CPS721: Assignment 4

Due: Tuesday, November 28, 2023, 9pm
Total Marks: 100 (worth 4% of course mark)
You MUST work in groups of 2 or 3

Late Policy: The penalty for submitting even one minute late is 10%. Assignments are not accepted more than 24 hours late.

Clarifications and Questions: Please use the discussion forum on the D2L site to ask questions as they come up. These will be monitored regularly. Clarifications will be made there as needed. A Frequently Asked Questions Page will also be created. You may also email your questions to your instructor, but please check the D2L forum and frequently asked questions first.

Collaboration Policy: You can only discuss this assignment with your group partners or with your CPS721 instructor. By submitting this assignment, you acknowledge that you have read and understood the course policy on collaboration as stated in the CPS721 course management form.

PROLOG Instructions: When you write your rules in PROLOG, you are not allowed to use ";" (disjunction), "!" (cut), and "->" (if-then). You are only allowed to use ";" to get additional responses when interacting with PROLOG from the command line. Note that this is equivalent to using the "More" button in the ECLiPSe GUI.

We will be using ECLiPSE Prolog release 6 to mark the assignments. If you run any other version of PROLOG, it is your responsibility to check that it also runs in ECLiPSE Prolog release 6.

SUBMISSION INSTRUCTIONS: You should submit ONE zip file called `assignment5.zip` containing 4 files:

`laundry.pl`	`laundry_report.txt`
`philosophers.pl`	`philosophers_report.txt`

These files have been given to you and you should fill them out using the format described. Your submission should not include any other files. If you submit a `.rar`, `.tar`, `.7zip`, or other compression format aside from `.zip`, you will lose marks. All submissions should be made on D2L. Submissions by email will not be accepted. As long as you submit your assignment with the file name `assignment5.zip` your group will be able to submit multiple times as it will overwrite an earlier submission. You do not have to inform anyone if you do. The time stamp of the last submission will be used to determine the submission time. Do not submit multiple `zip` files with different names. If you do, we will use the last submitted one, but you may lose marks.

If you write your code on a Windows machine, make sure the files are saved on plain text and are readable on Linux machines. Ensure your PROLOG code does not contain any extra binary symbols and that they can be compiled by ECLiPSE Prolog release 6.

READ THIS INFORMATION BEFORE BEGINNING THE ASSIGNMENT

In this assignment, you will be creating programs to solve two planning problems. For each, there are three provided files: the file containing the generic planner (this is shared between the two problems), the file with initial state information, and the main submission file in which you will be defining the operators and declarative heuristics. Below, we provide more information on each of these. We also provide some information about how we will test your programs.

Main File

The main files for question 1 and 2 are `laundry.pl` and `philosophers.pl`, respectively. To run your programs you should load these files. You might notice these files have several Prolog features that you haven't seen before. First, in the `dynamic` section, you will see rules of the form

```
:- dynamic clean/2.
```

This line tells Prolog to allow the predicate `clean` to be defined in different non-consecutive lines in your program. It is necessary to allow the initial state of the planning problem to be stored in a different file. You do not need to edit these sections of the files.

Next, notice the `planner` and `init` sections. They contain rules of the form

```
:- [planner].
```

This line tells Prolog to load in the file `planner.pl`, which contains the generic planner used for both problems. You do not need to edit the `planner` section. Notice that a similar line is used to load in the initial state in the `init` section. You may find it useful to create other initial states to help test or debug your program, for example, by creating different situations and checking if different actions are possible and how they affect the state (more info on the initial state files below). However, when doing your final tests, you should use the initial state we have given you.

Next, you will see the `goal_states` section. Here, multiple different goals have been defined for you. Notice that the predicate uses a slightly different notation than we covered in class. Here, we use `goal_state(ID, S)`, where `S` is the situation as defined previously, and `ID` is the number of the corresponding goal. This is to allow you to easily jump between goals when testing your program, by calling `solve_problem` with a different goal ID, as you will see in the planner file. You may find it useful to add additional goals for testing, especially when starting out. You can do so in this section.

