Homework 4: Planning

Arya Farahi

03/20/2017

1 Graphplan

You have 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 (Mo hri, 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 cannot be underneath another book, and neither can the destination). You can only read a book if it is clear and you have not read it before. The PDDL specification is as follows:

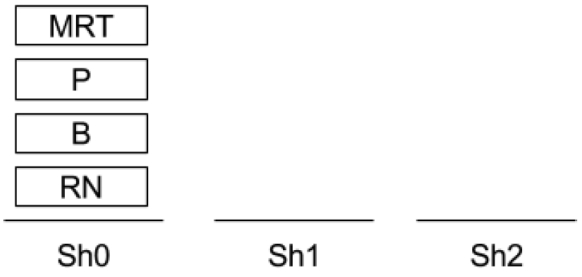


Figure 1: Bookshelves

```
Actions
    Move(x, y, z):
        Precond: Clear(x), On(x,y), Clear(z)
        Effect: On(x,z), NOT On(x,y), Clear(y), NOT Clear(z)
    Read(x):
        Precond: Book(x), Clear(x), Unread(x)
        Effect: NOT 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), NOT Unread(RN),
NOT Unread(B), NOT Unread(P), NOT Unread(MRT)
```

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

Because x should be clear and on something if we want to move it, and under close world assumption whatever is not explicitly mentioned we assume is false so it does not satisfy the precondition of Move action, `On(x,y)`, therefor by definition we cannot move it.

B. 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.

See 2.

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. How many distinct pairs of mutex actions (including persistence actions) are there in A0? List the pairs and categorize them. Use the notation `[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.

There are 6 pairs of mutex actions are there in A0.

