CPS721: Assignment 5

Due: Monday, November 24, 2025, 9pm
Total Marks: 100 (worth 4% of course mark)
In groups of no more than 3

Late Policy: Late submissions are penalized by $3^n\%$ when they are n days late.

Clarifications and Questions: Please use the discussion forum on the D2L site to ask questions. These will be monitored regularly. A Frequently Asked Questions (FAQ) page will also be created. Only ask questions by email if it is specific to your submission. Any questions asked with less than 12 hours left before the deadline are not likely to be answered.

PROLOG Instructions. Your code must run on ECLiPSe 7.1. If you cannot install this on your own machine, test it on the lab machines or the department moon servers.

You are NOT allowed to use ";" (disjunction), "!" (cut), "->" (if-then), `setof`, `bagof`, `findall`, or any other language features and library predicates that are not covered in class. Similarly, you should only use "=" and "is" for equality, and "not" for negation. Do NOT use "=:=", "\=", "+\", or other such commands. You will be penalized if you use these commands/features. You are only allowed to use ";" to get additional responses when interacting with PROLOG from the command line (this is equivalent to using the "More" button in the ECLiPSe GUI).

You ARE allowed to use the built-in `member` or `append` methods if they are useful to you.

Academic Integrity and Collaboration Policy: Any submission you make must be the work of your group and your group alone. You are also not allowed to post course materials (including assignments, slides, etc.) anywhere on the web, and you are also prohibited from using generative AI systems or online "help" websites for completing assignments.

Submission Instructions: You should submit ONE zip file called `assignment5.zip` on D2L. It should contain the following files:

```
assignment5_submission_info.txt
pathfinding.pl                  pathfinding_report.txt
dishwashing.pl                  dishwashing_report.txt
```

All these files have been given to you and you should fill them out using the format described. Ensure your names appear in all files — especially `assignment5_submission_info.txt` — and your answers appear in the file sections as requested. Do NOT edit, delete, or add any lines that begin with "`%%%%% SECTION:`". These are used by the auto-grader to locate the relevant code. You will lose marks if you edit such lines.

Your submission should not include any files other than those above. You will be penalized for using compression formats other than `.zip`. Submissions by email will not be accepted.

You may submit your solution multiple times. Doing so will simply overwrite your previous submission. We will always mark the last submission your group makes.

READ THIS INFORMATION BEFORE BEGINNING THE ASSIGNMENT

In this assignment, you will be creating programs to solve several planning problems. For each, there are multiple provided files:

- A file containing the generic planner (this is shared between all problems). You should NOT edit this file.
- File(s) with initial state information. You should NOT edit these files, but we encourage you to create your own and use them for testing as detailed below.
- The main submission file in which you will be defining the axioms and declarative heuristics.
- A report file in which you will be documenting your results.

Below, we provide more information on each of these files. We also provide some information about how we will test your programs.

Main File

The main files for questions 1 and 2 are `pathfinding.pl` and `dishwashing.pl`, respectively. To run your programs you should load these files. They have been set up so that they will also load the additional files as needed. You will notice that in the `setup` section of these files, there are rules of the form `:- dynamic agentLoc/2.` This line tells Prolog to allow the predicate `agentLoc` to be defined in different non-consecutive lines in your program. This is necessary to allow the initial state of the planning problem to be stored in a different file than your axioms. You should NOT edit these sections of the files.

Next, you will see the `goal_states` section. Here, multiple different goals have been defined for you. Notice that the predicate is defined as `goal_state(ID, S)`, where `S` is the situation, and `ID` is the number of the corresponding goal. This is slightly different than the `goal_state` predicate introduced in class. This newer version will allow you to easily jump between goals when testing your program, by calling `solve_problem` with a different goal ID. We will describe this in detail below. You may find it useful to add additional goals for testing, especially when starting out. You can do so in this section. Please include this section in the submission, but its contents will be ignored.