The remaining sections then provide space to define helpers as needed, your action precondition axioms, your successor state axioms, and your declarative heuristics, as required by the different problems.

Planner File

The planner can be found in the file `planner.pl`. This file is shared between the two problems you are to solve. You do NOT have to edit it, but you should look at it to understand how to run the planner. In it, you will see mostly familiar predicates, though with slight changes in some cases.

The main predicate you will call to run the planner is

```
solve_problem(Mode, GoalID, Bound, Plan)
```

Just as in class, the `Bound` variable is a maximum on the length of the plan to find, and `Plan` is the found plan. The other two arguments are input arguments that are intended to simplify the process of testing your program. `GoalID` defines which goal to use. This will allow you to easily test with the different goal conditions defined in the main files. The `Mode` argument allows you to specify which "mode" to run the planner in. If it is set to `heuristic`, then the planner will use declarative heuristics in the form of the `useless` predicate to cut off search to avoid unnecessary work. If it is set to `regular`, then the declarative heuristics are ignored, and the standard reachable definition is used (*ie.* without pruning). Notice that the `reachable` predicate has been modified accordingly to allow for these different usages.

For example, if you call `solve_problem(regular, 2, 5, Plan)`, you are asking the planner to find a plan of no more than 5 actions, such that the goal with ID 2 is satisfied, and it should do so without using declarative heuristics for pruning.

You do NOT have to submit the planner file as part of your submission as we will automatically include it when testing.

Initial State File

For each problem, there is a different file, `laundryInit.pl` and `philosophersInit.pl`, which identifies the fluents and auxiliary predicates hold in the initial situation. You can add different initial situations by creating new files, and changing the `init` section in the main files to load in your desired initial state. This may be useful when testing out your axioms or debugging your program. However, you should complete your final tests with the given initial states. You do NOT have to submit the initial state files as part of your submission. This includes both any initial states you create and the ones given to you.

Testing

We will be testing your programs in two ways. First, we will try your axioms on different initial states to ensure that they correctly compute preconditions and effect. Second, we will also run your complete planner to ensure the whole system works together. Importantly, we may run your program on DIFFERENT start and goal states. Thus, your declarative heuristics should not be specific to those. In other words, make sure your declarative heuristics are general enough so that they can be applicable to solving any planning instance of the planning problem with arbitrary constant names, and different combinations of initial/goal states.

Finally, Remember to also submit a report of your experiments for both questions (part **c** of both questions). Describe their your computer system and what did you observe when running your program with various initial/goal states.

1 The Laundry Robot Problem (50 Marks)

In the not-too-distant future, robot assistants will be helping humans both at homes and at work. We hope that giving instructions to them will be easy. Instead of programming every detailed step of every motion that the robot has to do, the future robots will simply take verbal orders such as "do laundry" when they are given the piles of dirty clothes. In this part, the task is to figure out how the robot will have to solve this planning problem. For the sake of generality, subsequently we talk about containers. In this assignment, the dresser, washer, dryer, cupboard, and hamper, are all considered containers.

We consider below terms (actions) and predicates (fluents) that are general enough to deal with any collections of clothes, washing and drying machines. In particular, we consider the following ten actions (they are **terms**).

- $fetch(O, C)$: fetch object O from container C. After fetching, the robot is holding O, and it is no longer in C. This action is only possible if you are not currently holding anything.
- $putAway(O, C)$: put away object O into container C. After doing this action, the robot is no longer holding O, but it is now in C. This action is only possible if you are currently holding O.
- $addSoap(P, W)$: after adding soap P to washer W, W has soap. This action is only possible if the robot is holding P, and W does not currently have soap.
- $addSoftener(T, W)$: add fabric softener T to washer W. This action is only possible if the robot is holding T and W does not currently have softener.
- $removeLint(D)$: remove lint from dryer D. After doing this action, D no longer has lint. It is only possible if D currently has lint and the robot is not holding anything.
- $washClothes(C, W)$: wash clothes C in washer W. In the situation resulting from execution of this action, C is clean and wet, and W has neither soap nor softener. This action is only possible if C is in W, C is not clean, and W has soap and softener.
- $dryClothes(C, D)$: dry clothes C in dryer D. In the situation resulting from execution of this action, C is not wet, D has lint. The action is only possible if C is in D, C is wet, and D does not have lint.
- $fold(C)$: fold clothes C. This action is only possible if C is not folded, clothes are clean and dry, and the robot is not currently holding anything.
- $wear(C)$: wear clothes C. After doing this action, C is neither clean nor folded. This action is only possible if C is folded.
- $move(C, F, T)$: move clothes C from a location F to a different location T. After doing this action, C is in T, but it is no longer in F. The robot can do this action if it is not currently holding anything, C is in F, both F and T are containers. The container can be a dresser, or a hamper, or a washer, or a dryer that is empty.

