cps721: Assignment 5 (100 points).

Due date: upload zip file on *Monday, November 30th, 2020***, before 21:00. Read Page 6.**

Students who will submit their ZIP file before *Friday, November 27th, 21:00*, will get 10% extra marks.

You must work in groups of TWO or THREE. You cannot work alone.

YOU MAY NOT USE ";" AND "!" AND "–>" IN YOUR PROLOG RULES.

You can discuss this assignment only with your CPS721 group partners or with the CPS721 instructor. By submitting this assignment you acknowledge that you read and understood the course Policy on Collaboration in homework assignments stated in the CPS721 course management form.

This assignment will exercise what you have learned about problem solving and planning using Prolog. More specifically, in this assignment, you are asked to solve planning problems using the situations and fluents approach. For each of questions 1 and 2, you should use a file containing the rules for `solve_problem`, `reachable`, `max_length(List,Bound)` as explained in class, and then add your own precondition and successor state axioms using terms and predicates specified in the assignment. (Download the planner from the Assignments page).[1]

1 (50 points). Consider a machine shop domain where the goal is to manufacture certain parts. There are the following actions: they require different types of parts processing. (Recall that actions are **terms**: they can be only arguments of predicates or equality.)

- $drill(X)$: drill a part X, provided X is free (i.e., it is not fastened to any other part).
- $shape(X)$: shape a part X, this action will undo the effects of actions $drill(X)$ and $paint(X)$. It is always possible to execute this action.
- $bolt(X, Y)$: fasten a part X to a part Y by bolts provided both X and Y are drilled and not fastened to anything.
- $paint(X, Colour)$: paint a part X using $Colour$, provided X is free and $Colour$ is available (see below).

To represent features of this domain, it is sufficient to introduce the following predicates with situation argument (fluents). Recall that fluents are **predicates** with situation as the last argument, where situations are represented by lists of actions that have been already executed.

- $connected(X, Y, S)$: parts X and Y are connected in S if they are fastened by bolts. Connected parts remain connected after any action unless the agent shapes one of them.
- $painted(X, C, S)$: a part X has colour C in S. A painted part remains painted unless it is shaped, drilled or painted in a different colour.
- $shaped(X, S)$: a part X is shaped in a situation S. A shaped part remains shaped no matter what actions are subsequently executed.
- $free(X, S)$: a part X is free, i.e., it is not fastened to anything in S.
- $drilled(X, S)$, a part X is drilled in S. A drilled part remains drilled unless it is shaped.

Finally, we use the predicate $available(C)$ to describe all available colours: the truth value of this predicate does not change from one situation to another. Before you can solve planning problems in this domain, you have to write precondition axioms and successor state axioms. Write all your Prolog rules in the file **factory.pl** provided for you.

[1]This is not directly related to the assignment, but out of curiosity, you might wish to watch a 4min episode about the crow who can solve a difficult problem solving task that requires 8 steps: https://www.youtube.com/watch?v=AVaITA7eBZE This episode is taken from the BBC 2 series "Inside the Animal Mind".

Consider the following initial and goal situations:

```
/* In the initial state, both parts are free and we have 2 colours. */
free(a,[]). free(b,[]).
available(green).  available(blue).

/* In the goal state, both parts are shaped, painted and fastened. */
goal_state(S) :-   connected(a,b,S),
                   painted(a,green,S), painted(b,blue,S),
                   shaped(a,S), shaped(b,S).
```

1. Download the file **initFactory.pl** with descriptions of your initial and goal states from the Assignments folder on D2L. Keep both this file and the file **factory.pl** in the same folder. When ECLiPSe Prolog compiles the file **factory.pl** it loads the file **initFactory.pl** automatically. Do not copy content of **initFactory.pl** into your file **factory.pl** because a TA will use other initial and goal states to test your program in **factory.pl** Solve this simple planning problem using the planner with the upper bound 7 on the number of actions. It should take less than a minute for your planner to solve this problem (depending on CPU where you run it). If it takes longer than 1min, your program most probably has a bug. If you would like to debug your program, you can try a simpler goal state (e.g., when only the part a gets shaped and painted, this needs 2 actions only). Request several plans using ";" command. Explain why there are so many different solutions and discuss briefly your results (including computation time) in the file **factory.txt**

