Homework 4: Planning (100 pts)

Due: Monday March 20 at 11:59 pm

Instructions

- Submit a pdf of your work to gradescope, making sure to mark all pages relevant to each outlined section on gradescope (including relevant code and output)
- Append all written pddl code and all graphplan output to the relevant portions of your pdf (no need to submit on canvas too)
- You have 3 late days, with a 10 point grade reduction per day. No submissions will be accepted after the third late day.

Overview

This assignment will give you practice with algorithms and representations used by planning agents. Topics covered include PDDL, graphplan, partial order planning, and planning with partial observability and nondeterminism. By the end, you should understand the design principles involved in planning agents, and be able to construct new planning agents to solve problems in new situations.

Background

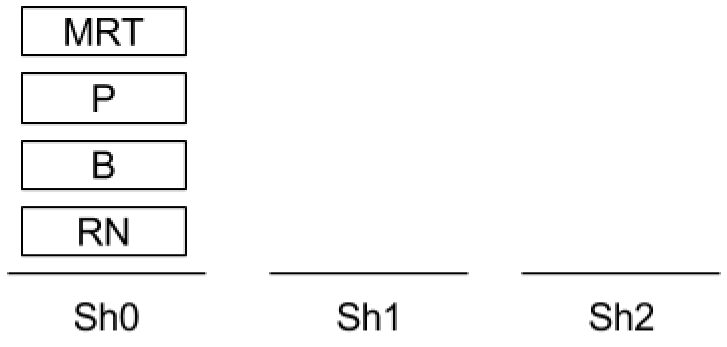
Before starting this assignment, you should read and be familiar with chapter 10 and section 11.3 of R & N (Russell and Norvig, 3rd edition).

Problem 1: Graphplan (60 pts)

You've got three bookshelves labeled Sh0, Sh1, and Sh2. Shelf Sh0 supports a stack of your favorite AI textbooks: *Artificial Intelligence: A Modern Approach (Russel and Norvig), Pattern Recognition and Machine Learning (Bishop), Probabilistic Reasoning in Intelligent Systems (Pearl),* and *Foundations of Machine Learning (Mohri, Rostamizadeh, Talwalker)*. Abbreviate these as RN, B, P, and MRT respectively. See Figure 1 for the setup.

Your objective is to read all of the books, and also reorganize Sh0 so that RN is on top (since you need it for EECS 492, of course) and the rest of the books are in the same order as before. To do this, you have two available actions: `Read` and `Move`. You can only move a book to a destination if both the book and the destination are clear (the book can't be underneath another book, and neither can the destination). You can only read a book if it is clear and you haven't read it before. The PDDL specification is as follows:

Figure 1: Bookshelves



```
Actions

Move(x, y, z):
    Precond: Clear(x), On(x,y), Clear(z), x ≠ z
    Effect: On(x,z), ¬On(x,y), Clear(y), ¬Clear(z)

Read(x):
    Precond: Book(x), Clear(x), Unread(x)
    Effect: ¬Unread(x)

Objects
Sh0, Sh1, Sh2, RN, B, P, MRT

Predicates
Clear, On, Unread

Initial
On(MRT,P), On(P,B), On(B,RN), On(RN,Sh0), Clear(MRT), Clear(Sh1),
Clear(Sh2), Book(RN), Book(P), Book(B), Book(MRT), Unread(MRT),
Unread(P), Unread(B), Unread(RN)

Goal
Clear(RN), On(RN,MRT), On(MRT,P), On(P,B), On(B,Sh0), ¬Unread(RN),
¬Unread(B), ¬Unread(P), ¬Unread(MRT)
```

A. (2 pts) In this world, we prohibit moving shelves. Why don't we need `Book(x)` as a precondition for `Move` to accomplish this?