We consider the following fluents (they are **predicates** with the last argument S).

- $in(O, C, S)$: object O is inside C in situation S.
- $holding(O, S)$: the robot is holding object O in situation S after fetching O from somewhere. The robot is no longer holding an object, if the robot put it away, or if the object is added to a washer machine, in the case that the object is a soap or a fabric softener.
- $hasSoap(W, S)$: washer W has soap in situation S.
- $hasSoftener(W, S)$: washer W has fabric softener in situation S.
- $hasLint(D, S)$: dryer D has lint in situation S.

- $clean(C, S)$: clothes C are clean in situation S.
- $wet(C, S)$: clothes C are wet in situation S (after washing, and before drying).
- $folded(C, S)$: clothes C are folded in situation S.

Finally, there are auxiliary predicates that do not have a situational argument: $cupboard(B)$, $washer(W)$, $soap(P)$, $softener(T)$, $dryer(D)$, $clothes(C)$, $hamper(H)$, $container(N)$. These predicates state that the object in question (*ie.* the argument) is of the type given by the predicate name. As stated above, several of these object types are also containers. Rules stating as such have already been added for you to laundry.pl.

You should now complete the following:

a. [39 marks] Write the following rules in the corresponding sections in the given file **laundry.pl**

- Write precondition axioms for all actions in your domain. Recall that to avoid potential problems with negation in Prolog, you should not start bodies of your rules with negated predicates. Make sure that all variables in a predicate are instantiated by constants before you apply negation to the predicate that mentions these variables.
- Write successor-state axioms that characterize how the truth value of all fluents change from the current situation S to the next situation $[A|S]$. Recall that you will need two types of rules for each fluent:

  a) rules that characterize when a fluent becomes true in the next situation as a result of the last action

  b) rules that characterize when a fluent remains true in the next situation, unless the most recent action changes it to false.

  Note that when you write successor state axioms, you can sometimes start bodies of rules with negation of a predicate, e.g., with negation of equality. This can help your program work a bit more efficiently.

- Test your program with the given initial state in `laundryInit.pl` and goal state 1 given in the `laundry.pl` file. You should run the planner in "regular" move (*ie.* without declarative heuristics). You should set the upper bound on the number of actions to **6**.

  It should take about 10-30 seconds or less to solve the problem depending on the computer you run it on. As stated above, you may want to use your own custom made initial states or goal states to debug your rules to ensure they correctly identify when actions are possible and that the effects are computed correctly. Avoid queries where you set the situation argument to a variable (*ie.* `S`) since they amount to solving the unbounded planning problem. Keep a copy of your session with Prolog and add it to `laundry_report.txt` as described below in part (c).

b. [6 marks] You will now write declarative heuristics that the planner can use for pruning when run in "heuristic" mode. Recall that the predicate `useless(A,ListOfActions)` is true if an action A is useless given the list of previously performed actions. If this predicate is defined using proper rules, then it helps speed-up the search that your program is doing to find a list of actions that solves the problem. This predicate provides (domain dependent) declarative heuristic information about the planning problems that your program solves. The more inventive you are when you implement this predicate, the less search will be required to solve the planning problems. However, any implementation of rules that define this predicate should **not** use any information related to the specific initial or goal situations. Your rules should be general enough to work with any initial and goal states. When you write rules that define this predicate use common sense properties of the application domain. Write your rules for the predicate `useless` in the file **laundry.pl**: it must include the program that you created in the previous part of this assignment. You have to write at least 4 different rules. Make sure they make sense. **Put comments beside each explaining what they are doing**.