2. The predicate `useless(A,ListOfPastActions)` is true if an action A is useless given the list of previously performed actions. The predicate `useless(A,ListOfPastActions)` helps to solve the planning problem by providing *declarative heuristics* (advises) to the planner. If this predicate is correctly defined using a few rules (one rule per action A), then it helps to speed-up the search that your program is doing to find a list of actions that solves a planning problem. Write at least 5 rules that define this predicate: think about useless repetitions you would like to avoid, and about order of execution (i.e., use common sense properties of the application domain). Your rules have to be general enough to be applicable to any factory domain as outlined above, in other words, they have to help in solving a planning problem for any instance (i.e., any initial and goal states), not just those which are given to you. Once you have wrote rules for `useless(A,ListOfPastActions)`, test them using same planning problem as above (that takes 7 actions to solve), or any similar simple problems. You have to use the following rule for the predicate **reachable(S,Plan)** when you do these tests:

   ```
   reachable(S2, [M | ListOfActions]) :- reachable(S1,ListOfActions),
                               legal_move(S2,M,S1),
                               not useless(M,ListOfActions).
   ```

   Notice if solving planning problems takes more time or less time. Include your testing results and a brief discussion of your results in the report **factory.txt**

Handing in solutions: (a) An electronic copy of your file **factory.pl** and a copy of **factory.txt** must be included in your **zip** archive. The file **factory.txt** must include a copy of session(s) with Prolog, showing the queries you submitted and the answers returned. Write also what computer (manufacturer, CPU, memory) you use to solve this planning problem. It is recommended to use Linux servers: read the posted handout about running ECLiPSe Prolog in labs.

2 (50 points). Consider a planning problem in the following logistics domain consisting of places in cities, trucks, airplanes and large containers (boxes). Boxes can be loaded onto trucks or airplanes (if they are not currently loaded into any vehicle) and unloaded from trucks or airplanes. Trucks can be driven between distinct places inside a city, but trucks cannot drive from one city to another. Airplanes can fly between airports in different cities. Given initial locations of boxes and trucks in different cities, and goal locations of boxes and trucks, a planning problem is how to compute a sequence of actions that leads from an initial state to a goal state. This application domain is represented using four action **terms** $load(Box, Vehicle, Location)$, $unload(Box, Vehicle, Location)$, $drive(Truck, LocFrom, LocTo, City)$, $fly(Airplane, LocFrom, LocTo)$ and three **fluents**:
$loaded(Box, S)$, $in(Box, Vehicle, S)$, $at(X, Loc, S)$, where X can be a box, a truck, or an airplane. Fluents are predicates with a situation as the last argument. This representation is general enough to deal with any collection of entities (boxes, trucks, airplanes, locations, cities) in this domain.

- The fluent $in(Box, Vehicle, S)$ is the predicate that is true if Box is loaded in a truck or an airplane $Vehicle$ in situation S. For simplicity, we assume that arbitrary many boxes can be loaded into any vehicle. Each loaded box remains in a vehicle, when vehicle moves from one located to another, until it will be unloaded at a destination.
- The fluent $loaded(B, S)$ is the predicate that is true if the box B is loaded into a vehicle in situation S.
- The fluent $at(X, Loc, S)$ is the predicate that is true if X is at a location Loc in situation S. If X is a truck, it remains at Loc unless it drives to another location. Similarly, if X is an airplane, it remains at Loc, unless it flies to another airport. If X is a box loaded into a vehicle, then it is there where a vehicle is. Unloaded boxes remain in places where they have been last time unloaded. (For the sake of simplicity, arbitrary many boxes can be at any location.)

All actions are terms with the following meaning.

- The action $load(Box, Vehicle, Location)$ loads Box (from a location in a city) onto $Vehicle$. It is possible only if both Box and $Vehicle$ are at the same location, Box is not currently loaded, if $Vehicle$ is either a truck or an airplane, but Box is neither truck, nor an airplane.
- The action $unload(Box, Vehicle, Location)$ unloads Box from $Vehicle$ into the current $Location$. It is possible only if Box is in $Vehicle$, $Vehicle$ is at $Location$, and $Vehicle$ is either a truck or an airplane. Once a box has been unloaded, it is no longer in a vehicle.
- The action $drive(Tr, Loc1, Loc2, City)$ moves the truck Tr from $Loc1$ to $Loc2$ and is possible only if the truck Tr is at $Loc1$ and both locations are in the same city (see below).
- The action $fly(Air, LocFrom, LocTo)$ is possible if Air is an airplane at $LocFrom$, and if both locations are airports in distinct cities (see below).

