Unit 6: Selecting Productions on the Basis of Their Utilities and Learning these Utilities

Occasionally, we have had cause to set parameters of productions so that one production will be preferred over another in the conflict resolution process. Now we will examine how production rule utilities are computed and used in conflict resolution. We will also look at how these utilities are learned.

6.1 The Theory

There can be multiple productions that match the buffers' current contents and the issue arises of which production to select to fire. Each production has associated with it a utility, which reflects how much the production is expected to contribute to achieving the model's current objective. The utility of a production *i* is defined as:

$$U_i = P_i G - C_i$$

This is calculated from three quantities:

P_i: The expected probability that production i firing will lead to a successful completion of the current objective. The objective is considered complete when a production that is marked as being either a success or a failure fires.

C_i: Expected cost of achieving that objective. Cost is measured in time and C is an estimate of the time from when the production is selected until the objective is finally completed.

G: Value of the objective. Given that the units of cost are measured in time, so is the value of the objective. G is typically set to 20 seconds, which is the default value.

For example, if $P_i = .9$, $G = 20$, $C_i = 3$ the utility of production i is __.

The values of P and C can be set for each production, or as we will describe they can be learned from experience. The value of G is a global value that is set as the parameter :g with the sgp command.

Among the productions that match, ACT-R will select the production with the highest utility. However, the equation above actually only gives the expected utility. Like activations, utilities have noise added to them so the full equation becomes

$$U_i = P_i G - C_i + \varepsilon$$

The noise, ε, is controlled by the utility noise parameter s which is set with the parameter :egs. The noise is distributed according to a logistic distribution with a mean of 0 and a variance of

$$\sigma^2 = \frac{\pi^2}{3} s^2$$

As with activations for chunks, there is also a threshold which specifies the minimum utility necessary for a production to fire. The utility threshold is set with the :ut parameter.

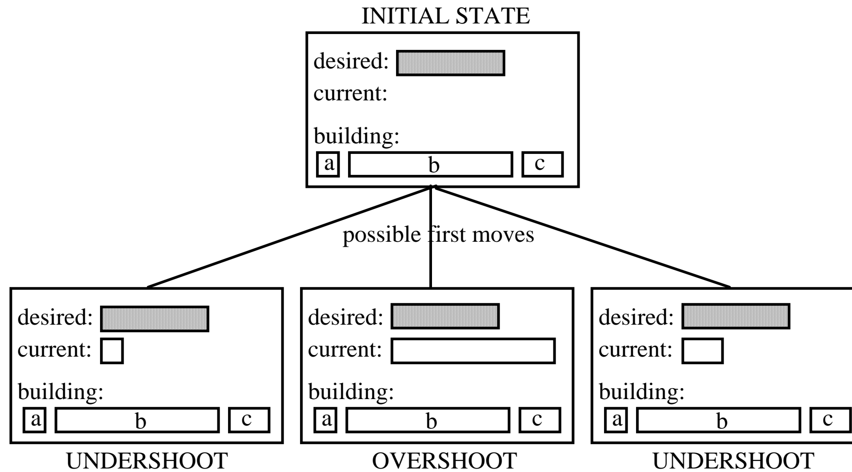
If there are a number of productions competing with expected utility values U_j the probability of choosing production *i* is described by the formula

$$\Pr obability(i) = \frac{e^{U_i/\sqrt{2}s}}{\sum_j e^{U_j/\sqrt{2}s}}$$

where the summation is over all the competing productions (those that match the current buffer contents) including i and the utility threshold.

6.2 Building Sticks Example

We will illustrate these ideas with an example from problem solving. Lovett (1998) looked at participants solving the building-sticks problem illustrated in the figure below. This is an isomorph of Luchins waterjug problem that has a number of experimental advantages. Participants are given an unlimited supply of building sticks of three lengths and are told that their objective is to create a target stick of a particular length. There are two basic strategies they can select – they can either start with a stick smaller than the desired length and add sticks (like the addition strategy in Luchins waterjugs) or they can start with a stick that is too long and "saw off" lengths equal to various sticks until they reach the desired length (like the subtraction strategy). The first is called the undershoot strategy and the second is called the overshoot strategy. Subjects show a strong tendency to hillclimb and choose as their first stick a stick that will get them closest to the target stick.