`Read(MRT)` and `P[Unread(MRT)]`: Inconsistent effect, Interference

`Move(MRT,P,Sh1)` and `P[On(MRT,P)]`: Inconsistent effect, Interference

`Move(MRT,P,Sh2)` and `P[On(MRT,P)]`: Inconsistent effect, Interference

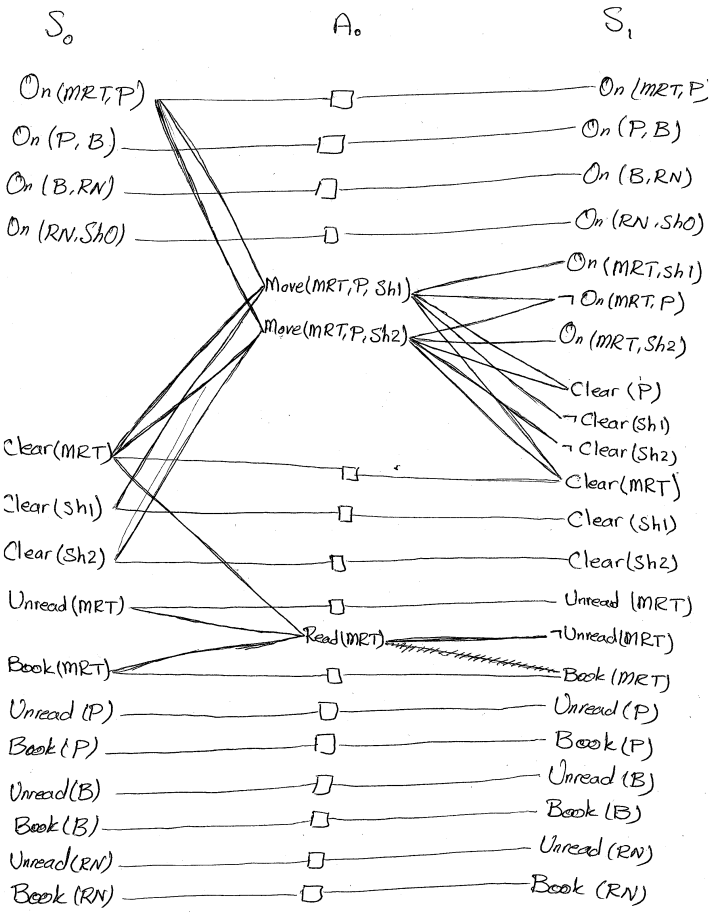


Figure 2: The planning graph

`Move(MRT,P,Sh1)` and `P[Clear(Sh1)]`: Inconsistent effect, Interference

`Move(MRT,P,Sh2)` and `P[Clear(Sh2)]`: Inconsistent effect, Interference

`Move(MRT,P,Sh1)` and `Move(MRT,P,Sh1)`: Interference

Inconsistent Effect : 5 pairs

Interference : 6 pairs

Competing Needs : 0 pair

ii. 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.

15 distinct pairs of mutex literals are there in S1. 4 complement, and 15 inconsistent support.

1. `On(MRT,P)` and ¬`On(MRT,P)` : inconsistent support, complementary
2. `Clear(Sh1)` and ¬`Clear(Sh1)` : inconsistent support, complementary
3. `Clear(Sh2)` and ¬`Clear(Sh2)` : inconsistent support, complementary
4. `Unread(MRT)` and ¬`Unread(MRT)` : inconsistent support, complementary
5. `On(MRT,P)` and `On(MRT,Sh1)` : inconsistent support

6. `On(MRT,P)` and `On(MRT,Sh2)` : inconsistent support
7. `Clear(Sh1)` and `On(MRT,Sh1)` : inconsistent support
8. `Clear(Sh2)` and `On(MRT,Sh2)` : inconsistent support
9. `Clear(P)` and `On(MRT,P)` : inconsistent support
10. `On(MRT,P)` and ¬`Clear(Sh1)` : inconsistent support
11. `On(MRT,P)` and ¬`Clear(Sh2)` : inconsistent support
12. `On(MRT,Sh1)` and `On(MRT,Sh2)` : inconsistent support
13. ¬`Clear(Sh1)` and ¬`Clear(Sh2)` : inconsistent support
14. `On(MRT,Sh1)` and ¬`Clear(Sh2)` : inconsistent support
15. `On(MRT,Sh2)` and ¬`Clear(Sh1)` : inconsistent support

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

The number of mutex literals monotonically decreases, and if a pair is not mutex on level N then in any level M where M > N it cannot become mutex. We can think of a scenario in which clear(sh1) and clear(sh2) persist all the time. So they cannot become mutex.

iv. 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.

There are 11 non-persistence actions in level A1:

`Read(MRT)`, `Read(P)`, `Move(MRT,P,Sh1)`, `Move(MRT,P,Sh2)`, `Move(P,B,MRT)`, `Move(P,B,Sh1)`, `Move(P,B,Sh2)`, `Move(MRT,Sh1,P)`, `Move(MRT,Sh2,P)`, `Move(MRT,Sh1,Sh2)`, `Move(MRT,Sh2,Sh1)`

There are 24 mutexes due to competing needs:

1. `Read(P)` and { `Move(MRT,P,Sh1)`, `Move(MRT,P,Sh2)` }
2. `Move(MRT,P,Sh1)` and { `Move(P,B,MRT)`, `Move(P,B,Sh1)`, `Move(P,B,Sh2)`, `Move(MRT,Sh1,P)`, `Move(MRT,Sh2,P)`, `Move(MRT,Sh1,Sh2)`, `Move(MRT,Sh2,Sh1)` }
3. `Move(MRT,P,Sh2)` and { `Move(P,B,MRT)`, `Move(P,B,Sh1)`, `Move(P,B,Sh2)`, `Move(MRT,Sh1,P)`, `Move(MRT,Sh2,P)`, `Move(MRT,Sh1,Sh2)`, `Move(MRT,Sh2,Sh1)` }
4. `Move(P,B,Sh1)` and { `Move(MRT,Sh1,P)`, `Move(MRT,Sh1,Sh2)` }
5. `Move(P,B,Sh2)` and { `Move(MRT,Sh2,P)`, `Move(MRT,Sh2,Sh1)` }
6. `Move(MRT,Sh1,P)` and { `Move(MRT,Sh2,P)`, `Move(MRT,Sh2,Sh1)` }
7. `Move(MRT,Sh2,P)` and { `Move(MRT,Sh1,Sh2)` }
8. `Move(MRT,Sh1,Sh2)` and { `Move(MRT,Sh2,Sh1)` }

v. 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 ⋯)