You have to solve planing problems in this application domain using the situations and fluents approach considered in class. The task of planner will be to find a shortest list of actions such that after doing actions from this list (starting from the initial situation) all boxes will be at specified locations. In addition to fluents and actions mentioned above, you have to use also the following predicates. They are not fluents, because their truth values do not change when actions are executed. The meaning of all these predicates should be clear from their names: $truck(X)$, $airplane(Y)$, $airport(Z)$, $inCity(Location, CityName)$.

Before you can solve planning problems in this domain, you have to write precondition axioms and successor state axioms. Write all your Prolog rules in the file **log.pl** that you can download from the Assignments folder on D2L. More specifically, you have to do the following.

1. Write precondition axioms for all four actions. 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.

2. Write successor-state axioms that characterize how the truth value of all fluents change from the current situation S to the next situation $[A|S]$. 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, and (b) rules that characterize when a fluent remains true in the next situation, unless the most recent action changes it to false. (When you write successor state axioms, you can usually start bodies of rules with negation, but in some cases it is safer to use negation later in a rule.) Keep all your rules in the file **log.pl**

3. Consider the following initial and goal configurations (notice that none of the boxes is loaded into any vehicle):

```
/*----------------- The initial state.----------------- */
at(bigBox1,memorialUnivNewfoundland,[]). at(bigBox2,ryerson,[]).
at(bigBox3,univBritishColumbia,[]). at(bigBox4,univOfTor,[]).

at(daimler,pearsonAirport,[]). at(volvo,univOfTor,[]).
at(gmc,stJohnAirport,[]).  at(ford,yvrAirport,[]). %% near Vancouver %%

at(airbus380,pearsonAirport,[]). at(an124,pearsonAirport,[]).
at(boeing747,stJohnAirport,[]). at(md90,yvrAirport,[]).

/*------------------ Goal state ----------------------- */
%%This simple planning problem takes 3 steps and can be solved in seconds
% goal_state(S) :- at(bigBox1,stJohnAirport,S).

%% This problem requires 5 actions and can be solved in a few seconds
%% goal_state(S) :-  in(bigBox1,boeing747,S).

% The following problem needs 6 actions to solve and can take a few minutes
% goal_state(S) :- at(bigBox1,pearsonAirport,S).
```

   Solve each of these simple planning problems using the planner with an upper bound on the number of actions. (The program with atomic statements characterizing the initial situation and goal states is provided in **initLogistics.pl** that you have to download too). Each time, remove the comment symbol from the line including the goal state that you would like to use (remember to comment out lines that you do not need). For the planning problem that requires 6 actions, your program should spend no more than a few minutes to compute a plan. However, you should start with the easiest problem first to convince yourself that your program works correctly.

   Do not copy content of **initLogistics.pl** into your file **log.pl** because the TA will use other initial and goal states to test your program in **log.pl** Request several plans using ";" command or by clicking on the button "more" (keep all plans and results in your file **log.txt**). Discuss briefly your results in the file **log.txt** What do you observe when you try different goal states ?

Handing in solutions: (a) An electronic copy of your files **log.pl** and **log.txt** must be included in your **zip** archive. (b) The file **log.txt** must include a copy of session(s) with Prolog, the queries to solve the planning problems, the computed plans and information about time that your program spent on finding several plans. Write also on what computer (manufacturer, CPU, memory) you solved planning problems.

3. Bonus work (**20-30 points**):

To make up for a grade on another assignment or test that was not what you had hoped for, or simply because you find this area of Artificial Intelligence interesting, you may choose to do extra work on this assignment. *Do not attempt any bonus work until the regular part of your assignment is complete.* If your assignment is submitted from a group, write whether this bonus question was implemented by all people in your team (in this case bonus marks will be divided evenly between all students) or whether it was implemented by one person only (in this case only this student will get all bonus marks).