You can go through a version of this by opening the model **bst-nolearn** in your unit6 folder. By evoking the command (do-set) you will be presented with a pair of problems:

```
? (do-set)
(UNDER OVER)
```

It returns a list of the solutions you initially tried on each of the problems, and in this version of the task there are only two problems. As it turns out both of these problems can only be solved by the overshoot strategy. However, the first one looks like it can be solved more easily by the undershoot strategy. The exact lengths of the sticks in pixels are:

A = 15 B = 200 C = 41 Goal = 103

The difference between B and the goal is 97 pixels while the difference between C and the goal is only 62 pixels – a 35 pixel difference of differences. However, the only solution to the problem is B – 2C – A. The same solution holds for the second problem:

A = 10 B = 200 C = 29 Goal = 132

But in this case the difference between B and the goal is 68 pixels while the difference between C and the goal is 103 pixels – a 35 pixel difference of differences in the other direction. You can run the model on these problems and it will tend to choose under for the first and over for the second but not always. One can run it multiple times by calling the function collect-data with its argument being the number of runs. The following is the outcome of 100 trials:

```
? (collect-data 100)
(25 73)
```

where the two numbers in the list returned are the number of times overshoot was chosen on the first problem and the second problem respectively.

The model for the task involves a good number of productions for encoding the screen and selecting sticks. However, the behavior of the model is really controlled by four production rules that make the decision to apply the overshoot or undershoot strategy.

```
(p decide-over
   =goal>
     isa      try-strategy
     state    choose-strategy
     strategy nil
     under    =under
   < over     (!eval! (- =under 25))
==>
   =goal>
     state    prepare-mouse
     strategy over
   +visual-location>
     isa      visual-location
     kind     oval
     screen-y 60)

(p force-over
   =goal>
     isa      try-strategy
     state    choose-strategy
   - strategy over
==>
   =goal>
     state    prepare-mouse
     strategy over
   +visual-location>
     isa      visual-location
     kind     oval
     screen-y 60)


(p decide-under
   =goal>
     isa      try-strategy
     state    choose-strategy
     strategy nil
     over     =over
   < under    (!eval! (- =over 25))
==>
   =goal>
     state    prepare-mouse
     strategy under
   +visual-location>
     isa      visual-location
     kind     oval
     screen-y 85)

(p force-under
   =goal>
     isa      try-strategy
     state    choose-strategy
   - strategy under
==>
   =goal>
     state    prepare-mouse
     strategy under
```

```
   +visual-location>
     isa      visual-location
     kind     oval
     screen-y 85)
```

The key information is in the slots over, which encodes the pixel difference between the stick b and the goal, and under, which encodes the difference between the goal and stick c. These values have been computed by prior productions that encode the problem. If one of these differences is more than 25 pixels less than the other, then decide-under or decide-over can fire to choose the strategy. In all situations, the other two productions, force-under and force-over, can apply. Thus, if there is a clear difference in how close the two sticks are to the goal there will be three productions (one decide, two force) that can apply and if there is not then just the two force productions can apply. The choice among the production rules is determined by their relative utilities which we can see in the **Procedural Memory Viewer** window, or by using the spp command:

```
? (spp force-over force-under decide-over decide-under)
 Parameters for production Force-Over:
 :Chance  1.000
 :Effort  0.050
 :P  0.500
 :C  0.050
 :PG-C  9.950

 Parameters for production Force-Under:
 :Chance  1.000
 :Effort  0.050
 :P  0.500
 :C  0.050
 :PG-C  9.950

 Parameters for production Decide-Over:
 :Chance  1.000
 :Effort  0.050
 :P  0.650
 :C  0.050
 :PG-C 12.950

 Parameters for production Decide-Under:
 :Chance  1.000
 :Effort  0.050
 :P  0.650
 :C  0.050
 :PG-C 12.950

(Force-Over Force-Under Decide-Over Decide-Under)
```

The only differences among the productions are the values of P which were set by the spp command in the **Commands** window.

```
(spp decide-over :p .65)
(spp decide-under :p .65)
(spp force-over :p .5)
(spp force-under :p .5)
```

The P parameters for the force productions are .50 while they are a more optimistic .65

for the decide productions. With G set at the default value of 20 this leaves a difference of 3 between the PG-C values for the decide and choose productions.