The previous actions shows up in the next level, and there are potentially new actions to be added. So the number of actions cannot be less than the number of actions of the previous level, therefor the number of actions in the action layers is always monotonically increasing.

D. 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
```

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

The bookshelf.pddl

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Bookshelf domain
(define (domain bookshelf-domain)
  (:requirements :strips)

  (:predicates
      (Clear ?x)
      (On ?x ?y)
      (Unread ?x)
      (Book ?x) )

  (:action Move
             :parameters (?x ?y ?z)
             :precondition (and  (Clear ?x)
                                 (On ?x ?y)
                                 (Clear ?z)
                           )
             :effect (and (On ?x ?z)
                          (not (On ?x ?y))
                          (Clear ?y)
                          (not (Clear ?z))
                     )
  )

  (:action Read
             :parameters (?x)
```

```
            :precondition (and  (Book ?x)
                                (Clear ?x)
                                (Unread ?x)
                          )
            :effect (not (Unread ?x))
  )

) ;;; Close parenthesis for domain definition

;;; Initial problem state
(define (problem bookshelf-organize)
  (:domain bookshelf-domain)
  (:objects Sh0 Sh1 Sh2 RN B P MRT)
  (:init (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 RN) (Unread P) (Unread B) (Unread MRT))
  (:goal (and (Clear RN) (On RN MRT) (On MRT P) (On P B)
              (On B Sh0) (not (Unread RN)) (not (Unread B))
              (not (Unread P)) (not (Unread MRT))))
  (:length (:serial 20) (:parallel 20)))
```

And the output is

```
dribbling to file "bookshelf.out"

NIL
GP(6): (plan 'bookshelf-organize)
Operators are ground at graph expansion time
Dynamic variable ordering is enabled
Least preconditional action ordering is enabled

Levels 1 2 3 4 5 6 7 8
(((READ DOMAINS::MRT)
  (DOMAINS::MOVE DOMAINS::MRT DOMAINS::P DOMAINS::SH1))
 ((READ DOMAINS::P) (DOMAINS::MOVE DOMAINS::P DOMAINS::B DOMAINS::MRT))
 ((READ DOMAINS::B) (DOMAINS::MOVE DOMAINS::B DOMAINS::RN DOMAINS::P))
 ((READ DOMAINS::RN)
  (DOMAINS::MOVE DOMAINS::RN DOMAINS::SH0 DOMAINS::SH2))
 ((DOMAINS::MOVE DOMAINS::B DOMAINS::P DOMAINS::SH0))
 ((DOMAINS::MOVE DOMAINS::P DOMAINS::MRT DOMAINS::B))
 ((DOMAINS::MOVE DOMAINS::MRT DOMAINS::SH1 DOMAINS::P))
 ((DOMAINS::MOVE DOMAINS::RN DOMAINS::SH2 DOMAINS::MRT)))
GP(7): (dribble)
~
```

E. Answer the following questions about the plan generated by SGP:

- **Briefly describe in English what the plan does.** It reads the book from the top pf Sh0 then move it to Sh1. Continues above process until hit the MRT. It reads the MRT and then move it to Sh2. Then it start with the book on top of Sh1 and move it to Sh0 and at the end it move MRT from Sh2 to Sh0. Note that for each move the user can first move the book then read it they do not have an order.

