cps721: Assignment 5 (100 points).
Due date: Electronic file - Monday, November 28, 2022, 21:00 (sharp).

You can get extra 10% by submitting this assignment before Friday, November 25, 23:00.

You have to work in groups of TWO, or THREE; you cannot work alone.

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

ALSO, YOU CANNOT USE SYSTEM PREDICATES OR LIBRARIES NOT MENTIONED IN CLASS.

You can discuss this assignment only with your CPS721 group partners or with the CPS721 instructors. You cannot use any external programming resources (including those which are posted on the Web) to complete this assignment. Failure to do this will have negative effect on your mark. 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. *The final exam may include similar questions*.

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 Parts 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. Read your lecture notes for examples: we discussed in class tiles-on-grid, coins-on-desk and sorting examples implemented using situations and fluents approach. (Download the planner from the Assignments page).

1 (70 points). Consider the problem of changing a flat tire. More precisely, the goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk. If you have never replaced a flat tire, you can find detailed guides posted online at

`http://www.edmunds.com/how-to/how-to-change-a-flat-tire.html`

and at `http://www.wikihow.com/Change-a-Tire`

(the latter Web page links a short video too). You have to solve this planning problem using an approach considered in class. Specifically, you have to take the generic planner written in Prolog, but you have to provide missing rules as explained below. Subsequently, we consider a simplified version of this problem and several different goal states that are increasingly more difficult to reach from a given initial state. To make sure that you can solve problems in this domain with different goal states, the provided planner has an extra parameter G that can be instantiated with a number from 0 to 12 to indicate which of the goal states should be used when problem solving. Start with writing the precondition axioms: these are rules (one rule per action) which say when actions are physically possible to execute in the current state. When you write the precondition axioms, concentrate on the common-sense meaning of the actions (not on whether an action is useful or not). The domain has the following actions (Recall that actions are **terms**: they can be only arguments of predicates or equality.)[1]:

- $open(C)$: open a container C, provided it is closed in situation S.
- $close(C)$: this action will undo the effects of the open action. It is possible to execute this action if C is a container that is not closed.
- $fetch(X, Y)$: take X out of container Y. This action is possible if Y is a container, X is inside Y in current situation S, and if Y is not closed in S.
- $putAway(X, Y)$: put X into container Y. This action is possible if Y is a container, you have an object X in S, and Y is not closed in current situation S. The actions $fetch(X, Y)$ and $putAway(X, Y)$ are reversible and have opposite effects: once you fetched an object X, you have X, but when you put X away, you no longer have it.
- $loosen(X, Y)$: loosen nuts X on a hub Y with a wrench. This action is possible to do if you have a wrench, nuts X are on the hub Y, they are tight, and the hub Y is on the ground (to prevent the hub from rotating when you apply force to the hub with a wrench).
- $tighten(X, Y)$: tighten nuts X on hub Y to undo the effect of the opposite action $loosen(X, Y)$. This action is possible if you have a wrench in S, nuts X are on the hub Y in S, but they are not tight, and the hub Y is firmly on the ground in S. The actions $loosen(X, Y)$ and $tighten(X, Y)$ are reversible and have opposite effects on the tightness of the nuts. However, these actions do not remove and do not put the nuts on the hub, they can only make the nuts tight or not.

[1] Be careful: do not start the right hand sides of your precondition rules with negation, since this may lead to unsound logical inferences. Revisit the slide "Caution: variables and negation" (page 16 in Prolog Basics), if you forgot details. Make sure that all variables in a predicate are instantiated by constants before you apply negation to the predicate that mentions these variables.

- $jackUp(Object)$: if you have a jack in S, and $Object$ is not lifted, but it remains on the ground, then you can jack this $Object$ up no matter how heavy is it.

