Assignment 3

EECS 3401 A

November 25, 2020

This assignment is **due on December 7** (Monday) at 23:59.

Your report for this assignment should be the result of your own **individual work**.

Question 1 [40 points] — Belief Networks

Consider the following example:

> Metastatic cancer is a possible cause of a brain tumour and is also an explanation for an increased total serum calcium. In turn, either of these could cause a patient to fall into an occasional coma. Severe headache could also be explained by a brain tumour.

(a) Represent these causal links in a **belief network**. Use the following glossary:

a = "metastatic cancer"

b = "increased total serum calcium"

c = "brain tumour"

d = "occasional coma"

e = "severe headaches"

(b) Give an example of an independence assumption that is implicit in this network.

(c) Suppose that the following probabilities are given:

$\boldsymbol{Pr}(a) = 0.2$	$\boldsymbol{Pr}(d \mid b, c) = 0.8$	$\boldsymbol{Pr}(e \mid \neg c) = 0.6$
$\boldsymbol{Pr}(b \mid a) = 0.8$	$\boldsymbol{Pr}(d \mid b, \neg c) = 0.8$	$\boldsymbol{Pr}(d \mid \neg b, c) = 0.8$
$\boldsymbol{Pr}(c \mid a) = 0.2$	$\boldsymbol{Pr}(b \mid \neg a) = 0.2$	$\boldsymbol{Pr}(d \mid \neg b, \neg c) = 0.05$
$\boldsymbol{Pr}(e \mid c) = 0.8$	$\boldsymbol{Pr}(c \mid \neg a) = 0.05$	

and assume that it is also given that *some patient is suffering from severe headaches but has not fallen into a coma.* **Calculate the joint probabilities** for the eight remaining possibilities (that is, according to whether a, b, and c are true or false).

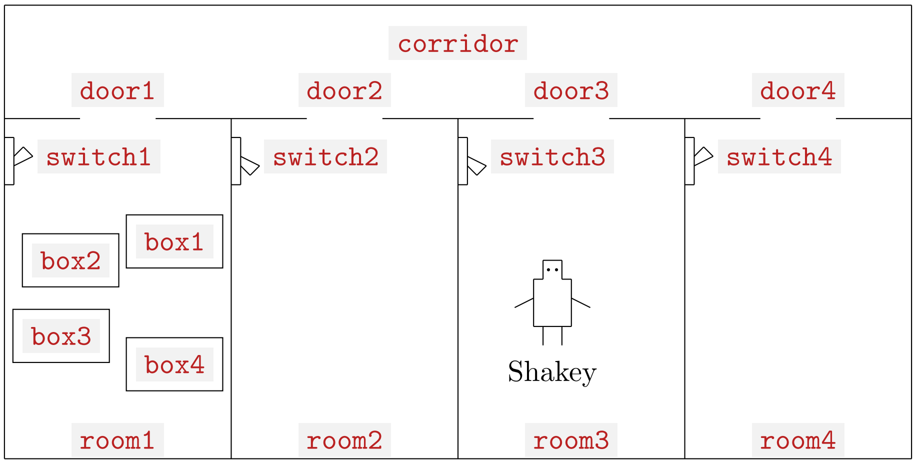
(d) According to the numbers given, the *a priori* (prior) probability that the patient has metastatic cancer is 0.2. Given that the patient is suffering from severe headaches but has not fallen into a coma, are we now **more** or **less** inclined to believe that the patient has cancer? Justify your answer.

Put your answer to Question 1 in a PDF file called `a3q1.pdf`.

Question 2 [70 points] — GOLOG

> **Note!**
> You are provided with starter code in `a3q2.prolog`. Please write your code there.
> Refer to the included file `simple_elevator.prolog` for inspiration.
> For parts (c)–(e), refer to the GOLOG tutorial at the end of this document.

Consider Shakey's world:[1]



Shakey can move between landmarks within a room, can pass through the door between rooms, can climb boxes, can push boxes around, and can flip light switches (but needs to stand on a box to reach them).

(a) Develop a **Prolog implementation** of a situation calculus action theory for Shakey's world as described above. Put all your code in file `a3q2.prolog`.

Write a **precondition axiom** for each action and a **successor state axiom** for each fluent. Also write axioms describing the **initial state** pictured above.