- **How many actions were in the plan?** 12 actions
- **How many levels of the planning graph were created?** 8 levels
- **Why is the number of levels less than the number of actions?** Because the user can read the book then move it or it can move it then read it. There is no order for those pair actions.

E. If we give the graphplan algorithm a goal that is impossible to achieve, will graphplan necessarily terminate? Why or why not?

No, if the goal is not reached then the graphplan continues, each level it add at least the previous layer actions and it will continue the same process until the goal literals achieved and they are not mutex, though if it cannot find the solution it continues to add actions and it never ends. However if we define a maximum level then once the algorithm hits the maximum level then it stops without reaching the goal. This is how the SGP/Lisp works/

2 Partial-Order Planning

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: Not LoggedOn

Initial State
     AtHome, LoggedOn, HaveBook

Goal
     Not LoggedOn, HomeworkDone, MovieWatched
```

A. In PDDL, define the `Start` and `Finish` actions that will be used by your partial order planner (easy, not a trick question).

`Start` = Initial State

Precond: -

Effect : `AtHome`, `LoggedOn`, `HaveBook`

`Finish` = Goal State

Precond : ¬`LoggedOn`, `HomeworkDone`, `MovieWatched`

Effect: -

B. 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:

```
P0: Initial plan

Actions: (Start, Finish)

Causal Links: {}

Orderings: {Start < Finish}

Unsatisfied Preconditions: {(MovieWatched, Finish), (¬LoggedOn,
Finish), (HomeworkDone, Finish)}

Threats: {}

---------------------------------------------------------------------

P1: Add action Netflix

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: {}

P2: Add causal link textbf(Start, LoggedOn, Netflix)

Actions: (Start, Finish, Netflix)

Causal Links: { (Netflix, MovieWatched, Finish), textbf(Start, LoggedOn, Netflix) }

Orderings: {Start < Finish, Start < Netflix, Netflix < Finish }

Unsatisfied Preconditions: { (¬LoggedOn, Finish), (HomeworkDone, Finish), (BookRead, Netflix), (LaundryDone, Netflix) }

Threats: {}

P3: Add action **DoHomework**

Actions: (Start, Finish, Netflix, **DoHomework**)

Causal Links: {(Netflix, MovieWatched, Finish), (Start, LoggedOn, Netflix), **(DoHomework, HomeWorkDone, Finish)**}

Orderings: {Start < Finish, Start < Netflix, Netflix < Finish, **Start < DoHomework**, **DoHomework < Finish** }

Unsatisfied Preconditions: {(¬LoggedOn,Finish), (BookRead,Netflix), (LaundryDone,Netflix), **(LoggedOn,DoHomework)**}

Threats: {}

P4: Add causal link **(Start, LoggedOn, DoHomework)**

Actions: (Start, Finish, Netflix, DoHomework)

Causal Links: {(Netflix, MovieWatched, Finish), (Start, LoggedOn, Netflix), (DoHomework, HomeWorkDone, Finish), **(Start, LoggedOn, DoHomework)**}

Orderings: {Start < Finish, Start < Netflix, Netflix < Finish, Start < DoHomework, DoHomework < Finish}

Unsatisfied Preconditions: {(¬LoggedOn,Finish), (BookRead,Netflix), (LaundryDone,Netflix)}

Threats: {}

P5: Add action **LogOff**

Actions: (Start, Finish, Netflix, DoHomework, **LogOff**)

Causal Links: {(Netflix, MovieWatched, Finish), (Start, LoggedOn, Netflix),
(DoHomework, HomeWorkDone, Finish), (Start, LoggedOn, DoHomework)
(LogOff, ¬LoggedOn, Finish)}

Orderings: {Start < Finish, Start < Netflix, Netflix < Finish, Start < DoHomework,
DoHomework < Finish, **Start < LogOff**, **LogOff < Finish** }

