there is a cycle between 6 and 7
2. the following causal links with the steps defined are also not consistent:
    Step(2,"...",[...],[..., Condition(True,"on b table0"), ...]),
    Step(3,"...",[...],[..., Condition(False,"on b table0"), ...]),
    Step(4,"...",[..., Condition(True,"on b table0"), ...],[...]),
    Link(2,3,Condition(True,"..."))
    Link(3,4,Condition(True,"..."))
Since there is a step that can appear between these two steps that conflicts with it.
Here are more explanations. A plan has a conflict with its casual links if there exists any action C that is permitted to appear between actions A and B and one of C's post conditions contradicts a causal link between A and B.
For example, let's say there exists a causal link "*EatCookie* achieves *HandsFree* for *WashHands*" and *PickUpIceCream* has the effect "not *HandsFree*". If *PickUpIceCream* is allowed to appear between *EatCookie* and *WashHands*, based on the ordering and casual links, then a conflict exists. However if there is, say, an ordering constraint saying that *PickUpIceCream* must come after *WashHands* then no conflict exists.

def createLinearization(self, plan)

Takes a plan and constructs an arbitrary linearization of that plan

It returns a list of operators representing that plan. In the block world, each operation must be one of the following: *"start"*, *"finish"*, or *"move x y"* where x and y are then names of blocks or table locations.
Your code is not required to generate linearization if the test case doesn't have a solution.

Please remove the code between #TODO START and #TODO END. And please **DO NOT** modify any other code except for the main function.

There are several other functions in planner.py, and here are some brief explanations:

Helper functions:

def getDirectPrecursors(self, step, plan)

given a step and a plan, return a set of steps who are the director precursors (parent if in a graph) of the target step. A director precursor is the step that has to immediately occur before the target step

def getAllPrecursors(self, step, plan)

given a step and a plan, return a set of steps who are the precursors (antecedent if in a graph) of the target step. A precursor is any step that has to occur before the target step, regardless whether it is immediate or not.

def get_parameters(self, file_name)

import the test cases defined with the file_name. You can either use test1 or test1.py

def run_test(self, steps, ordering_constraints, causal_links, test_name="")

given the steps, ordering_constraints, causal_links, return whether it is complete, consistent, a solution and list the linearization. The test_name is optional which is the name of the currently test running, so that it is convenient which test results is which when run in bulk

Other functions:

def isSolution(self, plan)

Returns true if the plan is a solution, false otherwise.
A plan is a solution if it is consistent and complete.

def main()

The function that will run if you run this python script. Feel free to add/delete code in the designated section. We have provided some sample code to show how the planner is typically invoked.

Other functions are irrelevant to this part.

To test your code, the results of the following test cases are provided for you:

```
This plan is:  test1
Complete:  False
Consistent:  False
Solution:  False
==========================================================
This plan is:  test4
Complete:  False
Consistent:  True
Solution:  False
==========================================================
This plan is:  test5
Complete:  True
Consistent:  False
Solution:  False
==========================================================
This plan is:  test6
Complete:  False
Consistent:  True
Solution:  False
==========================================================
This plan is:  test9
Complete:  True
Consistent:  False
Solution:  False
==========================================================
This plan is:  test10
Complete:  False
Consistent:  True
Solution:  False
```

Test 12 – 18, and 20 - 22 are also test cases for this part. We will only provide the test cases but not the results to those test cases. We will use those tests together with tests that are not released to grade your submissions. The tests for linearization can be seen in the next section.

There are 46 test cases in total, including the ones provided to you. Each test case worth 1 autograder point. For each case, we will test:

- If solution exists:
  - consistent: 0.3 autograder pts
  - complete: 0.3 autograder pts
  - linearization: 0.4 autograder pts
- If no solution exists:
  - Consistent: 0.5 autograder pts
  - Complete: 0.5 autograder pts

Part B: Steps, Ordering Constraints and Causal Links (24 autograder points)

In this part, you will finish up some cases from student_tests. Please fill in the designated #TODO in **planner.py**. Please make sure the case is both consistent and complete after you finish them.

Steps

You will finish up the steps for two test cases: test2 and test8
You will fill in the preconditions and effects in the function run_test1_steps for test2, and you will need up design the entire steps in the function run_test2_steps for test8.
Hint: the preconditions and effects are relevant to the causal links.

Ordering Constraints

You will finish up the ordering constraints for two test cases: test7 and test11
You will fill in the descendants of the constraints in the function run_test1_ordering_constraints for test7, and you will need up design the entire constraints in the function run_test2_ordering_constraints for test11.
Hint: the causal links are relevant to the causal links.

Causal Links

You will finish up the causal links for two test cases: test3 and test19
You will fill in the predicates of the causal links in the function run_test1_causal _links for test3, and you will need up design the entire causal links in the function run_test2_causal _links for test19.
Hint: the causal links are relevant to the preconditions and effects.

You can use line 362 to 378 to test your cases. If you would like to test your code, but didn't have the functions in part A ready, you can submit your code to gradescope and it will let you know whether your implementation is complete/consistent or not.

Each test case worth 4 autograder pts. For each case, we will test:
Consistent and complete: 3 autograder pts
Linearization: 1autograder pt

After you finish this part and your solution is complete and consistent, you can use those test cases to check the linearization in part A. The correct solutions are documented in the corresponding functions.

Part C: Your Own Test Case (30 autograder points)

In this part, you will come up your own test case. Please finish it in the file student_test_case.py. We will evaluate your test case from the following aspect:

3 or more table slots are used: 2 autograder pts
4 or more blocks are used: 2 autograder pts
3 or more steps: 5 autograder pts
the precondition for the step "start" is correct: 5 autograder pts
the effect for the step "finish" is correct: 5 autograder pts
test case is complete: 5 autograder pts
test case is consistent: 5 autograder pts

However, if your test case doesn't make sense and look quite random, you may receive 0 points for this part. Please also make sure the "start" and "finish" have the correct ids.

Submission

Please submit the following two files to gradescope (via the link from canvas):

- planner.py
- student_test_case.py

The current autograder will provide instant feedback to you for the following parts:

- test cases provided in part A that are with answers
- Part B

The maximum points you will see is 30 autograder points for now. We will add more test cases and add the autograder for part C after the due date.