Assume there is a light in each room (except the corridor), and that it is on if the switch is up. For the initial location of the robot, use a constant such as `locInitRobot`, and similarly for the boxes and switches. Since Shakey is able to climb boxes, use a constant `floor` to denote when Shakey is not on a box, e.g., `onTop(floor,S)`.

Use the following primitive actions:

- `go(Loc1,Loc2)` : go from location `Loc1` to location `Loc2`. This is possible if Shakey is at location `Loc1` and on the floor (as opposed to on a box), and both locations `Loc1` and `Loc2` are in the same room. A door between two rooms is considered to be in both rooms.
- `push(B,Loc1,Loc2)` : push a box `B` from location `Loc1` to location `Loc2` within the same room. Must ensure that `B` is a box, both locations belong to the same room, both Shakey and the box are at location `Loc1`, and Shakey is not on a box.

[1]The original STRIPS planner was designed to control Shakey the robot. The problem is adapted here from problem 10.4 in 3rd ed. of Russell & Norvig.

- `climbUp(B)` and `climbDown(B)` : climb onto box `B`; climb down from box `B`. Must ensure that `B` is a box, that the locations match, and that Shakey is where he is supposed to be in order for the action to make sense.
- `turnOn(Sw)` and `turnOff(Sw)` : turn a light switch `Sw` on or off. To do this, Shakey must be on top of a box `B`, and the box must be at the light switch's location.

Use the following fluents:

- `robotLoc(Loc, S)` : Shakey is at location `Loc` in situation `S`
- `boxLoc(Box, Loc, S)` : box `Box` is at location `Loc` in situation `S`
- `onTop(B, S)` : Shakey is on top of `B` in situation `S`. `B` could be a box or `floor`.
- `up(Sw, S)` : switch `Sw` is up in situation `S`
- `onn(Light, S)` : light `Light` is on in situation `S`

Use the following non-fluent predicates:

- `in(Loc, R)` : location `Loc` is in room `R`
- `controls(Sw, Light)` : switch `Sw` controls the light `Light`
- `box(B)` : `B` is a box
- and any others you might need.

(b) Suppose that we want to achieve the goal of having `box2` in `room2`. **Express this goal as a situation calculus sentence**. Specifically, implement the predicate

```
goalPartB(S) :- /* your goal here */
```

Also **write a ground situation term** that represents a plan that achieves this goal when executed in the initial state (pictured above). Have the following predicate succeed with `S` bound to the plan:

```
solPartB(S) :- /* describe S here */
```

Use your Prolog implementation of the action theory in (a) to confirm that the plan is **executable (legal)**[2] and **achieves the goal**. Put your log in the file `a3q2tests.txt`.

(c) Write a GOLOG program that represents the plan in (b) (as a sequence of primitive actions) and show that it can be executed successfully (put your execution log in the file `a3q2tests.txt`). Use the supplied GOLOG interpreter.

(d) Write a GOLOG procedure `allLightsOn` that can be executed to turn on all the lights. The procedure should always terminate successfully and should succeed in turning on all the lights as long as a there is a box in some room that can be used by the robot to reach the switch. Run your procedure and show (save the log in the file `a3q2tests.txt`) that it can be executed successfully in the initial situation and that all the lights are on in the resulting situation. Your code should define sub-procedures as appropriate and not be unnecessarily complex.

[2]Look at the bottom of the starter code for a complete implementation of `executable`.

(e) Use the GOLOG iterative deepening planning procedure (in the GOLOG interpreter file) to generate a plan to achieve the goal of having `box2` in `room2`. For the search procedure to find a solution, you will have to define the forward filtering fluent `acceptable(A, S)` appropriately. Try using the following task-specific heuristics:

   i) until the robot is at the location of `box2`, `go` is the only acceptable action, and

   ii) once the robot is at the location of `box2`, `push` applied to `box2` is the only acceptable action.

Save the execution log in `a3q2tests.txt`. Document your code appropriately.

To hand in your report for this assignment, put the files `a3q1.pdf`, `a3q2.prolog`, and `a3q2tests.txt` in a ZIP archive called `a3answers.zip` and submit it **through eClass** by the deadline.

GOLOG Tutorial

The important bit to remember is that a GOLOG program is a *complex action*, i.e., a syntactic structure which describes, at a high level of abstraction, a legal sequence (generally, many possible sequences) of primitive actions. A *GOLOG procedure* is simply a complex action which is defined under a specific name, so that the programmer can refer to it by name.