Unsatisfied Preconditions: {(BookRead,Netflix), (LaundryDone,Netflix), **(LoggedOn,LogOff)**}

Threats: {(**LogOff, (Start, LoggedOn, Netflix)**),
(**LogOff, (Start, LoggedOn, DoHomework)**)}

P6: Resolve threat LogOff, (Start, LoggedOn, Netflix) by adding LogOff > Netflix

Actions: (Start, Finish, Netflix, DoHomework, LogOff)

Causal Links: {(Netflix, MovieWatched, Finish), (Start, LoggedOn, Netflix),
(DoHomework, HomeWorkDone, Finish), (Start, LoggedOn, DoHomework)
(LogOff, ¬LoggedOn, Finish)}

Orderings: {Start < Finish, Start < Netflix, Netflix < Finish, Start < DoHomework,
DoHomework < Finish, Start < LogOff, LogOff < Finish, **LogOff > Netflix** }

Unsatisfied Preconditions: {(BookRead,Netflix), (LaundryDone,Netflix), (LoggedOn,LogOff)}

Threats: {(LogOff, (Start, LoggedOn, DoHomework))}

P7: Resolve threat LogOff, (Start, LoggedOn, DoHomework) by adding LogOff > DoHomework

Actions: (Start, Finish, Netflix, DoHomework, LogOff)

Causal Links: {(Netflix, MovieWatched, Finish), (Start, LoggedOn, Netflix),
(DoHomework, HomeWorkDone, Finish), (Start, LoggedOn, DoHomework)
(LogOff, ¬LoggedOn, Finish)}

Orderings: {Start < Finish, Start < Netflix, Netflix < Finish, Start < DoHomework,
DoHomework < Finish, Start < LogOff, LogOff < Finish, LogOff > Netflix, **LogOff > DoHomeWork** }

Unsatisfied Preconditions: {(BookRead,Netflix), (LaundryDone,Netflix), (LoggedOn,LogOff)}

Threats: {}

P8: Add action **Read**

Actions: (Start, Finish, Netflix, DoHomework, LogOff, **Read**)

Causal Links: {(Netflix, MovieWatched, Finish), (Start, LoggedOn, Netflix),
(DoHomework, HomeWorkDone, Finish), (Start, LoggedOn, DoHomework)
(LogOff, ¬LoggedOn, Finish), **(Read, BookRead, Netflix)** }

Orderings: {Start < Finish, Start < Netflix, Netflix < Finish, Start < DoHomework,
DoHomework < Finish, Start < LogOff, LogOff < Finish, LogOff > Netflix, LogOff > DoHomeWork,
Start < Read, **Read < Netflix** }

Unsatisfied Preconditions: {(LaundryDone,Netflix), (LoggedOn,LogOff), **(HaveBook,Read)**}

Threats: {}

P9: Add action **DoLaundry**

Actions: (Start, Finish, Netflix, DoHomework, LogOff, Read, **DoLaundry**)

Causal Links: { (Netflix, MovieWatched, Finish), (Start, LoggedOn, Netflix),
(DoHomework, HomeWorkDone, Finish), (Start, LoggedOn, DoHomework)
(LogOff, ¬LoggedOn, Finish), (Read, BookRead, Netflix), **(DoLaundry, LaundryDone, Netflix)**}

Orderings: {Start < Finish, Start < Netflix, Netflix < Finish, Start < DoHomework,
DoHomework < Finish, Start < LogOff, LogOff < Finish, LogOff > Netflix, LogOff > DoHomeWork,
Start < Read, Read < Netflix, **DoLaundry > Start**, **DoLaundry < Netflix** }

Unsatisfied Preconditions: {(LoggedOn,LogOff), (HaveBook,Read), **(AtHome,DoLaundry)**}

Threats: {}

P10: Add causal link **(Start, HaveBook, Read)**

Actions: (Start, Finish, Netflix, DoHomework, LogOff, Read, DoLaundry)

