In this homework, you will be learning the dialogue strategy for the system. For this homework, you will be using the car domain.

**Question 1: Random Action Selection**

In the previous questions, the IS engine decided which system action rule to apply by choosing the first applicable action rule. Thus, it is important how the action rules are ordered and what preconditions the action rules have. The two aspects determine the *hand-crafted strategy* that the system follows. Some of the aspects of the hand-crafted strategy are part of common-sense knowledge, such as to output an item before quitting, and to immediately quit after outputting an item. Other aspects are based on guess work, such as what order to ask the parameters, and the number of items at which to summarize (when 5 or less cars were found).

The aspects of the hand-crafted strategy that are based on guess work are the ones that we will eventually learn with Reinforcement Learning. To prepare for that, in this question, you will remove the preconditions based on guess-work. You will also make a control strategy for the system that will randomly choose one of the applicable actions.

First, in `ApplyRules`, add a new `ruleSelection` method called `random`. It should determine which rules are applicable and then randomly apply one of them. If there are no applicable rules, it should print an error message and quit.

**Writeup 1.1** Hand in a copy of `ApplyRules`.

Second, change `RunSide` so that the `random` method is used for system action rules. User action rules should still be done using the method `first`.

Most of the hand-crafted strategy for the system was determined by the order of the action rule, so using the `random` method will remove that. You should also remove the precondition on the system action rule for summarizing that only allows the rule to be used if there are 5 or less cars.

Just as with the previous homework, determine what the average dialogue cost is over 5000 dialogue runs.

**Writeup 1.2** Report the average dialogue cost. Why is this version doing so much worse than the hand-crafted version?

**Question 2: RL States**

The IS state may include everything that has transpired in a dialogue. For example, it will include the exact values of the slots that the user has told the system. For the car domain, after the user has told the system the color, the IS state includes the actual color.

For RL, on the other hand, we often want to keep the number of states as small as possible. For the car domain, we will only track whether we know the color the user wants, whether we know whether the user wants power steering, etc. Hence, when using RL with IS, you will need to add some extra variables, and designate some of the IS variables as defining the RL state.

First, you should change `SetupISVar` so that it allows you to specify whether a variable will be used to define the RL state. This will be done by including a final optional parameter to `SetupISVar` of `-RL`. Here is the new version of `SetupISVar`. Note that `args` is a special variable name in Tcl, which means to include
all following values as a list in the variable. Also note that the RL variables are being kept track of in the
global variable ::RLVars.

proc SetupISVar {agentS var val type args} {
    foreach agent $agentS {
        set ::ISVarType($var) [string tolower $type]
        lappend ::ISVars($agent) $var
        set ::InitValue($var,$agent) $val
        if {$args == "-RL"} {
            lappend ::RLVars $var
        }
    }
}

Now you will adding to (or modifying) the IS variables to include the RL variables. For each of the 11 car
attributes, you should add a boolean variable that will be part of the RL state and that will track if the
system knows that attribute. Call this variable the same name as the attribute but with a final P. Here is
the definition of the new color variable.

SetupISVar {A} colorP 0 boolean -RL

Next, the RL state does need to know some information about how many cars match the current description
so it can decide whether to summarize or ask additional attributes. But, the RL state does not need to keep
track of the exact number of cars match. You will just use a quantized version of the number of cars. You
should track whether no cars match, 1 car matches, 2-5 cars match, 6-15 cars match, or 16 or more, using
the values 0, 1, 2-5, 6-15, and 16-, respectively. Call this variable NData.

The RL state also needs to track the value of Finish so that RL knows when to do the bye action and no
other action. As this variable is already defined, you just need to change its definition to include -RL.

Writeup 2.3  Show the specification for all of the IS variables for the user and system, including the
original IS variables, and the new RL variables that you are adding.

Now you need to revise the effects of the system update rules so that the IS variables that you added are
updated. For updating NData, you should use a procedure that takes as input the number of cars and
outputs the quantized version. This procedure should be called in the effects of the deliberation rule.

Writeup 2.4  Show any update rules that you changed, and explain what you changed in the rule.