- $jackDown(Object)$: if $Obect$ is lifted in S, and $Object$ is not on the ground in S, then you can jack $Object$ down. This action has an opposite effect on $Object$ with respect to $jackUp(Object)$: the latter makes an object lifted, while the former makes it not lifted. In addition, when you jack an $Object$ down, it becomes located on the ground, but when you lift it up, it looses its contact with the ground. Moreover, if there are any other things on $Object$, then when you jack $Object$ up, all those things also are no longer on the ground, i.e., not only $Obejct$ itself, but also all other things attached to it loose contact with the ground. Respectively, when you jack X down, not only X becomes on the ground, but all other objects on X also end up on the ground. For example, if a wheel is on a hub, and you jack the hub up, then not only the hub itself, but also the wheel on this hub are no longer on the ground.

- $remove(Nuts, Hub)$: it is possible to remove $Nuts$ from a Hub if the variables $Nuts$ and Hub have the right types (as determined by the predicates $nut(X)$ and $hub(Y)$, see below), Hub is lifted and fastened, $Nuts$ are actually on the Hub, the $Nuts$ are not tight, and if you have a wrench in S. Note that this action can be executed only if the Hub is lifted, because otherwise Hub is pressed down by the car body and $Nuts$ cannot be removed. Once someone has removed $Nuts$, the Hub is no longer fastened, but it becomes fastened again, if someone puts $Nuts$ back on the Hub.

- $putOn(N, H)$: it is possible to put nuts N on a hub H if you have nuts N in S, if the hub H is lifted, but not already fastened in S, and if you have a wrench in S.

- $remove(W, H)$: remove a wheel W from a hub H provided the hub is lifted, but not fastened and if the wheel W is on the hub H in current situation S. After executing this action, a wheel W is no longer on a hub H. This action has an opposite effect to an action $putOn(W, H)$. When the latter action is executed, W is on H in the resulting situation.

- $putOn(W, H)$: put W on a hub H provided W is a wheel, you have it in S, H is a hub that is lifted and free in S, but that is not fastened in S, and if the wheel W is not already on the hub H in S. Once you have put a wheel on a hub, the hub is no longer free, but it can become free, if the wheel is removed, i.e., the actions $remove(W, H)$ and $putOn(W, H)$ have opposite effects on the fluent $free(Hub, S)$.

There are several auxiliary predicates that are not fluents: their truth values do not change after executing any of the actions. In particular, $nut(N)$ means that N is nuts, $hub(H)$ means that H is a hub, $wheel(W)$ means that W is a wheel, and $is\_container(C)$ means that C is a container. For example, a car itself or its trunk can be a container. These predicates do not change from one situation to another. To represent features of this domain that can change, it is sufficient to introduce the following predicates with situation argument (fluents). Recall that fluents are **predicates** with a situation S as the last argument, where situations are represented by lists of actions that have been already executed. Situation can be understood as a history, i.e., as a sequence of executed actions. Since fluents are predicates, action terms can occur as arguments in fluents.

- $inflated(W, S)$: a wheel W is inflated in S. For simplicity, there are no actions that have any effect on this fluent. So, if one of the tires is flat, this cannot be fixed. If a spare tire is inflated, it remains to be inflated no matter what actions are being executed.

- $isClosed(C, S)$: container C is closed in S. This fluent becomes true after doing $close(C)$ action; C remains closed unless $open(C)$ action is executed.

- $inside(X, C, S)$: an object X will be inside a container C in situation S if someone puts X away, and then X remains inside C unless someone fetches X from C.

- $have(X, S)$: you have an object X if you fetched it from a container, or if you removed X from something else. In case if X is a jack, you have it once you have moved some load down. When you move a load up, you no longer have a jack (since it is occupied to support its load). Similarly, you do not have an object X, if you put it away in a container, or if you put it on something.

- $tight(N, H, S)$: nuts N are tight on a hub H in a current situation S. This fluent is affected by $loosen$ and $tighten$ actions.

- $lifted(X, S)$: X is lifted in S if you jack an object X up, and it remains lifted until you jack X down.