Causal Links: {(Netflix, MovieWatched, Finish), (Start, LoggedOn, Netflix),
(DoHomework, HomeWorkDone, Finish), (Start, LoggedOn, DoHomework)
(LogOff, ¬LoggedOn, Finish), (Read, BookRead, Netflix), (DoLaundry, LaundryDone, Netflix),
(Start, HaveBook, DoLaundry)}

Orderings: {Start < Finish, Start < Netflix, Netflix < Finish, Start < DoHomework,
DoHomework < Finish, Start < LogOff, LogOff < Finish, LogOff > Netflix, LogOff > DoHomeWork,
Start < Read, Read < Netflix, DoLaundry > Start, DoLaundry < Netflix }

Unsatisfied Preconditions: {(LoggedOn,LogOff), (AtHome,DoLaundry)}

Threats: {}

P11: Add causal link **(Start, AtHome, DoLaundry)**

Actions: (Start, Finish, Netflix, DoHomework, LogOff, Read, DoLaundry)

Causal Links: {(Netflix, MovieWatched, Finish), (Start, LoggedOn, Netflix),
(DoHomework, HomeWorkDone, Finish), (Start, LoggedOn, DoHomework)
(LogOff, ¬LoggedOn, Finish), (Read, BookRead, Netflix), (DoLaundry, LaundryDone, Netflix),
(Start, HaveBook, DoLaundry), **(Start, AtHome, DoLaundry)** }

Orderings: {Start < Finish, Start < Netflix, Netflix < Finish, Start < DoHomework,
DoHomework < Finish, Start < LogOff, LogOff < Finish, LogOff > Netflix, LogOff > DoHomeWork,
Start < Read, Read < Netflix, DoLaundry > Start, DoLaundry < Netflix }

Unsatisfied Preconditions: {(LoggedOn,LogOff)}

Threats: {}

P12: Add causal link **(Start, LoggedOn, LogOff)**

Actions: (Start, Finish, Netflix, DoHomework, LogOff, Read, DoLaundry)

Causal Links: {(Netflix, MovieWatched, Finish), (Start, LoggedOn, Netflix),
(DoHomework, HomeWorkDone, Finish), (Start, LoggedOn, DoHomework)
(LogOff, ¬LoggedOn, Finish), (Read, BookRead, Netflix), (DoLaundry, LaundryDone, Netflix),
(Start, HaveBook, DoLaundry), (Start, AtHome, DoLaundry), **(Start, LoggedOn, LogOff)**}

Orderings: {Start < Finish, Start < Netflix, Netflix < Finish, Start < DoHomework,
DoHomework < Finish, Start < LogOff, LogOff < Finish, LogOff > Netflix, LogOff > DoHomeWork,
Start < Read, Read < Netflix, DoLaundry > Start, DoLaundry < Netflix }

Unsatisfied Preconditions: {}

Threats: {}

C. 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.

see 3.