The GOLOG interpreter provided to you in the file `golog_swi.prolog` is written in Prolog. Thus, when programming in GOLOG, we end up writing a logical program *within* a logical program, which is confusing (to say the least). To keep the two levels separate, GOLOG programs are written as Prolog *terms*—not predicates. Moreover, GOLOG uses a custom-made language—not Prolog—for writing logical formulas (which can appear as parts of complex actions, see examples below). At the same time, the Situation Calculus theory describing the domain of application is written natively in Prolog (so that the queries about the domain could be evaluated using the built-in SLDNF-resolution), and the GOLOG code has to interface with it somehow. Specifically, when a GOLOG program needs to evaluate a logical formula ϕ (written in its custom language), the GOLOG interpreter converts ϕ to a Prolog query and uses Prolog to prove it.

Example A complex action is a syntactical structure such as `if(hungry,eat,sleep)` constructed according to the formal grammar. The *meaning* behind any such structure is implemented in the Prolog predicate `do(A,S,S1)`. To execute a complex action (or a procedure) starting in situation `s0`, you need to pose a Prolog goal

```
?- do(if(hungry,eat,sleep), s0, S).
S = do(eat, s0)
```

This goal succeeds and binds `S` to a concrete sequence of primitive actions which represents the execution of the program `if(hungry,eat,sleep)`.
But how did Prolog know that this is a legal execution? That's because I also loaded the following code:

```
hungry(s0).
primitive_action(eat).
primitive_action(sleep).
poss(sleep,_).
poss(eat,_).
restoreSitArg(hungry,S,hungry(S)).
```

This code constitutes a very simple situation calculus theory and every single line here is necessary. Specifically, I've defined `hungry` as a fluent which holds in `s0`, declared two primitive actions, and wrote trivial preconditions for them. The last line here tells GOLOG that `hungry` is a fluent (as opposed to a regular predicate) and how to add/remove its situation argument. **This is an important technical peculiarity of the GOLOG interpreter**: when referring to a *fluent* from within a complex action, you should *omit* its situation term. When GOLOG gets down to evaluating that fluent, it will restore the situation argument (according to how you tell it to in `restoreSitArg`) and will use your situation calculus theory for proof. The reason we don't want to have situation arguments inside a complex action is the fact that the complex actions occur inside the `do(_, S, S1)` predicate, which has its own way to keep track of situations.

The semantics of the conditional in this example (and in general) is defined in the GOLOG interpeter as

```
do(if(P,E1,E2),S,S1) :- do((?(P) : E1) # (?(-P) : E2),S,S1).
```

which can be understood as

> To execute `if(P,E1,E2)`, non-deterministically choose between and execute **either** [prove that `P` is true and then do action `E1`] **or** [prove that `P` is false and do action `E2`].

Obviously, the two nondeterministic choices are mutually exclusive (`P` cannot be true and false at the same time), so there is only one way to execute this conditional, which depends on whether `P` can be proved or not. In our example, `P` is `hungry`. From the fact that we've defined `restoreSitArg` for `hungry`, GOLOG knows it to be a fluent, so it restores the situation term, obtaining `hungry(S)`, substitutes the starting situation from `do`, obtaining `hungry(s0)`, and finally poses it as a Prolog query `?- hungry(s0).` which succeeds, which means that the next step of execution is `do(eat, s0, S)`.

Full grammar and semantics of `do(...)` In file `golog_swi.prolog`, you can see the implementation of `do(A,S,S1)` **together with the grammar for complex actions** on lines 57–65:

```
57  do(E1 : E2,S,S1) :- do(E1,S,S2), do(E2,S2,S1).  % simple sequence
58  do(?(P),S,S) :- holds(P,S).  % prove FOL formula P, stay in same situation
59  do(E1 # E2,S,S1) :- do(E1,S,S1) ; do(E2,S,S1).  % nondet. choice of action
60  do(if(P,E1,E2),S,S1) :- do((?(P) : E1) # (?(-P) : E2),S,S1).  % if-else
61  do(star(E),S,S1) :- S1 = S ; do(E : star(E),S,S1).  % repeat action E zero+ times
62  do(while(P,E),S,S1):- do(star(?(P) : E) : ?(-P),S,S1). % while loop
63  do(pi(V,E),S,S1) :- sub(V,_,E,E1), do(E1,S,S1).  % nondet. choice of value
64  do(E,S,S1) :- proc(E,E1), do(E1,S,S1).  % execute procedure named E
65  do(E,S,do(E,S)) :- primitive_action(E), poss(E,S).  % exec. primitive action
```