- $on(X, Y, S)$: X is on Y in S. There are several possibilities here: if Y is the ground itself, then an object X is on the ground depending on whether you jack X up or down. Also, as discussed previously, all other objects on X can end up being on the ground or not, depending on whether you jack X itself up or down. In addition, X is on Y, if someone puts X on Y, and X is not on Y, if someone removes X from Y.
- $fastened(H, S)$: a hub H is fastened in situation S if you put nuts on H. A hub X is not fastened if you remove nuts from this hub.
- $free(H, S)$: a hub H is free, when you remove a tire, and becomes not free, when you put a tire on.

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 **tire.pl** provided for you. Write precondition axioms for all 12 actions. Write successor-state axioms that characterize how the truth value of 8 fluents can 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. To make sure that there are no errors in your rules, test them on solving simple planning problems as recommended below. Consider the following description of initial situation $[]$:

```
nut(nuts1). nut(nuts2). nut(nuts3). nut(nuts4).
hub(hub1). hub(hub2). hub(hub3). hub(hub4).
% wheel5 is a spare wheel that is inflated and can be found inside the trunk
wheel(wheel1). wheel(wheel2). wheel(wheel3). wheel(wheel4). wheel(wheel5).
is_container(trunk).     is_container(car).        isClosed(trunk,[]).
inside(jack,trunk,[]).   inside(wheel5,trunk,[]).  inside(wrench,trunk,[]).

on(wheel1,hub1,[]).   on(wheel2,hub2,[]).    on(wheel3,hub3,[]).    on(wheel4,hub4,[]).
on(wheel1,ground,[]). on(wheel2,ground,[]).  on(wheel3,ground,[]).  on(wheel4,ground,[]).
on(hub1,ground,[]).   on(hub2,ground,[]).    on(hub3,ground,[]).    on(hub4,ground,[]).
on(nuts1,hub1,[]).    on(nuts2,hub2,[]).     on(nuts3,hub3,[]).     on(nuts4,hub4,[]).

tight(nuts1,hub1,[]). tight(nuts2,hub2,[]).  tight(nuts3,hub3,[]).  tight(nuts4,hub4,[]).
fastened(hub1,[]).    fastened(hub2,[]).  fastened(hub3,[]).  fastened(hub4,[]).

inflated(wheel5,[]).  inflated(wheel1,[]). inflated(wheel2,[]).  inflated(wheel3,[]).
```

1. Download the file **initTire.pl** with descriptions of the initial and goal states from the Assignments Web page. Keep both this file and the file **tire.pl** in the same folder. When ECLiPSe Prolog compiles the file **tire.pl** it loads the file **initTire.pl** automatically.

   Do not copy content of **initTire.pl** into your file **tire.pl** because a TA may use other initial and goal states to test your program in **tire.pl** Do *not* submit the file **initTire.pl**

   Solve several simple planning problem using the planner with the upper bound 7 on the number of actions: use goal states numbered from 0 to 5. Solving these planning problems does not need lots of search, so your program should solve each of them in a few seconds (less than 10 seconds even if your CPU is slow). If your program cannot solve any of these simple problems in 10sec or sooner, or if it computes a wrong plan, then there is a bug in your program. 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 simple goal state. Usually, this strategy of debugging your program with the help of testing queries can help in locating a bug faster than trying to trace your program with a debugger in tkECLiPSe. Request several plans using "more" command. Make sure all plans are correct. Discuss briefly your results in the file **tire.txt**