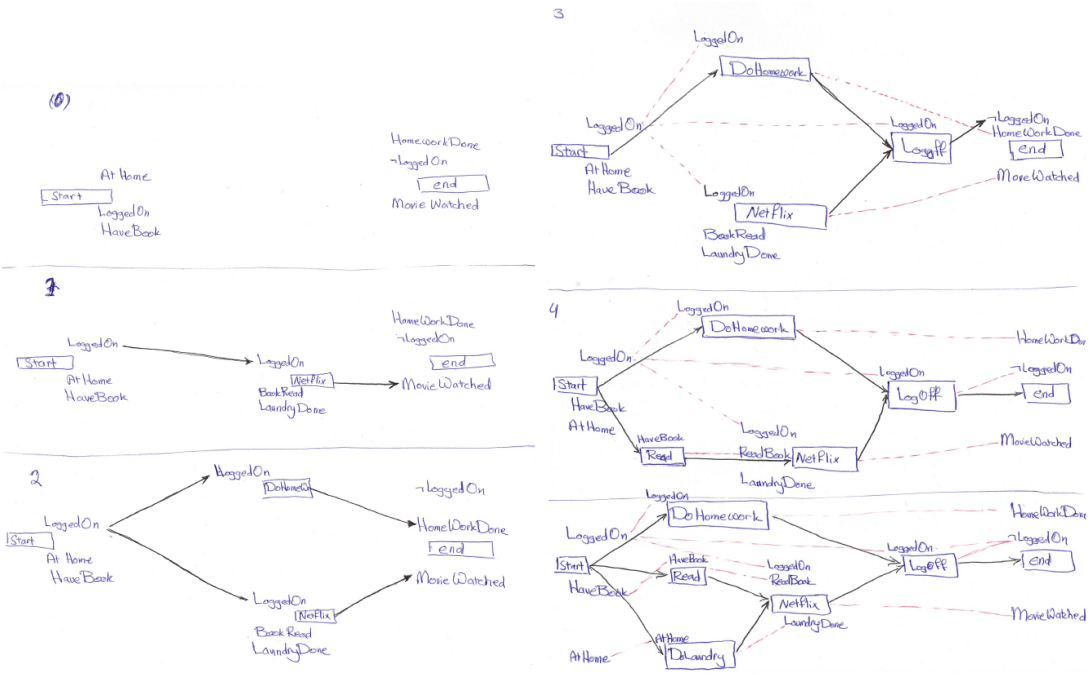


Figure 3: The partial-order diagram

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

- **(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?)** Because one action may appears more than once, and the pre-codintions and the effects maybe different. For example action Move(x,y,z) can have different outcomes due to the different x, y, and z. So we need an indexed list of actions to distinguish between similar actions with different preconditions and outcomes.
- **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.**

  8 different ways:

  1. { Start, DoHomework, Read, DoLaundry, Netflix, LogOff, End }
  2. { Start, Read, DoHomework, DoLaundry, Netflix, LogOff, End }
  3. { Start, Read, DoLaundry, DoHomework, Netflix, LogOff, End }
  4. { Start, Read, DoLaundry, Netflix, DoHomework, LogOff, End }
  5. { Start, DoHomework, DoLaundry, Read, Netflix, LogOff, End }
  6. { Start, DoLaundry, DoHomework, Read, Netflix, LogOff, End }
  7. { Start, DoLaundry, Read, DoHomework, Netflix, LogOff, End }
  8. { Start, DoLaundry, Read, Netflix, DoHomework, LogOff, End }