The remaining sections then provide space to define your action precondition axioms, your successor state axioms, and your declarative heuristics.

Planner File

The planner can be found in the file `planner.pl`. You should NOT edit this file, 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)
```

Here, `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 should NOT submit the planner file as part of your submission as we will automatically include it when testing. We will ignore your submitted file if you do.

Initial State File

For each problem, we have provided multiple initial state files which identify the fluents and auxiliary predicates which hold in different initial situations. These are loaded in the `init` sections of the main files using a line like `:- [dishwashingInit1].` To change between initial states, simply comment out the unused initial states and uncomment the one you want to use. Please include this section in the submission, but its contents will be ignored. Note that it is a good idea to restart eclipse when switching initial states to avoid having fluents for multiple initial states open at the same time.

Changing the initial state is a good way to test your program in a variety of situations. You can add your own by creating new files that use this format. You then merely need to modify the `init` section in the main files to load in your desired initial state. This is an especially good idea when testing out your preconditions and axioms or debugging your program. However, you should complete your final tests with the given initial states. You should NOT submit any initial state files as part of your submission.

Self-Testing Your Axioms

Debugging axioms can be challenging. Don't forget you can always directly check if an action is possible by using `poss` as a query. For example, to check that an action `act` is applicable in some situation given by `[c, b, a]`, you can always query for `poss(act, [c, b, a])`. Similarly, if you want to check whether a fluent `f(5, S)` is true in a particular situation `[c, b, a]`, you can always query for `f(5, [c, b, a])`. Doing so is a much more effective approach for debugging than just testing your axioms by calling `solve_problem`, as the latter gives very little feedback for why failure is occurring. It will also be much faster, since queries like `poss(act, [c, b, a])` and `f(5, [c, b, a])` should complete almost instantaneously.

The initial states and goals given are of varying difficulty. Some are not feasible to solve without declarative heuristics. Thus, it is useful for you to first confirm your program is working on the easier ones, before even attempting the harder ones. To that end, you may find it useful to create your own initial states and goal states that will help you better understand if your program is working.

Marking Your Program

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 effects. Second, we will also run your complete planner to ensure the whole system works together. Importantly, we will run your program on DIFFERENT initial and goal states than those given. Thus, your declarative heuristics should not be specific to those provided. 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. We have tried to give different combinations of initial and goal states so you better understand the set of possible planning problems you will be tested on. More details is given on this topic in each of the questions.

When grading your programs using `solve_problem`, we will not require you to get all solutions by calling "More" or ";". You should try to get additional solutions when testing your own program, but we will not consider it for marking. Instead, we will take plans outputted by your system, and verify it using our solution to ensure it is a valid plan, and is the shortest of all possible plans.

For your declarative heuristics, you don't need to get the absolute fastest performance possible. Full marks will be awarded if a reasonable speedup is seen, and the returned plans are still valid (and optimal). Different parts of the assignment may also be marked manually.

Additional Tips

- Remember that your precondition axioms not only define when an action is applicable, but are used to match and thereby *find* the set of actions applicable in any particular state. Thus, you should avoid starting the bodies of your precondition axioms with negated predicates, due to the way negation works in Prolog with variables that are not yet set.