2. The predicate `useless(A,ListOfPastActions)` is true if an action A is useless given the list of previously executed actions. The predicate `useless(A,ListOfPastActions)` helps solve the planning problem by providing *declarative heuristics* (advises) to the planner. If this predicate is correctly defined using a few rules (one or more rules per action A), then it helps speed-up the search that your program is doing to find a list of actions solving a problem. Write as many rules as you can to implement 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 should never

use any constants mentioned in the initial or goal states because your rules have to be domain specific, but they should be independent of a particular instance of the planning problem that you are solving. Consequently, your rules defining $useless$ can use only variables. Keep these new rules in the same file **tire.pl** Your rules have to be general enough to be applicable to any tire replacement domain as outlined above. In other words, they have to speed-up 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 on same planning problem as above, but now with the goal states numbered from 6 to 11, or any similar simple planning problems. Warning: solving an instance of a planning problem for goal states 6, 7 and 8 without declarative heuristics can take up to a few minutes of CPU time (if you work in a lab, or if your CPU is slow). It is not recommended to try goal states with numbers higher than 8 without heuristics: you program can keep computing for hours or longer. However, if you have a clever set of heuristics solving the problems 6-12 should take less than 1 minute. In other words, well-thought heuristics should provide 100 times speed-up. *Hints: think about purposes of the action that closes a container. In a reasonable plan, once you have closed a container, there is no point in opening it again any time later. Otherwise, it was premature to close the container and it should remain opened. Similarly, you never need to jack up more than one hub, assuming that only one tire can go flat at once.*

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 for each instance. **Explain why**. Include your testing results and a *brief discussion of your results* in the report **tire.txt**

Handing in solutions: An electronic copy of your files **tire.pl** as well as a copy of **tire.txt** must be included in your **zip** archive. The file **tire.txt** must include a copy of session(s) with Prolog, showing all the queries you submitted and the answers returned. Write also what computer (manufacturer, CPU, memory) you use to solve this planning problem. If you are working in lab, check system info or ask a system administrator about CPU and memory installed on your machine. It is recommended to use Linux servers instead of Windows machines: read the handout about running ECLiPSe Prolog in labs.

2 (30 points). The application domain is as follows: NASA needs to plan the activities of a collection of satellites. Each satellite possesses several instruments and each instrument supports various *modes*. For example, an instrument that supports modes `spectrograph` and `thermograph` can take both spectrographic and thermographic images of an object of interest. There is a finite set of predefined directions that a satellite can point to (such as comets, stars, galaxies, ground stations, etc.) Since satellites have a limited power supply, a satellite can only power up one instrument at a time. When an instrument is powered up, it becomes uncalibrated. An instrument can be calibrated if it is pointed in a prespecified direction. Assume that each instrument has a specific *calibration target*, which will be the ground station it must be pointing to in order to run the calibration procedure. The satellite can take an image (in a specific mode) using any instrument that supports that mode, as long as the instrument is powered up, calibrated, and pointing in the direction of interest.

Generally, a goal will require that we have a collection of images of various "directions" in specific modes. For example, one may request a thermographic image of Andromeda nebula and a spectrographic image of Halley comet.

We consider below terms (actions) and predicates (fluents) that are general enough to deal with any collection of objects (satellites, instruments, modes, directions). In particular, we consider the following five actions:

- $up(Ins, Sat)$ – power up an instrument Ins on a satellite Sat (this is possible only if no other instrument at Sat is powered);
- $down(Ins, Sat)$ – power down an instrument Ins on a satellite Sat;
- $turnTo(Sat, Dir1, Dir2)$ – turn a satellite Sat from a direction $Dir1$ to another direction $Dir2$;
- $runCalibrateProc(Ins, Sat, G)$ – run calibrate procedure using a ground station G for an instrument Ins on a satellite Sat;
- $takeImage(Ins, Sat, M, Dir)$ – instrument Ins on a satellite Sat takes image of object in Dir using a mode M.

We also consider the following fluents:

- $powered(Instr, Satell, S)$ – $Instr$ on $Satell$ is powered in a situation S;
- $pointsTo(Satell, Dir, S)$ – a satellite $Satell$ points in a direction Dir in a situation S;
- $calibrated(Instr, Satell, S)$ – $Instr$ on $Satell$ is calibrated in a situation S;
- $hasImage(Sat, M, Dir, S)$ – a satellite Sat has an image in mode M of an object in Dir in a situation S.