When running a simulated dialogue, we will need to determine what RL state we currently are in. We can
view this state as a list of slots fillers, with an equal sign between the slot and its filler, and spaces between
each slot-filler pair. For example, the state in which color, powersteering, and mileage is known, and we have
not summarized and 25 cars match, would be:

Finish=0 colorP=1 doorsP=0 powerwindowsP=0 powerbrakes=0 powersteeringP=1
cylindersP=0 typeP=0 airbagsP=0 yearP=0 mileageP=1 transmissionP=1 NData=16-

Create a procedure called CreateRLState that will create the RL state for agent A based on its IS state.
The routine should not hard-code which variables are in the RL state. Instead, it should use ::RLVars,
thus using the information specified with SetupISVar. The order that the variables are included should be
exactly as they are specified in ::RLVars.

Writeup 2.5  Hand in a copy of the code. Make sure your code does not include a beginning or trailing
space.
Question 3: Preconditions and RL State

The preconditions for the system action rules should only make distinctions captured by the RL variables. The action rules for the attributes, such as the one below, do not cause a problem.

SetupRule {A} action askColor
AddPre {!$is(Finish)}
AddPre {$is(color) == ""}
AddPre {$is(QUD) == ""}
AddEff {set is(NextMove) {query color}}
AddEff {set is(HaveTurn) 0}
AddEff {set is(Cost) 1}

Writeup 3.6 Explain why each of the three preconditions on the action rule above are not a problem.

Question 4: RL Transitions

RL tracks the transitions at a courser level than IS. IS tracks how the state changes after each update rule is applied: understanding rules, deliberation rules, and action rules. RL, on the other hand, only tracks transitions between system actions. These transitions are needed to update the Q scores for each state-action pair seen in the dialogue. For each transition, RL needs to know what RL state the system was in before applying the action, what action was applied, and what the incremental cost was in getting to the next state. Actually, in combining RL and IS, we will have RL track not the action, but what action rule was applied. We will also assume that all of the cost in executing an system action and the next RL state is just on the system’s action rule. All other rules, including the user action rule, will be assumed to have a zero cost.

Change Run and RunSide so that it keeps a list of the tuples of RL state, what action rule was applied, and the incremental cost. Save this as in the global variable ::RLHistory, which should be initialized at the beginning of each dialogue run.

Writeup 4.7 Hand in a copy of Run and RunSide showing what has been changed.

Writeup 4.8 Hand in the value of ::RLHistory variable for a dialogue run.

Question 5: Saving the Dialogue for Debugging

In the previous question, we saved the history of the dialogue so that we will be able to update the Q scores. But, we might also want to save the dialogue so that we can understand what is happening in our code. For example, if we want 500 dialogues, we might want to examine the dialogue that had the lowest cost.

Just as we did for ::RLHistory, let’s add some code to Run and RunSide to save the dialogue in ::DialogHistory. In Run, initialize ::DialogHistory to {}. In RunSide, save the tuple of (a) who the speaker is, (b) what the RL state was before the action, and (c) what next move was (not what action rule was applied). If it is the user’s turn, there is no RL state, and so just record the RL state as -.

Now, use the following code to execute Run 50 times. It will keep track of what the worst dialogue is so far, in terms of its cost and its dialog history. At the end of the 50 runs, it will print out the best dialogue.

```python
set bestC ""
set bestH {}
```
for {set r 0} {$r < 50} {incr r} {
    set c [Run]
    if {$bestC == "" || $c < $bestC} {
        set bestC $c
        set bestH $::DialogHistory
    }
}

puts "Best Dialogue"
foreach i $bestH {
    if {[lindex $i 0] == "A"} {
        puts "\n RL State for A: [lindex $i 1]"
    }
    puts "[lindex $i 0] [lindex $i 2]"
}

Here is a copy of my output. Yours should look similar.

Best Dialogue

RL State for A: Finish=0 colorP=0 doorsP=0 powerwindowsP=0 powerbrakesP=0 powersteeringP=0 cylindersP=0 typeP=0 airbagsP=0 yearP=0 mileageP=0 transmissionP=0 NData=16-
A query year
B 1993

RL State for A: Finish=0 colorP=0 doorsP=0 powerwindowsP=0 powerbrakesP=0 powersteeringP=0 cylindersP=0 typeP=0 airbagsP=0 yearP=1 mileageP=0 transmissionP=0 NData=16-
A query type
B car