B. (8 pts) Draw the planning graph for this problem up to level S1. Remember, the ordering of levels goes S0, A0, S1, A1, S2 … etc. See figures 10.8 and 10.10 in the textbook for examples. Do not draw the mutex lines (the graph is already cluttered enough). You may draw this on paper and scan it into the computer, or (preferably) you may draw it on the computer using a program like Google Drawings. **Note:** The examples in the book show lots of negated literals in S0. Since we will use the closed-world assumption in this problem, do not include negated literals in the graph unless they are necessary (necessary meaning that they are preconditions of non-persistence actions or effects of non-persistence actions). When reasoning about mutexes, only reason about literals and actions that are shown in the graph. Technically, any literal added by a non-persistence action is mutex with its negation - but there is no need for us to include all those cases unless they arise naturally in the graph from effects of other non-persistence actions. **Second note:** You should use the unique names assumption in this problem; that is, two objects with different names are assumed to be not equal. Use this assumption when satisfying the $x \neq z$ precondition of the move action; however, do not show the inequality conditions explicitly in your planning graph or count them in your mutexes. You can assume them in the background, and just not add actions like Move(MRT, P, MRT) to your planning graph.

C. Answer the following questions about your planning graph. The types of mutexes can be found in section 10.3.2 of the textbook or the lecture slides.

    i. (8 pts) How many distinct pairs of mutex actions (including persistence actions) are there in A0? List the pairs and categorize them. Use the notation `P[On(RN, Sh0)]` to denote persistence actions. State how many mutexes are due to each of *inconsistent effects*, *interference*, and *competing needs*. Remember, some mutexes can fall into more than one category.

    ii. (8 pts) How many distinct pairs of mutex literals are there in S1? List them and categorize them. State how many are due to each of *inconsistent support* and *complement*. Remember, some can fall into more than one category.

    iii. (3 pts) Will `clear(Sh1)` and `clear(Sh2)` ever be mutex? Why or why not?

    iv. (10 pts) List all non-persistence actions in level A1 (you don't have to draw them in your graph). How many are there? How many distinct pairs of them are mutex due to competing needs? Next to each action you list, give the number of competing needs mutexes that it is a part of with other non-persistence actions.

    v. (4 pts) In any planning graph, the number of actions in the action layers is always monotonically increasing. Explain why. (Note: understanding this may help you answer the previous question, for it is easy to forget some actions…)

D. (7 pts) In this part, you will use an existing graphplan library to solve the problem. You will need an installation of Lisp, as well as the Sensory Graphplan (SGP) package. Follow the instructions in the separate SGP/Lisp handout to set up the necessary software. Once you can successfully run graphplan on the demo problems in SGP, create your own planning file titled "bookshelf.pddl". Specify the PDDL under the domain "bookshelf-domain", and specify our particular instance of the problem under problem "bookshelf-organize". We have provided you with an outline bookshelf.pddl file doing most of this for you - fill in the rest of the file yourself. Place the file "bookshelf.pddl" in the "domains" subdirectory of the SGP package. To construct a plan, navigate to the SGP directory, start Lisp, and type the following commands in the Lisp command line:

```
(load "C:\\full\\path\\to\\file\\loader.lisp")  ;; For windows
(load "loader.lisp") ;; for Mac or Linux
(in-package :gp)
(load-gp)
(load-domains "bookshelf.pddl")
(dribble "bookshelf.out") ;; Send output to a file
(plan 'bookshelf-organize) ;; the apostrophe is necessary
(dribble) ;; Stop sending output to the file
```

Note that in Lisp, semicolons are used for comments, so don't type the semicolons in your commands. To debug your code, you can use the commands `(trace-gp top-level)` and `(trace-gp actions)` before invoking the planner to print more information about the planning graph, but do not include this extra output in your pdf, as it takes up a huge amount of space.

Paste your completed version of bookshelf.pddl in your pdf, as well as the output / generated plan.

E. Answer the following questions about the plan generated by SGP:
   i. (2 pts) Briefly describe in English what the plan does.
   ii. (1 pt) How many actions were in the plan?
   iii. (1 pt) How many levels of the planning graph were created?
   iv. (2 pts) Why is the number of levels less than the number of actions?