3 Nondeterminism and Partial Observability

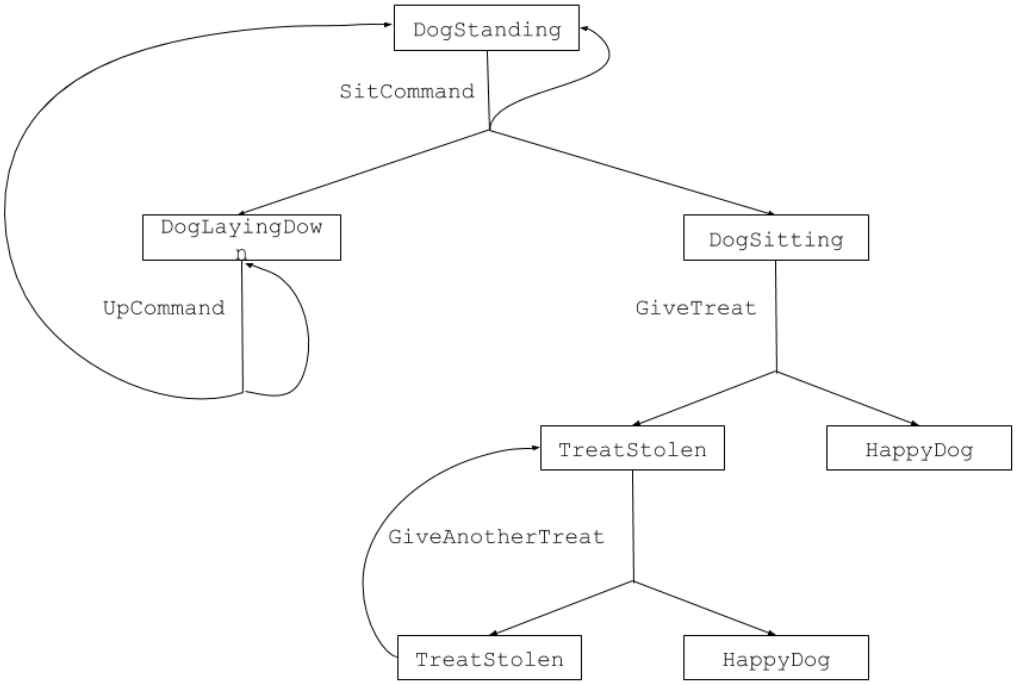
You are constructing a robot that can train your dog to sit. Unfortunately, dogs often do not do what they are expected to do, introducing lots of nondeterminism into the environment. Your agent can tell the dog to sit, but the dog may instead remain standing, or it may lie down. If it remains standing, 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, Not DogStanding OR DogStanding OR DogLyingDown, Not DogStanding

UpCommand:
    Precond: DogLyingDown
    Effect: DogStanding, Not DogLyingDown OR DogLyingDown

GiveTreat:
    Precond: DogSitting
    Effect: HappyDog OR TreatStolen

GiveAnotherTreat:
    Precond: TreatStolen
    Effect: TreatStolen OR HappyDog, Not TreatStolen

Initial State :
    DogStanding

Goal:
    HappyDog
```

A. Write pseudocode for a contingent planner that solves this problem (see the pseudocode in section 11.3.2 for an example).

```
SitCommand
while ¬DogSitting {
     if DogLyingDown then {
          while ¬DogStanding {
              UpCommand
          }
     }
     SitCommand
}
GiveTreat
While TreatStolen
      GiveAnotherTreat
```

B. 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. 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. 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.

```
Action Schemas:

ChangeView(x,y):
    Precond: InView(x) XOR ¬InView(x)
    Effect: InView(y) XOR ¬InView(y)

SitCommand:
    Precond: DogStanding
    Effect: DogSitting, ¬DogStanding OR DogStanding OR DogLyingDown, ¬DogStanding

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

PrepareTreat:
    Precond: DogSitting
    Effect: TreatPrepared

Wait:
    Precond: BirdPresent, TreatPrepared
    Effect: -

GiveTreat:
    Precond: TreatPrepared, ¬BirdPresent
    Effect: HappyDog

Initial State :
    InView(Nothing)

Goal:
    HappyDog
```

D. Write pseudocode for a new conditional plan based on the changes you made in part C.

```
ChangeView(Nothing, Dog)
while ¬DogStanding {
     UpCommand
}
SitCommand
while ¬DogSitting {
     if DogLyingDown then {
          while ¬DogStanding {
              UpCommand
          }
     }
     SitCommand
}
PrepareTreat
ChangeView(Dog, Bird)
InView(Bird)
While BirdPresent {
      Wait
      InView(Bird)
}
GiveTreat
```

E. What would you need to know to decide whether an online replanner or a contingency planner is a better choice in this scenario?

Here is a very simple scenario, there is a Bird which may steal the treat though one can imagine other scenarios, like the dog does not like the treat and it do not eat it or become happy. The agent may ran out of treat. The dog may see a cat and start to chase it. Modeling and considering all these scenarios in advance would be hard and not efficient specially if the rate of these event are rare, then there is no point in modeling all scenarios, and online planing can become relevant. When something other than predefined scenarios happened then the robot should re-plan.