Part VII: Reinforcement Learning

Let us quickly review the type of operator preferences you have used thus far in the tutorial. First, acceptability (+): only acceptable operators are considered for application. Acceptable preferences must be further described as either relatively differentiated or indifferent. Differentiated preferences (such as > and <) establish a relative ordering from which Soar will choose the most preferred operator. If all preferences are labeled as indifferent (=), Soar's random operator choice can be further affected by numerical indifferent parameters.

Reinforcement learning (RL) in Soar allows agents to alter behavior over time by dynamically changing numerical indifferent preferences in procedural memory in response to a reward signal. This learning mechanism contrasts starkly with chunking. Whereas chunking is a one-shot form of learning that increases agent execution performance by summarizing sub-goal results, RL is an incremental form of learning that probabilistically alters agent behavior.

1. Reinforcement Learning in Action

Before we get to the nuts and bolts of RL, consider first an example of its effects. Left-Right is a simple agent that can choose to move either left or right. Unbeknownst to this agent, one direction is more preferable than the other. After deciding its destination, the agent will receive a "reward," or feedback regarding how good a decision it made. In this case, it will receive a reward of -1 for moving left and +1 for moving right. Using RL, the agent will learn quickly that moving right is preferable to moving left.

1.1 The Left-Right Agent

Given the description above, let's begin creating the Left-Right agent. This agent will have a *move* operator that will choose between moving left and right. Because the agent does not know a priori which direction is better, the agent will be indifferent as to the preference between these actions. As you are unfamiliar with the particulars of RL agent design, either type the following code into your favorite editor or open the VisualSoar *left-right* project in the *Agents* directory:

Initialization

The agent stores directions and associated reward on the state

```
sp {propose*initialize-left-right
   (state <s> ^superstate nil
             -^name)
-->
   (<s> ^operator <o> +)
   (<o> ^name initialize-left-right)
}

sp {apply*initialize-left-right
   (state <s> ^operator <op>)
   (<op> ^name initialize-left-right)
-->
   (<s> ^name left-right
        ^direction <d1> <d2>
        ^location start)
   (<d1> ^name left ^reward -1)
   (<d2> ^name right ^reward 1)
}
```

Move

The agent can move in any available direction. The chosen direction is stored on the state.

```
sp {left-right*propose*move
   (state <s> ^name left-right
              ^direction.name <dir>
              ^location start)
-->
   (<s> ^operator <op> +)
   (<op> ^name move
         ^dir <dir>)
}

sp {left-right*rl*left
   (state <s> ^name left-right
              ^operator <op> +)
   (<op> ^name move
         ^dir left)
-->
   (<s> ^operator <op> = 0)
}

sp {left-right*rl*right
   (state <s> ^name left-right
              ^operator <op> +)
   (<op> ^name move
         ^dir right)
-->
   (<s> ^operator <op> = 0)
}

sp {apply*move
   (state <s> ^operator <op>
              ^location start)
   (<op> ^name move
         ^dir <dir>)
-->
   (<s> ^location start - <dir>)
   (write (crlf) |Moved: | <dir>)
}
```

Reward

When an agent chooses a direction, it is afforded the respective reward.

```
sp {elaborate*reward
   (state <s> ^name left-right
              ^reward-link <r>
              ^location <d-name>
              ^direction <dir>)
   (<dir> ^name <d-name> ^reward <d-reward>)
-->
   (<r> ^reward.value <d-reward>)
}
```

Conclusion

When an agent chooses a direction, the task is over and the agent halts.

```
sp {elaborate*done
   (state <s> ^name left-right
              ^location {<> start})
-->
   (halt)
}
```

You will notice a number of unexpected elements in the code above, namely the *rl* rules for the *move* operator and the reward elaboration. The reasons for these components will be made clear in later sections.

1.2 Running the Left-Right Agent

Start the Soar Debugger and load the source for the Left-Right agent. By default, reinforcement learning is disabled. To enable this learning mechanism, enter the following commands:

```
rl --set learning on
indifferent-selection –-epsilon-greedy
```

Note that these commands have been added to the *_firstload* file of the included *left-right* project. The first command turns learning on, while the second sets the exploration policy (more on this later.)

Next, click the "Step" button. This will run Soar through the first cycle. You will note initialization has been chosen, no surprise. In the debugger, execute the following command:

```
print --rl
```

This command shows you the numerical indifferent preferences in procedural memory subject to RL updating. The output is presented here:

```
left-right*rl*right  0.  0
left-right*rl*left  0.  0
```

This result shows that the preference for the two operator instances after 0 updates have a value of 0. Click "Step" two more times, then execute *print --rl* again, to see RL in action:

```
left-right*rl*right  1.  0.3
left-right*rl*left  0.  0
```

After applying the move operator, the numerical indifference value for the rule associated with moving right has now been updated 1 time to a value of 0.3. Note that since the move preferences are indifferent, and thus the decision process is made probabilistically, your agent may have decided to move left instead of right. In this case the *left-right*rl*left* preference would have been updated 1 time with a value of -0.3.