F. (4 pts) If we give the graphplan algorithm a goal that is impossible to achieve, will graphplan necessarily terminate? Why or why not?

Problem 2: Partial-Order Planning (22 pts)

As a dutiful college student, you want to accomplish several tasks before you go to bed. You want to have your homework done and be logged off your computer (so your roommate cannot mess with your social media while you are asleep). You also want to watch a relaxing movie on Netflix, but have decided that before watching Netflix, you must first read a chapter of your textbook and do your laundry. The PDDL specification of this problem is below:

```
Actions: Netflix, Read, DoLaundry, DoHomework, LogOff
Predicates: LoggedOn, BookRead, LaundryDone, MovieWatched, HaveBook,
BookRead, AtHome, HomeworkDone

Action Specifications
```

```
Netflix:
    Preconditions: LoggedOn, BookRead, LaundryDone
    Effects: MovieWatched

Read:
    Preconditions: HaveBook
    Effects: BookRead

DoLaundry:
    Preconditions: AtHome
    Effects: LaundryDone

DoHomework:
    Preconditions: LoggedOn
    Effects: HomeworkDone

LogOff:
    Preconditions: LoggedOn
    Effects: ¬LoggedOn
```

<u>`Initial State`</u>
`AtHome, LoggedOn, HaveBook`

<u>`Goal`</u>
`¬LoggedOn, HomeworkDone, MovieWatched`

A. (2 pts) In PDDL, define the `Start` and `Finish` actions that will be used by your partial order planner (easy, not a trick question).
B. (10 pts) A *partial plan* consists of several items: a list of *actions*, a set of *causal links*, a set of *orderings*, a set of *threats*, and a set of *unsatisfied preconditions*. A **causal link** is a tuple (action 1, precondition, action 2) which indicates that action 1 causes the given precondition of action 2 to be true. An **ordering** is a statement "action 1 < action 2" which says that action 1 must be done before action 2. Note that every causal link is paired with an ordering, but there can be orderings without causal links. A **threat** is a pair (action, causal link) which indicates that the given action threatens the given causal link by making the precondition in the causal link false. Threats must be resolved by adding ordering constraints: the threatening action must either come before the first action in the causal link or after the second action in the causal link. Always add the fewest constraints necessary to resolve the problem. Finally, an **unsatisfied precondition** is a tuple (precondition, action) for some precondition of an action listed in the partial plan that is not yet part of a causal link. A plan is *complete* if it has no threats and no unsatisfied preconditions.

Partial-order planning works by taking an initial partial plan and making incremental refinements until the plan is complete. A **refinement** to a plan is one of three things:

- The addition of an action to the plan to resolve one unsatisfied precondition (along with the causal link, necessary ordering constraints, and any new unsatisfied preconditions and threats)
- The addition of a causal link between existing actions to the plan to resolve one unsatisfied precondition (along with the necessary ordering constraints and any new threats)
- The addition of an ordering constraint to the plan to resolve a threat

For example, here are the first two partial plans in a sequence that solves the problem:

<u>P0: Initial plan</u>
Actions: (Start, Finish)
Causal Links: {}
Orderings: {Start < Finish}
Unsatisfied Preconditions: {(MovieWatched, Finish), (¬LoggedOn, Finish), (HomeworkDone, Finish)}
Threats: {}

<u>P1: Add action Netflix</u>
Actions: (Start, Finish, **Netflix**)
Causal Links: { **(Netflix, MovieWatched, Finish)**}
Orderings: {Start < Finish, **Start < Netflix, Netflix < Finish**}
Unsatisfied Preconditions: {(¬LoggedOn, Finish), (HomeworkDone, Finish), **(LoggedOn, Netflix), (BookRead, Netflix), (LaundryDone, Netflix)**}
Threats: {}

<u>Using the same notation, extend this sequence of partial plans until you have found a complete plan. At each step, put new additions to the plan in boldface. State the refinement you made next to each partial plan number, by saying either "add action X," "add causal link X," or "resolve threat X by adding Y."</u> Although in practice a partial order planner may get stuck and need to backtrack, do not backtrack here (the problem is simple enough that you should be able to find an uninterrupted sequence of refinements that achieve a complete plan).