- It *is* ok to use negation at the beginning of your successor state axioms. For example, we saw several examples of the form `not A = action(P1, P2)`. This works because the successor state axioms are always called with a given list of actions (*ie.* the situation variable is set to a specific list). It can also help your programs efficiency.

- If you call "More" after getting a first solution, it is ok if you get duplicate solutions. However, duplicates are largely avoidable in this assignment. In particular, recall that duplicates happen in Prolog because the same answer can be found along different backchaining paths. In the case of planning, happens either because there are multiple precondition axioms that make a move applicable, or because there is overlap between cases where the successor-state axioms can hold. Thus, you can avoid duplicates by ensuring that only a single precondition axiom can make a move applicable, and by ensuring the successor-state axioms are mutually exclusive.
  Having said that, the most important thing is to make sure you get the right plans, so you don't need to get rid of duplicates. But you can consider addressing above if you are already convinced of the correctness of your program and want to make it more efficient. This is not required, however.

- Your successor state axioms should never "look into the future" when referring to a different fluent. That is, you should never have statements like
  `fluent1([A | S]) :- ..., fluent2([A | S]), ...`. To see why, think about each state/situation along the plan as being a different time step (*ie.* a second apart). The incorrect rule above would thus be saying `fluent` holds at some time step t if `fluent2` holds at time step t. This doesn't work temporally, since the idea with planning is that the previous step (at time step $t-1$) and action "causes" the next step. The result can lead to logical errors or infinite loops. However, it is ok to have rules like `fluent1([A | S]) :- ..., fluent2([S]), ...` since this is saying `fluent1` holds at time step t if `fluent2` is true in the state before action `A` is applied.

- You CANNOT introduce additional fluents or actions, or add anything to the initial states.

- For precondition axioms, remember to make sure you do not apply negation to unset variables. That is, make sure all variables are assigned some constant first.

- For persistence successor-state axioms, you almost always want to put the recursive call last. This is because that is the most "expensive" part of the call, since it may require multiple recursive calls. In contrast, checking that the last action is not the same as a given action is usually much cheaper.

- It is generally best to ensure your axioms work correctly in regular mode before testing your declarative heuristics. This will help you debug issues by helping you isolate which part of your program is causing issues. When working on your declarative heuristics, it is often best to first focus on problems of "medium difficulty" (*ie.* that take 10-30 seconds) in regular mode. This is because such tasks take long enough that you can see appreciable gains using the declarative heuristics, while not taking so long that it is hard to try a test out different heuristics. Once you make gains there, you can test your heuristics (and possibly add more complex heuristics) on harder problems.

1 Multi-Agent Pathfinding [35 marks]

In multi-agent pathfinding, we not only have one agent trying to get from its start location to a goal location in the map, but potentially an arbitrary number of agents. We are thus planning routes for all of them simultaneously, such that those routes don't lead to any collisions between agents or with obstacles in the environment. Here, we will consider doing such pathfinding on a grid map like the one below:

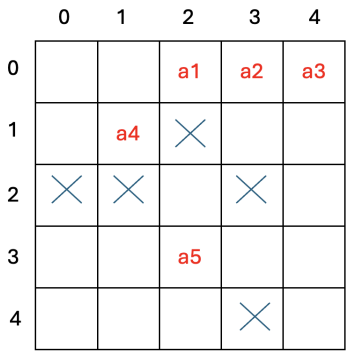

Figure 1: An example multi-agent pathfinding map.

In the figure, there are 5 agents, named `a1`, `a2`, `a3`, `a4`, and `a5` on a map. Their starting locations are shown. For example, `a1` starts at location (0, 2), meaning it is at row 1, column 2. Some subset of the agents may have their own goal location. For example, `a1` may want to get to location (1, 3) and `a2` may want to get to (0, 0). Importantly, the agents cannot be in the same location at the same time (*ie.* they can't collide), and they also cannot run into the walls (shown as Xs). Note, for simplicity, we will assume that only a single agent moves at one time (*ie.* the agents do not move concurrently).

For this problem, we will only have a single fluent:

- `agentLoc(Agent, Row, Col, S)`: `Agent` is at row `Row` and column `Col` in situation `S`.

We will also have the following auxiliary predicates that do not have a situational argument:

- `numCols(X)` defines that the number of columns for the map is `X`.
- `numRows(X)` defines that the number of rows for the map is `X`.
- `wallAt(Row, Col)` defines that there is a wall at the location `Row`, `Column`.
- `agent(A)` defines that `A` is an agent.

Facts based on these auxiliary predicates are included in the initial state files. See `pathfindingInit1.pl` and `pathfindingInit2.pl` as examples.

We also only need a single action for this domain:

- `move(Agent, Row1, Col1, Row2, Col2)`: this action means `Agent` moves from the location of `Row1, Col1` to `Row2, Col2`. Agents can only move one row or column at a time, and cannot move diagonally. This means it can only move one step up, one step right, one step down, or one step left. They also cannot move into a location where there is another agent or a wall. An agent cannot move off the map, or move to the same location that they are already at (*ie.* `move(A, 1, 2, 1, 2)` is not a valid action).

For this problem, you will write the precondition and successor-state axioms for this problem, as well as declarative heuristics. Notice that we have given you 7 goals for the first initial state and 3 for the second initial state. For all of these, you should manually solve the problem yourself so you know what the optimal solution is. In addition, the following will hold for a working solution:

- Goals 11-13 should take less than a second to solve on regular mode.

- Goal 14 will take a few seconds on regular mode
- Goal 15 will take roughly 10-30 seconds in regular mode.
- Goal 16 will take several minutes in regular mode.
- Goal 17 will take several minutes in heuristic mode
- Goal 21 (for initial state 2) make take 10-30 seconds in regular mode.
- Goal 22 and 23 are only solvable in a reasonable time with declarative heuristics.

You should now complete the following tasks:

a. [15 marks] Write precondition axioms for all actions. You should put your precondition axioms and any helpers for them in the `precondition_axioms_pathfinding` section. Finally, remember that your precondition axioms should only capture if you *can* apply an action, not if you *should* apply the action. Just let the planner decide if it should apply the action or add such information as declarative heuristics as required for part (c).

b. [10 marks] 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]`. These should be added to the section `successor_state_axioms_pathfinding`. Recall that you will need two types of rules for each fluent:

1. rules that characterize when a fluent becomes true in the next situation as a result of the last action.
2. rules that characterize when a fluent remains true in the next situation, unless the most recent action changes it to false.

c. [5 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. 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 of the initial and goal states, as well as similar ones involving specifying target locations for different subsets of agents. 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 `pathfinding.pl` in the section called `declarative_heuristics_pathfinding`. You should include at least 3 declarative heuristics, but you are encouraged to include as many as possible to speed up your search. **Put comments beside each declarative heuristic explaining what they are doing**.

We note that many of the declarative heuristics considered in class worked by avoiding sequences of actions that go back and forth between two states. For example, this was done when avoiding doing `climbOnBox` immediately after `climbOffBox` in the bananas problem from the lectures. There are not many such heuristics in this problem. Instead, you should consider other types of useless actions. This can include pruning actions that shouldn't be done more than once, or eliminating some of the "symmetries" that arise in this problem due to the fact that certain sets of actions can be done in a different order, but still end up in the same state. For example, if we do `move(a1, 3, 3, 3, 4)` and then `move(a3, 1, 1, 2, 1)` in situation `S`, the resulting state is the same as first doing `move(a3, 1, 1, 2, 1)` and then doing `move(a1, 3, 3, 3, 4)`. This is because these actions are *independent* in the sense that they have no effect on each others preconditions or effects.

Using declarative heuristics to prune such cases may not allow the planner to find all possible plans when calling "More" or ";", but it can dramatically speed up the planning process when searching for a single solution. You are encouraged to use such declarative heuristics for this problem. However, you should ensure your heuristics do not make it impossible to find solutions (or prune optimal solutions) in the types of initial state/goal state pairs we have given you.