Now click the "Init-soar" button. This will reinitialize the agent. Execute *print --rl*. Notice that the numeric indifferent values have not changed from the previous run. Storing these values between runs is the method by which RL agents learn. Run the agent 20 more times, clicking the "Init-soar" button after each halted execution. You should notice the numeric indifference value for moving right increasing, while the value for moving left decreases. Correspondingly, you should notice the agent choosing to move left less frequently.

2. Building a Learning Agent

Conversion of most agents to take advantage of reinforcement learning takes part in three stages: (1) use RL rules, (2) implement one or more reward rules, and (3) enable the reinforcement learning mechanism. As an example, we will update the basic Water-Jug agent from Tutorial Part I to take advantage of RL functionality. The modified code can be found in the *Agents* directory.

2.1 RL Rules

Rules that are recognized as updateable by the RL mechanism must abide by a specific syntax:

```
sp {my*proposal*rule
   (state <s> ^operator <op> +
              ^condition <c>)
-->
   (<s> ^operator <op> = 2.3)
}
```

The name of the rule can be anything and the left-hand side (LHS) of the rule, the conditions, may take any form. However, the right-hand side (RHS) must take the following form:

```
(<s> ^operator <op> = number)
```

To be specific, the RHS can only have one action, asserting a single numeric indifferent preference, and *number* must be a numeric constant value (such as *2.3* in the example above). Any other actions, including proposing acceptability of the operator, must take place in a separate rule.

Recalling the Water Jug problem, our goal will be to have the agent learn the best conditions under which to empty a jug (of particular volume), fill a jug (of particular volume), and pour one jug (of particular volume) to another. Thus we will modify the *empty*, *fill*, and *pour* operators to afford them RL updatable rules.

Modifying the Water-Jug agent's rules to make them compatible with RL will take two steps: (a) modify the existing proposal rules and (b) creating new RL rules. Modification of the existing proposal rule is trivial: simply remove the "=" (equal) sign from the operator preference assertion action on the RHS:

```
sp {water-jug*propose*empty
   (state <s> ^name water-jug
              ^jug <j>)
   (<j> ^contents > 0)
-->
   (<s> ^operator <o> +)
   (<o> ^name empty
        ^empty-jug <j>)}

sp {water-jug*propose*fill
   (state <s> ^name water-jug
              ^jug <j>)
   (<j> ^empty > 0)
-->
   (<s> ^operator <o> +)
   (<o> ^name fill
        ^fill-jug <j>)}




sp {water-jug*propose*pour
   (state <s> ^name water-jug
              ^jug <i> { <><i><j> })
   (<i> ^contents > 0 )
   (<j> ^empty > 0)
-->
   (<s> ^operator <o> +)
   (<o> ^name pour
        ^empty-jug <i>
        ^fill-jug <j>)}
```

To be clear, these modified rules propose their respective operators with an acceptable preference. Next, we will write RL rules whose conditions detect these acceptable preferences and compliment with numeric indifferent preferences.

The second step of agent modification can be much more laborious. In order for RL to provide feedback for each action in each state of the problem, it must have an RL rule for each state-action pair. In the Water Jug problem, a state can be represented by the volume of each of the jugs and the action (empty, fill, or pour) with one of the two jugs. As an example, one RL rule for the emptying the 3-unit jug (currently storing 2 units) when the 5-unit jug has 4 units could be written as follows:

```
sp {water-jug*empty*3*2*4
   (state <s> ^name water-jug
              ^operator <op> +
              ^jug <j1> <j2>)
   (<op> ^name empty
         ^empty-jug.volume 3)
   (<j1> ^volume 3
         ^contents 2)
   (<j2> ^volume 5
         ^contents 4)
-->
   (<s> ^operator <op> = 0)
}
```

For simple agents, like the Left-Right agent above, enumerating all state-action pair as RL rules by hand is plausible. However, the Water-Jug agent requires (3 * 2 * 4 * 6) = 144 RL rules to fully represent this space. Since we can express these rules as the output of a simple combinatorial pattern, we will use the Soar *gp* command to generate all the rules we need:

```
gp {rl*water-jug*empty
   (state <s> ^name water-jug
              ^operator <op> +
              ^jug <j1> <j2>)
   (<op> ^name empty
         ^empty-jug.volume [3 5])
   (<j1> ^volume 3
         ^contents [0 1 2 3])
   (<j2> ^volume 5
         ^contents [0 1 2 3 4 5])
-->
   (<s> ^operator <op> = 0)
```

```
        }
```

```
        gp {rl*water-jug*fill
           (state <s> ^name water-jug
                      ^operator <op> +
                      ^jug <j1> <j2>)
           (<op> ^name fill
                 ^fill-jug.volume [3 5])
           (<j1> ^volume 3
                 ^contents [0 1 2 3])
           (<j2> ^volume 5
                 ^contents [0 1 2 3 4 5])
        -->
           (<s> ^operator <op> = 0)
        }
```

```
        gp {rl*water-jug*pour
           (state <s> ^name water-jug
                      ^operator <op> +
                      ^jug <j1> <j2>)
           (<op> ^name pour
                 ^empty-jug.volume [3 5])
           (<j1> ^volume 3
                 ^contents [0 1 2 3])
           (<j2> ^volume 5
                 ^contents [0 1 2 3 4 5])
        -->
           (<s> ^operator <op> = 0)
        }
```