Once you have the rules for the predicate `useless(A,List)`, solve the given planning problems with goal states 2 and 3 when using the planner in "heuristic" mode. In goal state 2, we simply ask the robot to make sure clothes are dry. This problem can be solved with 8 actions in about 30sec or less. (Without the declarative heuristics, computing a plan for this 2nd goal can take a few minutes). In the 3rd planning problem, we ask the robot additionally to figure out what has to be done to store dry folded clothes in a dresser. When you solve this 3rd planning problem look for a plan that has no more then 10 actions. If your rules are creative enough, solving this problem should be fast, e.g., in less than 2 min. Request several plans using "more" button (or using ";" command). Once, you solved it, try to solve the 2nd or 3rd planning problems without using rules for the predicate `useless(A,L)`: call `solve_problem` in `regular` mode. Warning: your program may take a few minutes to solve this version (depending on how fast is your computer). Your interaction with Prolog will be used for part (c).

c. [5 marks] Write a **brief** report in `laundry_report.txt` that includes all queries that you have submitted to your program (with or without heuristics) and how much time your program spent to find several plans when you added more heuristics. Discuss briefly your results and explain what you have observed, including a description of the effect of the declarative heuristics. This part should only be a paragraph or two in length. Include a description of what computer you ran it on (*ie.* mention manufacturer, CPU speed, and RAM).

2 The Philosophers Problem (50 Marks)

In computer science, the dining philosophers problem is an example often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them. In this assignment, it is used as an instance of a planning problem. The following rendering of this problem is adapted from *Wikipedia*. Let n silent philosophers sit at a circular table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers. Each philosopher can be in one of the following states: thinking, waiting, or eating. However, a philosopher can only eat spaghetti when they have both the forks on their left and right. Each fork can be held by only one philosopher at a time and so a philosopher can use the fork only if it's not being used by another philosopher. After a philosopher finishes eating, they need to put down both the forks so they become available to others. A philosopher can grab the fork on their right or the one on their left as they become available, but they can't start eating before getting both of them. Eating is not limited by the amount of spaghetti left: assume an infinite supply. However, at most 2 philosophers can eat spaghetti from the bowl at the same time. If 2 philosophers are currently eating, and the third one is trying to eat, then they will not be able to eat and will have to wait.

You have to solve a simplified version of this planning problem using an approach considered in class. In this version, only one philosopher acts at any one time, and the actions are given as follows (recall that actions are **terms** and so they can be only arguments of predicates or equality):

- $pickUp(P, F)$: a philosopher P picks up an adjacent fork F provided this fork is currently available in situation S. This action is possible for P if either F is the fork on their left, or if F is the fork on their right. A philosopher P cannot pick up forks which they are not adjacent to. When P picks up F, the fork F is no longer (freely) available because P now has it in their hand.

- $putDown(P, F)$: this action will undo the effects of the pick up action. A philosopher P can execute this action if they currently hold F in their hand.

- $tryToEat(P)$: this action is possible if a philosopher P has two different forks in their hands.

To represent features of this domain that can change, it is sufficient to introduce the following predicates with situation argument (fluents):

- $available(F, S)$: a fork F is available in S. The fork becomes available if someone puts it down, and it is not available when someone picks it up.

- $has(P, F, S)$: a philosopher P holds a fork F in S. This fluent becomes true after doing $pickUp(P, F)$ action, and it remains true unless the $putDown(P, F)$ action is executed.

- $thinking(P, S)$: a philosopher P is thinking in situation S. This is true if they only have one fork and put down the only fork in their possession, or if P was already thinking and did not pick up any fork.