RL State for A: Finish=0 colorP=0 doorsP=0 powerwindowsP=0 powerbrakesP=0 powersteeringP=0 cylindersP=0 typeP=1 airbagsP=0 yearP=1 mileageP=0 transmissionP=0 NData=16-
A query mileage
B 40000

RL State for A: Finish=0 colorP=0 doorsP=0 powerwindowsP=0 powerbrakesP=0 powersteeringP=0 cylindersP=0 typeP=1 airbagsP=0 yearP=1 mileageP=1 transmissionP=0 NData=6-15
A query doors
B 2

RL State for A: Finish=0 colorP=0 doorsP=1 powerwindowsP=0 powerbrakesP=0 powersteeringP=0 cylindersP=0 typeP=1 airbagsP=0 yearP=1 mileageP=1 transmissionP=0 NData=2-5
A query cylinders
B 8

RL State for A: Finish=0 colorP=0 doorsP=1 powerwindowsP=0 powerbrakesP=0 powersteeringP=0 cylindersP=1 typeP=1 airbagsP=0 yearP=1 mileageP=1 transmissionP=0 NData=1
A query color
B black

RL State for A: Finish=0 colorP=1 doorsP=1 powerwindowsP=0 powerbrakesP=0 powersteeringP=0 cylindersP=1 typeP=1 airbagsP=0 yearP=1 mileageP=1 transmissionP=0 NData=1
A inform car1162

RL State for A: Finish=1 colorP=1 doorsP=1 powerwindowsP=0 powerbrakesP=0 powersteeringP=0
In the output, you can see the actions that the system and user made, and you can see what the RL state was before the system choose its action. In the best dialogue, you can see that the system summarizes when it just has a single matching car. You can also see that even after it had narrowed down the number of cars to 1, it continued to ask questions in this run. Of course, as we will see later in this homework, the learnt strategy should be behaving much better, even for its worst run.

**Writeup 5.9** Hand in a copy of **RunSide**.

**Question 6: Updating the Q Scores**

After a dialogue is run between the system and the simulated user, we need to update the Q scores. We will use the Monte Carlo method. For each state :math:`s` and action :math:`a` pair in a dialogue run (which you saved in `::RLHistory`), you should determine the cost from this :math:`s-a` to the end of the dialogue (by summing the incremental cost for this :math:`s-a` pair, and all of the subsequent ones in `::RLHistory`).

Take the cost from :math:`s-a` to the end of the dialogue, and average it with your current Q estimate for that :math:`s-a` pair. Thus, you should keep the Q score for each state-action pair, as well as the number of times you have seen the state-action pair. Call the Q score `::RLQ` and the count `::RLCnt`. Both should be associative arrays indexed by :math:`s-a`. To build the index for the associative array, simply `concat` the RL state with the action. Put your code in a procedure called `UpdateQ`.

To make sure you have the right code, test your code with the following.

```bash
foreach {sa c q} {{a=0 b=0 c=0 askA} 5 20 \ 
  {a=0 b=0 c=1 askA} 8 20 \ 
  {a=1 b=0 c=0 askB} 1 6 \ 
  {a=1 b=1 c=0 bye} 9 50} {
  set ::RLQ($sa) $q
  set ::RLCnt($sa) $c
}

set ::RLHistory {{{a=0 b=0 c=0} askA 2} \ 
  {{a=1 b=0 c=0} askB 1} \ 
  {{a=1 b=1 c=0} askC 1} \ 
  {{a=1 b=1 c=1} bye 100}}

UpdateQ

foreach sa [lsort [array names ::RLQ]] {
  puts [format "%-20s %2d %8.4f" $sa $::RLCnt($sa) $::RLQ($sa)]
}
```

You should get the following output.

```
   a=0 b=0 c=0 askA       6   34.0000
   a=0 b=0 c=1 askA       8   20.0000
   a=1 b=0 c=0 askB       1  102.0000
   a=1 b=1 c=0 askC       2  53.5000
   a=1 b=1 c=1 bye       10  55.0000
```

**Writeup 6.10** Hand in the code that you should add to the end of Run to update the scores.
**Question 7: Updating the Policy**

You should now make a procedure called `UpdatePolicy` that will determine for each state that has been seen what the best action is according to the $Q$ scores. This procedure will be called after every 100 epochs. For now, just make the procedure.