d. [5 marks] Document the results of testing your planner (*ie.* calling `solve_problem`) on this problem and put them in the file `pathfinding_tests.txt`. Fill out the sections with the following details:

- `cpu_details` : include information about the processor (mainly speed), amount of RAM, and operator system you ran your tests on.
- `summary` : summarize your results in 5-10 sentences. In particular, describe which states you tested on, your timing results, and how much speedup you saw when using declarative heuristics. Report any other interesting behaviour you saw.
- `log` : show the log of your tests (*ie.* copy the interaction) including the runtime and the output plan, when using both the `regular` and `heuristic` modes. You do not have to find more than one plan per problem (though you may want to do that yourself when testing).

You tests should be performed on at least goals `11`, `12`, `13`, `14`, `15`, and `21`. You may wish to include tests on other goals as well, but this is not required.

2 Dishwashing [65 marks]

In the future, people may have household robots to help them with chores. For this question, you will build a planning system for dishwashing that could be used by such a robot.

We consider a simple version of this problem described as follows. A robot with two arms is standing by the sink. There are *plates* and *glasses* on the counter. The robot should pick up the dirty dishes, wash them, and put them in the *dish rack* to dry. The robot has two utensils (called *scrubbers*) they can use for cleaning the dishes. These are a *sponge* and a *brush*. Plates can only be cleaned with a sponge and glasses can only be cleaned with a brush. To clean a dirty dish, it should be scrubbed with a soapy scrubber and then rinsed to get rid of the soap and dirt. Note that we are assuming that a scrubber can be used to scrub an arbitrary number of dishes without needing more soap or becoming dirty.

To describe this planning problem, we will use the following auxiliary predicates:

- `place(X)` states that `X` is a place that items can be located at. `X` will either be `counter` or `dish_rack`.
- `scrubber(X)` states that `X` is a utensil for cleaning dishes. `X` will either be a `sponge` or a `brush`
- `glassware(X)` states that `X` is glassware (*ie.* a glass cup or glass bottle).
- `plate(X)` states that `X` is a plate.
- `dish(X)` states that `X` is a plate or glassware.
- `item(X)` states that `X` is an `item` the robot can hold. This includes glassware, plates, and scrubbers.

The definition of `place`, `scrubber`, and `item` are given in the section `aux_dishwashing` in `dishwashing.pl`. Do not change this section. Notice that `glassware` and `plate` are used to assign the types to objects in the initial state files, while `dish` and `item` are useful predicates that can be used to refer to different groups of objects.

The fluents in this problem are as follows:

- `holding(X, S)` holds if the robot is holding item `X` in `S`.
- `numHolding(C, S)` holds if the robot is holding `C` different items in `S`. Since the robot has 2 hands, this is at most 2 items.
- `faucetOn(S)` holds if the faucet is on in `S`.
- `loc(X, P, S)` holds if the location of item `X` in situation `S` is `P`. Here, `P` is a `place`. If the robot is holding `X` in `S`, then no such `loc` fluent will hold for `X` in `S`.
- `wet(X, S)` holds if the item `X` is wet in `S`.
- `dirty(X, S)` holds if the item `X` is dirty in `S`. Recall that scrubbers never get dirty.
- `soapy(X, S)` holds if the `X` is soapy in `S`. `X` can be any item.

We can now define our actions as follows:

- `pickUp(X, P)`: picks up item `X` from place `P`. After applying this action, `X` will no longer be at `P` and it will be held by the robot. Note that the robot must not already be holding two items to pickup another.
- `putDown(X, P)`: puts down item `X` to place `P`. Only applicable if `X` is being held by the robot. After applying this action, `X` is no longer held by the robot.

- `turnOnFaucet`: turns on the faucet. This action is only applicable if the faucet is off, and the robot has a free hand (*ie.* it is holding at most one item) to turn on the faucet.
- `turnOffFaucet`: turns off the faucet. This action is only applicable if the faucet is on, and the robot has a free hand to turn off the faucet.
- `addSoap(X)`: adds soap to `X`. This action is only applicable if `X` is a scrubber that is held by the robot. The robot must have a free hand to add soap to the scrubber. The scrubber will be soapy after adding soap.