<u>At the end, state how many causal links and orderings are in your final plan, as well as how many different threats were encountered along the way.</u>

C. (4 pts) Draw the partial-order diagram for the complete partial plan you found in part B. See figure 10.13 in the book or the lecture slides for examples.

D. Answer the following questions about the plan you found in part B, and about partial-order planning in general:

   i. (3 pts) In general, why do we need to use an indexed *list* of actions (or some similar data structure) rather than a *set* of actions in a partial-order planner? Give an example of when this would be necessary. (Hint: what happens if you add more than one of the same object to a set?)

   ii. (3 pts) A linearization of a partial plan is a totally ordered sequence of all the actions in the plan that obeys the ordering constraints of the partial plan. How many ways can your complete plan be linearized? Show your work.

Problem 3: Nondeterminism and Partial Observability (18 pts)

You are constructing a robot that can train your dog to sit. Unfortunately, dogs often do not do what they are expected to do, introducing lots of nondeterminism into the environment. Your agent can tell the dog to sit, but the dog may instead remain standing, or it may lie down. If it remains standing, the agent needs to repeat the sit command until the dog obeys. If the dog lies down, the agent needs to make the dog stand up again. If the dog sits on command, it should get a treat; however, sometimes a bird swoops down and steals the treat, in which case the agent needs to use another treat. We specify this in PDDL as follows. The "OR" in the effects marks nondeterminism.

```
Action Schemas:
SitCommand:
    Precond: DogStanding
    Effect: DogSitting, ¬DogStanding OR DogStanding OR DogLyingDown,
            ¬DogStanding

UpCommand:
    Precond: DogLyingDown
    Effect: DogStanding, ¬DogLyingDown OR DogLyingDown

GiveTreat:
    Precond: DogSitting
    Effect: HappyDog OR TreatStolen

GiveAnotherTreat:
    Precond: TreatStolen
    Effect: TreatStolen OR HappyDog, ¬TreatStolen

Initial State:
DogStanding
```

<u>`Goal:`</u>
`HappyDog`

A. (3 pts) Write pseudocode for a contingent planner that solves this problem (see the pseudocode in section 11.3.2 for an example).
B. (4 pts) Draw your solution from A as an And-Or graph (see figure 4.12 in the textbook). Loop back to repeated states. Only draw arrows for the actions that are part of your solution (the bolded parts of figure 4.12). This means that every Or-node in your And-Or graph will have only one action arrow emanating from it. And-nodes, of course, may have multiple arrows.
C. (4 pts) Now we will modify the problem to account for partial observability. We now have `dog`, `bird`, and `nothing` as explicit objects, an `InView(x)` predicate, a `ChangeView(x,y)` action, and percept schema for the state of the world. Initially nothing is in view. You do not know the state of the dog unless the dog is in view, nor do you know the state of the bird unless the bird is in view. Instead of allowing for the possibility of treats being stolen, you will only give a treat if the bird is not present. To accomplish this, we remove the `GiveAnotherTreat` action and replace it with the `PrepareTreat` action, which requires that the dog is sitting, and the effect of which is `TreatPrepared`. The preconditions of `GiveTreat` are now `TreatPrepared` and `¬BirdPresent`. We also add a `Wait` action, which requires `BirdPresent` and has no effect. <u>Modify the given PDDL to fit this new description by adding percept schema and adding/deleting/modifying actions. You may also need to modify the initial state.</u>
D. (4 pts) Write pseudocode for a new conditional plan based on the changes you made in part C.
E. (3 pts) What would you need to know to decide whether an online replanner or a contingency planner is a better choice in this scenario?