Note that had the rules required a more complex pattern for generation, or had we not known all required rules at agent design time, we would have made use of rule *templates* (see the Soar Manual for more details).

2.2 Reward Rules

Reward rules are just like any other Soar rule, except that they modify the *reward-link* structure of the state to reflect reward associated with the agent's current operator decision. Reward values must be stored on the *value* element of the *reward* attribute of the *reward-link* identifier (state.reward-link.reward.value).

Of significant note, Soar does not remove or modify structures within the *reward-link*, including old reward values. It is the agent's responsibility to maintain the *reward-link* structure to reflect proper feedback to the RL mechanism. In most cases, this means reward rules will be i-supported, such as to create non-persistent reward values. If an attribute remains on the *reward-link* structure, such as through an o-supported rule, the reward will count multiple times in the reinforcement learning.

For the Water-Jug agent, we will provide reward only when the agent has achieved the goal. This entails making a minor modification to the goal-test rule:

```
        sp {water-jug*detect*goal*achieved
           (state <s> ^name water-jug
                      ^jug <j>
```

```
                 ^reward-link <rl>)
   (<j> ^volume 3 ^contents 1)
-->
   (write (crlf) |The problem has been solved.|)
   (<rl> ^reward.value 10)
   (halt)}
```

Now load this code into the debugger and run it a few times (if loading your own code, remember to enable reinforcement learning). After about five runs you should find that the agent has adopted a near optimal strategy. At any point during the runs you can execute the *print --rl* command to see the numeric indifferent values of the RL rules generated by the *gp* command. You can right-click and print any of these rules to see their conditions.

3. Further Exploration

Consider the following output from a run (watch level 0) of the learning left-right agent from section 1:

```
run
Moved: right
This Agent halted.
An agent halted during the run.

init-soar
Agent reinitialized.

run
Moved: right
This Agent halted.
An agent halted during the run.

init-soar
Agent reinitialized.

run
Moved: left
This Agent halted.
An agent halted during the run.
```

You should notice that at run 3 moving left is selected. By this point moving right has an obvious advantage in numerical preference values, thus why is left chosen? The answer lies with exploration policies.

There are times in learning when exploration of operations currently considered less-than-preferred may lead you down a useful path. Soar allows you to tune your level of exploring these alternate paths using the *indifferent-selection* command.

In the Soar Debugger, type "*indifferent-selection --stats*" (sans quotes). The result should look like this:

```
Exploration Policy: epsilon-greedy
Automatic Policy Parameter Reduction: off

epsilon: 0.1
epsilon Reduction Policy: exponential
epsilon Reduction Rate (exponential/linear): 1/0

temperature: 25
temperature Reduction Policy: exponential
temperature Reduction Rate (exponential/linear): 1/0
```

This command prints the current exploration policy as well as a number of tuning parameters. There are five exploration policies: *boltzmann*, *epsilon-greedy*, *softmax*, *first*, and *last*. You can change the exploration policy by issuing the following command (where "policy_name" should be replaced with one of the policies above):

```
indifferent-selection --policy_name
```

This tutorial will only discuss the *epsilon-greedy* policy. For information on the other policies you should read the Soar manual. Epsilon greedy is a policy common within reinforcement learning experimentation to allow parameter-controlled exploration of operators not currently recognized as most preferred. This policy is controlled by the *epsilon* parameter. The policy is summarized as such:

```
With ( 1 - epsilon ) probability, the most preferred operator is to be
chosen.  With epsilon probability, a random selection of all
indifferent operators is made.
```

When Soar is first started, the default exploration policy is *softmax*. However, the first time RL is enabled, the architecture automatically changes the exploration policy to *epsilon-greedy*, a policy more suitable for RL agents. The default value of *epsilon* is 0.1, dictating that 90% of the time the operator with greatest numerical preference value is chosen, while the remaining 10% of the time a random selection is made from all acceptable proposed operators. You can change the *epsilon* value by issuing the following command:

```
indifferent-selection --epsilon <value>
```

Acceptable values for *epsilon* are numbers between 0 and 1 (inclusive). You may note, by the definition, that a value of 0 will eliminate the chance of exploration and a value of 1 will result in a uniformly random selection.

With this explanation, you should experiment with different values of epsilon during different runs in the agents discussed in this tutorial.