Lets consider how these productions apply in the case of the two problems in the model. Since the difference between the under and over differences is 35 pixels, there will be one decide and two force productions that match the buffers. Let us consider the probability of choosing each production according to the equation.

$$\Pr obability(i) = \frac{e^{U_i/\sqrt{2}s}}{\sum_j e^{U_j/\sqrt{2}s}}$$

The parameter s is set at 3 and the utility threshold is set to -100 (we want the probability that none of the productions are over the threshold to be essentially 0). First, consider the probability of the decide production:

$$\Pr obability(decide) = \frac{e^{12.95/4.24}}{e^{12.95/4.24} + e^{9.95/4.24} + e^{9.95/4.24} + e^{-100/4.24}}$$

$$= \frac{e^{3/4.24}}{e^{3/4.24} + e^{0} + e^{0} + e^{-109.95/4.24}} = .504$$

Similarly, the probability of the two force productions can be shown to be .248. Thus, there is a .248 probability that a force production will fire that has the model try to solve the problem in the direction other than it appears.

6.3 Parameter Learning

So far we have only considered the situation where the production parameters are static. However, they will change as experience is gathered about the relative costs of different methods and their relative probabilities of success. The probability of success of a production is calculated as

$$\mathrm{P} = \frac{\text{Successes}}{\text{Successes} + \text{Failures}}$$

where Successes and Failures are the number of experienced successes and failures. A success or failure occurs when a production explicitly tagged as a success or a failure fires. In the bst models there is one production that recognizes failure and starts over

again and another production that recognizes success. One has the failure flag set to t and the other has the success flag set to t by a spp command:

```
(p pick-another-strategy
   =goal>
     isa      try-strategy
     state    wait-for-click
   =manual-state>
     isa      module-state
     modality free
   =visual-location>
     isa      visual-location
   > screen-y 100
==>
   =goal>
      state choose-strategy)

 (p read-done
   =goal>
     isa      try-strategy
     state    read-done
   =visual>
     isa      text
     value    "done"
==>
   +goal>
     isa      try-strategy
     state    start)

(spp read-done :success t)
(spp pick-another-strategy :failure t)
```

When such a production fires all the productions that have fired since the last marked production fired are credited with a success or failure.

A similar equation governs the learning of the cost:

$$\mathbf{C} = \frac{\text{Efforts}}{\text{Successes} + \text{Failures}}$$

where Efforts is the accumulated time over all the successful and failed applications of this production rule. The time for a particular success or failure credited to a production that is not the one marked as a success or failure is the difference in time between that production's selection and the selection time of the marked production. The time credited to a marked production is its effort – the amount of time it takes to fire.

Productions have initial values of the parameters Efforts, Successes, and Failures at the beginning of a run. By default each production rule is created with Efforts = .05 seconds (the cost of one firing), Successes = 1, and Failures = 0. This means that the default

value of P is 1 and the default value of C is .05 second. However, as we will see in the next section it is often necessary to set these to non-default values to reflect prior experience or biases. These prior values can be set with the spp command as shown below:

```
(spp decide-over :failures 7 :successes 13 :efforts 100)
(spp decide-under :failures 7 :successes 13 :efforts 100)
(spp force-over :failures 10 :successes 10 :efforts 100)
(spp force-under :failures 10 :successes 10 :efforts 100)
```

It is also possible to set the initial values for all of the productions by omitting a production name in the call to spp:

```
(spp :efforts 500 :successes 100)
```

6.4. Learning in the Building Sticks Task

Lovett did an experiment with a building sticks task. The following are the percent choice of overshoot for each of the problems in the training set from Lovett & Anderson (1996):