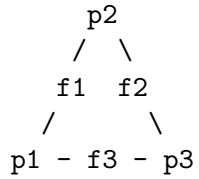
- $eating(P, S)$: a philosopher P is eating in situation S if their latest action was a successful attempt to eat (there were no 2 other philosophers eating at that time), or if they were eating before and have not put down one of the two forks that they are holding. If a waiting philosopher is trying to eat when 2 other philosophers are already eating, they cannot eat, but they keep waiting.

- $waiting(P, S)$: a philosopher P is waiting in situation S if they are neither thinking nor eating. P switches from thinking to waiting by picking up a fork. They switch from eating to waiting by putting down one of their two forks. The philosopher P remains waiting if they were waiting in the previous situation and could not start eating after attempting to eat (because spaghetti in the bowl was consumed by 2 other philosophers), or if they were waiting in the previous situation with at least one fork in their hands and they did not put down the last fork. Recall that if any waiting philosopher has only one fork, they start thinking after they put down the fork.

- *happy*(P, S): a philosopher P becomes happy after eating in the previous situation as soon as they put down at least one of their forks. If the philosopher P is happy, they remain happy no matter what they or the other philosophers do subsequently.

There are also several auxiliary predicates that are not fluents: their truth values do not change after executing any of the actions. In particular, *philosopher*(P) means that P is a philosopher, $fork(F)$ means that F is a fork, $left(P, L)$ means that the philosopher L is seating to the left from P, $right(P, R)$ means that the philosopher R is seating to the right from P, *between*$(P1, F, P2)$ means that a fork F is located in between the philosophers $P1$ and $P2$. Note that these predicates do not change from one situation to another.

The initial situation we have given you corresponds to the following diagram:



where `p1`, `p2`, and `p3` are the three philosophers, and `f1`, `f2`, and `f3` are the three forks.

You should now complete the following:

a. [**33 marks**] As in Question 1, write the precondition and successor state axioms for this problem in `philosophers.pl`. Recall again that your axioms should not encode whether the planner should apply the action, just if it CAN apply the action. Again, we may mark your program using other initial and goal states, e.g., with more than 3 philosophers.

You should test your rules using goal states 1 through 4 in regular mode and using a bound of 8. These should take only a few seconds, and at most a minute. If it takes more, that suggests there is a bug. You can either rely on Prolog's tracer to debug your program, or you can try queries testing intermediate states of computation that your program does (you can use *poss* or fluents to formulate testing queries), or you can try another simpler goal state. Testing your program with simple queries can help you to locate a bug faster than trying to trace it with a debugger. Request several plans using ";" command. We encourage you to add other queries to test your program but you will not get marks for them. Be sure to keep your Prolog interaction for part (c).

b. [**12 marks**] You will now write declarative heuristics that the planner can use for pruning when run in "heuristic" mode. Write as many meaningful `useless` rules as you can to implement this predicate and add them to `philosophers.pl`. Think about useless repetitions that should be avoided, and about order of execution (i.e., use common sense properties of the application domain). Your rules should never use any constants mentioned in the initial or goal states because while your rules have to be domain specific, they should be independent of a particular instance of the planning problem that you are solving.

Once you have wrote rules for `useless(A,ListOfPastActions)`, test them on same planning problems as above, or any similar simple planning problems. You should also test on goal state 5 with a bound of 13. **Warning**: solving an instance of this planning problem without declarative heuristics can take 5-7 hours of CPU time (or longer if you work in a lab, or if your CPU is slow). However, if you have a clever set of heuristics, then solving this or similar problems should take at most a few minutes. In other words, well-thought heuristics should provide 100 times speed-up. You have to try several different goal states to deserve full mark for this part. You will document this in part c.

c. [**5 marks**] Write a **brief** report in `philosophers_report.txt` that includes all queries that you have submitted to your program (with and without heuristics) and how much time your program spent to find several plans. Discuss briefly your results and explain what you have observed, including a description of the effect of the declarative heuristics. This part should only be a paragraph or 2 in length. Include a description of what computer you ran it on (*ie.* mention manufacturer, CPU speed, and RAM). Make sure to highlight any additional goal states you tested on.