- `scrub(X, Y)`: scrubs dish `X` using scrubber `Y`. Both `X` and `Y` must be held by the robot to apply this action. If `X` is glassware, it can only be scrubbed with a brush. If `X` is a plate, it can only be scrubbed with a sponge. If `Y` is soapy, then `X` will be soapy after it is scrubbed. If `X` is a dirty dish, it will remain dirty after it is scrubbed (*ie.* the food stays on it until the dirt and soap is rinsed off). Note that scrubbing a dish with a scrubber that has no soap on it will have no effect on the dish.
- `rinse(X)`: rinses item `X`. The faucet must be on and `X` must currently be held by the robot in order to rinse it. If `X` is a scrubber that is soapy, it will no longer be soapy after rinsing it. If `X` is a dish that is both soapy and dirty (*ie.* because it was scrubbed using a soapy scrubber), then it will be clean and not soapy after being rinsed. In all cases, an item will be wet after being rinsed.

As in the first question, you will write the precondition and successor-state axioms for this problem, as well as declarative heuristics. We have provided 3 initial states and a variety of goal states for testing:

- Goal states 11, 12, and 13 for initial state 1 have an optimal solution length of 2, 4, and 6 steps, respectively. These should all be solved in a matter of seconds.
- Goal state 14 for initial state 1 has a solution length of 8. It will take 10-30 seconds in regular mode.
- Goal state 15 for initial state 1 has a solution length of 10. It will take 30-120 seconds on heuristic mode, and quite a while on regular mode.
- Goal state 21 for initial state 2 has a solution length of 6, and will take 10-40 seconds in regular mode to solve.
- Goal state 22 for initial state 2 takes 11 steps, and will take 5-15 minutes when using declarative heuristics.
- Goal state 31 for initial state 3 takes 10 steps and will take 2-10 minutes with declarative heuristics.

It is a good idea to start by convincing yourself these are the actual plan lengths (at least for the easy problems), before starting to write your axioms. This will help ensure you fully understand the tasks given.

You should now complete the following tasks:

a. [20 marks] Write precondition axioms for all actions in your domain in the `precondition_axioms_dishwashing` section of `dishwashing.pl`. See the description of part (a) of Question 1 for useful suggestions when defining your preconditions.

b. [25 marks] Write successor-state axioms in section `successor_state_axioms_dishwashing` of `dishwashing.pl`. See the description of part (b) of Question 1 for useful suggestions when defining your successor-state axioms.

c. [15 marks] You will now write declarative heuristics for the planner to use when solving this problem in "heuristic" mode. You should write at least 10 rules.

When creating your heuristics, you can assume that the initial and goal states they will be used on will be "reasonable". That is, they should allow your planner to handle problems that involve getting all (or some subset) of a given set of dishes cleaned and into the dish rack. The initial states and goal states will never include states that would not be helpful to achieving this objective. For example, you can assume that the dishes on the counter always start dirty, you will never need to put dirty or soapy dishes in the dish rack, clean and wet dishes should always end up in the dish rack, and soapy and wet dishes should never be put on the counter. However, your declarative heuristics should never result in the planner finding suboptimal solutions.

You may wish to consider heuristics of different types such as those discussed in part (c) of Question 1. There are also other possible types of heuristics. Try to be creative about what you can do with the useless predicate. However, you are also not obligated to cover all different kinds of heuristics, just that you include at least 10 rules. It is also worth testing them individually (by removing and adding them), to see if they are actually helping at all. Please see part (c) of Question 1 for further tips on generating declarative heuristics.

Remember to put a comment beside each declarative heuristic indicating what it is doing.

d. [5 marks] Document the results of testing your planner (*ie.* calling `solve_problem`) on this problem and put them in the file `dishwashing_tests.txt`. Please see part (d) of Question 1 for full details on the expected documentation.

You tests should be performed on at least goals 11, 12, 13, 14, and 21. You may wish to include tests on other goals as well, but this is not required.