Test your procedure with the following code.

```bash
# lets assume that state a=0 b=0 c=0 had previously been set to askB
set sa {a=0 b=0 c=0};
set ::RLPolicy($sa) askB

# and that we now have the following Q values for the states
foreach {sa q} {{a=0 b=0 c=0 askA} 20 \ 
  {a=0 b=0 c=0 askB} 25.3 \ 
  {a=0 b=0 c=0 askC} 15.3 \ 
  {a=0 b=0 c=1 askA} 20 \ 
  {a=0 b=0 c=1 bye} 200.3 \ 
  {a=1 b=1 c=1 bye} 50} {
  set ::RLQ($sa) $q
}

UpdatePolicy

foreach sa [lsort [array names ::RLPolicy]] {
  puts [format "%-20s %-10s" $sa ::RLPolicy($sa)]
}
```

Make sure you get the following.

```
a=0 b=0 c=0 askC)
a=0 b=0 c=1 askA)  
a=1 b=1 c=1 bye)
```

**Writeup 7.11** Hand in the code for the `UpdatePolicy` procedure.

**Question 8: Exploration and Evaluation**

We now need to add support for two more methods for choosing system actions: `policy` and `epsilon`.

**policy:** Follow the action specified in `::RLPolicy` if there is one; otherwise follow `random`.

**epsilongreedy:** Follow the `random` method a certain percentage of the time (determined by `::Epsilon`), otherwise follow `policy`.

Here is the code to add to `applyRules`. This code should go before the `if` statement for the `random` method.

```bash
if {$howSelect == "epsilongreedy"} {
  if {[expr rand()] < $::Epsilon} {
    set howSelect random
  } else {
    set howSelect policy
  }
```
Now, we need to be able to run Run and RunSide so that they can choose the system actions either with the \texttt{epsilonGreedy} method, needed for exploring different policies, and with \texttt{policy}, which is needed to evaluate how well the current policy is doing. Change these two procedures so that they both take an argument for what method to use for choosing the system’s action rule.

Now, we are almost done. Just add in the following procedure, called \texttt{Train} into \texttt{ISAgent3.tcl}. It will run 100 dialogues of simulation using \texttt{epsilonGreedy}, and it will update the Q scores after each dialogue run. After the 100 dialogues have been run, it will update the policy. The group of 100 dialogues, after which the policy is updated, is called an \textit{epoch}.

\texttt{Train} starts epoch 1 with \texttt{::Epsilon} set to 0.40. After each epoch, \texttt{::Epsilon} is multiplied 0.97, so that it is slowly lowered.

Every 10th epoch, \texttt{Train} will evaluate the current policy and print out how it is doing. It will evaluate the current policy by running 500 dialogue simulations following the \texttt{policy} method.

```tcl
proc Train {} {
    set ::Epsilon 0.40
    set ::RunsPerTest 500

    set epoch 1
    while {1} {
        puts [format "Epoch %3d. Epsilon %5.3f" $epoch $::Epsilon]
        for {set r 0} {$r < 100} {incr r} {
            Run epsilonGreedy
            UpdateQ
        }
        UpdatePolicy

        if {[(expr $epoch % 10) == 0]} {
          set sumC 0
          set worseC ""
          for {set r 0} {$r < ::RunsPerTest} {incr r} {
              set c [Run policy]
              set sumC [expr $sumC + $c]
              if {($worseC == "" || $c > $worseC)} {
                set worstC $c
                set worstH ::DialogHistory
              }
          }
          set c [expr $sumC*1.0/$::RunsPerTest]
          puts "Test: $::RunsPerTest runs. Cost is $c"
        }
    }
}
```
foreach i $worstH {
    if {[lindex $i 0] == "A"} {
        puts "\n RL State for A: [lindex $i 1]"
    }
    puts "[lindex $i 0] [lindex $i 2]"
}
incr epoch
set ::Epsilon [expr ::Epsilon * 0.97]
}

Now, run Train.

**Writeup 8.12** Discuss how the dialogue cost is changing over the tests. For the first test, does it do better than the random policy you had did earlier? What is its initial policy? How many epochs does it take to do better than the hand-crafted policy? How does the learned policy seem to differ from the hand-crafted policy?

**Writeup 8.13** Why was it important to do the test on 500 dialogues, rather than a smaller number?