Lovett, M. C., & Anderson, J. R. (1996). History of success and current context in problem solving: Combined influences on operator selection. *Cognitive Psychology, 31,* 168-217.

a	b	c	Goal	%OVERSHOOT
15	250	55	125	20
10	155	22	101	67
14	200	37	112	20
22	200	32	114	47
10	243	37	159	87
22	175	40	73	20
15	250	49	137	80
10	179	32	105	93
20	213	42	104	83
14	237	51	116	13
12	149	30	72	29
14	237	51	121	27
22	200	32	114	80
14	200	37	112	73
15	250	55	125	53

The majority of these problems look like they can be solved by undershoot and in some cases the pixel difference is greater than 25. However, the majority of the problems can only be solved by overshoot. The first and last problems are interesting because they are identical and look strongly like they are undershoot problems. It is the only problem that can be solved either by overshoot or undershoot. Only 20% of the participants solve the first problem by overshoot but after the sequence of problems this rises to 53% for the last problem.

The model **bst-learn** is the one that simulates this experiment. This is the same as the model in **bst-nolearn** except that the learning mechanism is enabled (the :pl parameter is t) and all of the stimuli are presented by do-set. When the learning is on, we do not set the values of P and C directly. Instead, we set the parameters of the critical productions to have prior values of Successes, Failures and Efforts to produce the desired initial values of P and C:

```
? (spp force-over force-under decide-over decide-under)
 Parameters for production Force-Over:
 :Chance  1.000
 :Effort  0.050
 :P  0.500
 :C  5.000
 :PG-C  5.000
 :Successes   (10)
 :Failures   (10)
 :Efforts  (100)
 :Success    nil
 :Failure    nil

 Parameters for production Force-Under:
 :Chance  1.000
 :Effort  0.050
 :P  0.500
 :C  5.000
 :PG-C  5.000
 :Successes   (10)
 :Failures   (10)
 :Efforts  (100)
 :Success    nil
 :Failure    nil

 Parameters for production Decide-Over:
 :Chance  1.000
 :Effort  0.050
 :P  0.650
 :C  5.000
 :PG-C  8.000
 :Successes   (13)
 :Failures    (7)
 :Efforts  (100)
 :Success    nil
 :Failure    nil

 Parameters for production Decide-Under:
 :Chance  1.000
 :Effort  0.050
 :P  0.650
 :C  5.000
 :PG-C  8.000
 :Successes   (13)
 :Failures    (7)
 :Efforts  (100)
 :Success    nil
 :Failure    nil

(Force-Over Force-Under Decide-Over Decide-Under)
```

The following is the performance of the model on a 100 simulation run:

```
? (collect-data 100)
CORRELATION:  0.733
MEAN DEVIATION: 19.701

Trial 1    2    3    4    5    6    7    8    9    10   11   12   13   14   15
     25   46   52   63   83   32   60   73   38   36   32   35   67   65   37

DECIDE-OVER : 0.6578
DECIDE-UNDER: 0.6600
FORCE-OVER  : 0.6457
FORCE-UNDER : 0.4743
```

Also printed out are the average values of P for the critical productions after each run through the experiment over these 100 runs. As can be seen, the two decide productions retain their estimates of about 65% success and the force-under production retains its estimate of about 50% success. However, the system has learned that the force-over production is more generally successful -- about 65%. Here are the actual production parameters after one run through the experiment:

```
? (spp force-over force-under decide-over decide-under)
 Parameters for production Force-Over:
 :Chance  1.000
 :Effort  0.050
 :P  0.636
 :C  5.156
 :PG-C  7.571
 :Successes (21.0)
 :Failures (12.0)
 :Efforts (170.163)
 :Success    nil
 :Failure    nil

 Parameters for production Force-Under:
 :Chance  1.000
 :Effort  0.050
 :P  0.500
 :C  4.873
 :PG-C  5.127
 :Successes (13.0)
 :Failures (13.0)
 :Efforts (126.69200000000001)
 :Success    nil
 :Failure    nil

 Parameters for production Decide-Over:
 :Chance  1.000
 :Effort  0.050
 :P  0.650
 :C  5.000
 :PG-C  8.000
 :Successes   (13)
 :Failures    (7)
 :Efforts  (100)
 :Success    nil
 :Failure    nil

 Parameters for production Decide-Under:
 :Chance  1.000
 :Effort  0.050
```

```
:P  0.636
:C  4.951
:PG-C  7.776
:Successes (14.0)
:Failures  (8.0)
:Efforts (108.928)
:Success    nil
:Failure    nil
```