Finally, we consider auxiliary predicates that do not have situational argument (they represent properties that do not change): $supports(In, Sat, M)$ means that an instrument In on Sat supports a mode M; $available(Sat, D)$ means that a direction D is one of the predefined directions that a satellite Sat can point to; and $target(In, GS)$ means that an instrument In can be calibrated using a ground station GS. Write the following rules in your file **comet.pl** (you can download this file from the Assignments Web page):

1. Write precondition axioms for all five 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 use negation with the predicate that mentions these variables. Note also that *only one* instrument on a satellite can be powered at a time.

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. (Remember to start bodies of your rules with predicates without negation.)

3. Consider the following initial and goal configurations:

```
supports(instr1,sat1,therm).
available(sat1,ground17).
available(sat1,comet2).
available(sat1,orion).
target(instr1,ground17).

pointsTo(sat1,orion,[]).

goal_state(S) :- hasImage(sat1,therm,comet2,S).
```

   Solve this simple planning problem using the planner with the upper bound 10 on the number of actions (download the initial and goal configurations from the Assignments Web page: the file **initNASA.pl**). It should take no more than a few seconds for your planner to solve this problem. Keep a copy of your session with Prolog in your file **comet.txt**
   Do not copy content of **initNASA.pl** into your file **comet.pl** because TA may use other initial and goal states to test your program in **comet.pl** Request several plans using "more" command (keep all plans in your file **comet.txt**).

Handing in solutions: An electronic copy of your files **comet.pl** and **comet.txt** must be included in your **zip** archive. The file **comet.txt** should contain the query to solve the problem, the computed plans and information about time that your program spent on finding a plan. Write also on what you computer (CPU,memory) you solved this planning problem.

3. Bonus work (**20 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 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 **nasa.pl**: it must include the program **comet.pl** that you created in Part 2 of this assignment. The specification of the goal and initial state must remain in the file **initNASA.pl**, but you must make appropriate modifications (remove comments, etc) in that file. Once you have the rules for the predicate `useless(A,List)`, solve another planning problem, this time using the modified planner:

```
supports(instr1,sat1,therm).     supports(instr2,sat1,spectr).
available(sat1,andromeda).
available(sat1,ground17).
available(sat1,comet2).
available(sat1,orionStarts).
target(instr1,ground17).         target(instr2,ground17).

pointsTo(sat1,orionStars,[]).

goal_state(S) :- hasImage(sat1,therm,comet2,S),
                 hasImage(sat1,spectr,andromeda,S).
```

When you solve this planning problem look for a plan that has no more then 15 actions. If your rules are creative enough, solving this problem should be fast. Once, you solved it, try to solve the same planning problem without using rules for the predicate `useless(A,L)`: put comments on the rule that defines the predicate `reachable(S,[A|L])` and calls the predicate `useless(A,L)`. Warning: your program may take a few minutes to solve this version (depending on how fast is your computer). Request several plans using "more" button (or using ";" command).

Write a brief report (in the file **nasa.txt**): 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. 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 **nasa.pl** and **nasa.txt** must be included in your **zip** archive. You have to write brief comments in **nasa.pl** with explanations of your declarative heuristics. Your Prolog file **nasa.pl** should contain only your precondition and successor state axioms and rules for the predicate `useless(A,List)`.

How to submit this assignment. Read regularly *Frequently Answered Questions* and replies to them that are linked at
`https://www.cs.torontomu.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 `tire.pl` and `comet.pl` into a recent release 6 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 any *moon* Linux server:

```
zip yourLoginName.zip tire.pl tire.txt comet.pl comet.txt [bonus]
```

where `yourLoginName` is the TMU 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 only (**No individual files!**)

yourLoginName.zip into the "Assignment 5" folder on D2L.