FOL syntax in GOLOG In the grammar above, the variable `P` always denotes a *FOL condition* on the starting situation. These FOL conditions can be formulated over

- all of your non-fluent Prolog predicates, and
- all fluent predicates, but with the situation argument removed ("suppressed"),

using the logical connectives and quantifiers defined on lines 79–92 of the interpreter in the predicate `holds(P,S)`. This predicate is the semantic link between this custom FOL language and native Prolog. Given a custom FOL formula, `holds(P,S)` converts it to a set of Prolog goals and tries to prove them using the underlying situation calculus theory.

Namely, you can form the following kinds of expressions:

- `P & Q` — conjunction $P \wedge Q$
- `P v Q` — disjunction $P \vee Q$
- `P => Q` — implication $P \rightarrow Q$
- `P <=> Q` — bidirectional implication $P \leftrightarrow Q$
- `-P` — negation $\neg P$
- `all(V,P)` — universal quantification $\forall V\ (P)$
- `some(V,P)` — existential quantification $\exists V\ (P)$

Some examples of FOL conditions

- `-onTop(floor)` — "Shakey is not on the floor". Note that `onTop` is a fluent, but we are "suppressing" its situation argument.
- `all(l,isLight(l) => on(l))` — "All lights are turned on". **Important:** GOLOG requires you to use *constants* to represent quantified variables. This applies to both `all` and `some`. Writing `all(X,isLight(X) => on(X))` is wrong.
- `l = d` — "quantified variables l and d are equal". You can use `=` to denote FOL equality.
- `some(r,in(L,r) & in(d,r))` — "There exists a (room?) r such that L and d are both in it". Note: r is a quantified "variable", d could be either another quantified variable (quantified from outside, not shown here) or a variable obtained through non-deterministic choice of value as per line 63 (through the complex action `pi(d, E)`), and L is a true Prolog variable. Prolog variables may appear inside FOL conditions only if that condition is a part of a complex action which defines/implements *a GOLOG procedure* which takes L (or other variables) as an argument (see below).

An example of a procedure Consider a world which contains things that can be put on top of one another.

```
primitive_action(put(A,B)).  % put thing A on thing B
poss(put(A,B),S) :- \+on(A,B,S).
thing(cat).
thing(duck).
thing(lizard).
```

```
on(cat, duck, s0).  % Initial state
on(X,Y,do(A,S)) :- A = put(X,Y); on(X,Y,S).  % Successor state axiom
restoreSitArg(on(A,B), S, on(A,B,S)).
```

Let's define a procedure to pick some thing at random and put it on top of `X`:

```
proc(put_something_on_top_of(X),
    if(some(a, thing(a) & -(a=X) & -on(a,X)),
        pi(a, ?(thing(a) & -(a=X) & -on(a,X)) : put(a, X)),
        ?(true)
    )
).

% execution:
?- do(put_something_on_top_of(duck),s0,S).
S = do(put(lizard, duck), s0)
```

This procedure is a conditional which tests (using a FOL sentence) whether there exists a thing distinct from `X` which is not on top of `X` in the starting situation. If so, the then-clause uses `pi(a, ...)` to bind `a` to something, proves that the binding is a thing which is not on `X` (if proof fails, Prolog backtracks and tries a different binding for `a`), and finally puts `a` on top of `X` using a primitive action. The else-clause is a tautology and is equivalent to doing nothing.

Next, let us define a procedure which calls the first procedure and uses iteration:

```
proc(put_all_things_on_top_of(X),
  ?(all(v, thing(v) & -(v=X) => on(v,X))) #
  (put_something_on_top_of(X) : put_all_things_on_top_of(X))
).

% execution:
?- do(put_all_things_on_top_of(cat),s0,S).
S = do(put(lizard, cat), do(put(duck, cat), s0))
```

This uses recursion similarly to how Prolog does it. The body of the procedure is a non-deterministic choice of an action. The first choice is to prove that all things are already on top of `X`. If that succeeds, the procedure terminates in the same situation it started, and if that fails (i.e., there are some things still not on top of `X`), the interpreter is forced to try the second choice, which is a sequence involving a call to the first procedure and then a recursive call to the current procedure.

For more examples, study the provided file `simple_elevator.prolog`.