Consider the modified version of the generic planner:

```
reachable(S2, [M | ListOfActions]) :- reachable(S1,ListOfActions),
                        legal_move(S2,M,S1),
                        not useless(M,ListOfActions).
```

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 to speed-up the search that your program is doing to find a list of actions that solves a problem. This predicate provides (application 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 situation. Your rules should be good 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 **bonus.pl**: it must include the program **log.pl** that you created in Part 2 of this assignment. Write also brief comments to explain your rules. You have to write at least 10 different rules. The specification of the goal and initial states must remain in the file **initLogistics.pl**, but you must make appropriate modifications (remove comments, etc) in that file when you do testing. Once you have the rules for the predicate `useless(A,List)`, solve all of the following planning problems, this time using the modified planner:

```
% he following plan includes 9 actions; it cannot be computed (without heuristics)
%With the complete set of heuristics, this problem takes less than 1 sec to solve!!
% goal_state(S) :- at(bigBox1,ryerson,S).

%For this goal state the plan of 9 actions can be easily computed with heuristics
%in seconds, but without heuristics it cannot be computed even in a few hours.
% goal_state(S) :-  at(bigBox2,univBritishColumbia,S).

% The planning problem when 2 boxes should be delivered between universities
% requires 18 steps: it is computationally challenging even with declarative
% heuristics. On a fast machine it takes about 30min to solve this problem.
% However, without  heuristics this problem is computationally infeasible.
% goal_state(S) :-  at(bigBox1,ryerson,S),
% at(bigBox2,univBritishColumbia,S).
```

When you solve this last planning problem look for a plan that has no more then 18 actions. (**Warning**: your program may take about 30 minutes to solve this last planning problem depending on how fast is your computer). The previous two planning problems require only 9 actions, but because there are about 10 actions (or more) possible in any state, in the worst case, the program has to search through 10^9 sequences of actions. For this reason, without declarative heuristics, your program cannot find a solution quickly, but if you provide a good set of heuristics, then solving these planning problems becomes easy. Note that you are expected to write at least 10 different rules (implementing 10 different heuristics). The TA will mark your solution of this bonus question depending on how fast your program can solve planning problems. The faster your program works, the more marks you will get.

Write a brief report (in the file **bonus.txt**): the queries that you have submitted to your program (with heuristics) and how much time your program spent to find a plan when you added more heuristics. Request several plans using ";" command. Discuss briefly your results and explain what you observed. Provide comments about your heuristics: why they make sense in this domain ? Explain why heuristics help to decrease computation time. Note that TA who will be marking your assignment will use another specification of initial and goal states.

Handing in solutions: An electronic copy of both files **bonus.pl** and **bonus.txt** must be included in your **zip** archive. Remember to write brief comments in **bonus.pl** with explanations of your declarative heuristics. Your Prolog file **bonus.pl** should contain only your precondition and successor state axioms and rules for the predicate `useless(A,List)`: do not copy there anything from **initLogistics.pl**

How to submit this assignment. Read regularly *Frequently Answered Questions* and replies to them that are linked from the Assignments Web page at

`http://www.scs.ryerson.ca/~mes/courses/cps721/assignments.html`

If you write your code on a Windows machine, make sure you save your files as plain text that one can easily read on Linux machines. Before you submit your Prolog code electronically make sure that your files do not contain any extra binary symbols: it should be possible to load `factory.pl` or `log.pl` into a recent release of ECLiPSe Prolog, compile your program and ask testing queries. TA will mark your assignment using ECLiPSe Prolog. If you run any other version of Prolog on your home computer, it is your responsibility to make sure that your program will run on ECLiPSe Prolog (release 6 or any more recent release), as required. For example, you can run a command-line version of *eclipse* on moon remotely from your home computer to test your program (read handout about running *ECLiPSe Prolog*). To submit files electronically do the following. First, create a **zip** archive on `moon`:

```
zip yourLoginName.zip factory.pl factory.txt log.pl log.txt [bonus files]
```

where `yourLoginName` is the Ryerson login name of the person who submits this assignment from a group. Remember to mention at the beginning of each file *student, section numbers* and *names* of all people who participated in discussions (see the course management form). You may be penalized for not doing so. Second, upload your zip file

yourLoginName.zip

to D2L into "Assignment 5" folder. Make sure it includes **all** your files. Improperly submitted assignments will **not** be marked. In particular, you are **not** allowed to submit your assignment by email to a TA or to the instructor.

Revisions: If you would like to upload a revised copy of your assignment, then simply upload it again. (The same person must upload.) Do not ask your team members to upload your assignment, because TA will be confused which version to mark: only one person from a group should submit different revisions of the assignment. The groups that submit more than one copy of their solutions will be penalized. The time stamp of the last file you upload will determine whether you have submitted your assignment on time.