The values for the force productions had been 10 successes and 10 failures at the beginning of the run. In the case of force-over the system has experienced 11 more successes and only 2 more failures leading to totals of 21 and 12 and the more optimistic estimate of P of .636. In the case of force-under the system has experienced 3 additional successes and 3 additional failures leaving the estimate of P unchanged. The values for the decide productions had been 13 successes and 7 failures. In this run the decide-over production was never tried and so its values are unchanged. The decide-under production had been tried twice with one success and one failure leaving the values at 14 successes and 8 failures and a slightly reduced P value of .636.

6.5 Learning in a Probability Choice Experiment

Your assignment is to develop a model for a "probability matching" experiment run by Friedman et al (1964). The difference between this assignment and earlier ones is that you are responsible for almost all of the code for the model, including the code which presents the experiment. The experiment to be implemented is very simple. The basic procedure, which is repeated for 48 trials, is:

1. The participant is presented with a screen saying "Choose"
2. The participant either types H for heads or T for tails
3. The screen is cleared and presents as feedback the correct answer, either "Heads" or "Tails" for 1 second.

Friedman et al arranged it so that heads was the correct choice on 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, and 90% of the trials (independent of what the participant had done). For your experiment you will only be concerned with the 90% condition. Thus, your experiment will be 48 trials and "Heads" will be the correct answer 90% of the time. We have averaged together the data from the 10% and 90% conditions (flipping responses) to get an average proportion of choice of the dominant answer in each block of 12 trials. These proportions are 0.72, 0.78, 0.82, and 0.84. This is the data that your model is to fit. Note, this is not the percentage of correct responses – the correctness of the response does not matter. Your model must begin with a 50% chance of saying heads. Then, rapidly adjust its probabilities so that it averages close to 72% over the first block of 12 trials, and increases to about 84% by the final block. You will run the model through the experiment many times (resetting before each experiment) and average the data of those runs for comparison. As an aspiration level, this is the performance of the model that I wrote, averaged over 100 runs:

```
? (collect-data 100)
```

```
CORRELATION:  0.959
MEAN DEVIATION:  0.026
 Original     Current
   0.720       0.679
   0.780       0.785
   0.820       0.801
   0.840       0.817
```

In achieving this, the parameters I worked with were the noise in the utilities (set by the :egs parameter) and the initial number of successes and failures that I gave to the productions that chose heads and tails -- just as this is what I did in the case of the building sticks model.

The starting model you are given for this task, **choice**, contains only the functions necessary to run a person through one trial and to collect a key press response using the "trial at a time" experiment writing style. When either a person or the model presses a key, the string representing that key will be saved in the global variable ***response***, and the function **do-trial-person** will run one trial returning the key that was pressed. You will have to write a similar function to run the model through one trial, which should be named **do-trial-model**. You also need to write a function called **collect-data** that takes one parameter and runs the experiment that many times and prints out the average results of the runs and the correlation and deviation of the average data to the experimental data. You also must write the model for the task that fits the data.

My suggestion would be to first write the **do-trial-model** function and a model that does the task (without trying to fit the data), and make sure that works correctly. Next write a function to run a block of 12 trials and test that to make sure the model works correctly between trials. Then write a function to iterate over 4 blocks for running one pass of the experiment and test that. After that is working write the **collect-data** function to run the experiment multiple times. Only then should you be concerned with actually fitting the model to the data, once you are sure everything else works.

To write the experiment for the model to interact with you will need to use a few ACT-R functions that were discussed in the experiment description files. Those functions will be described again here, and the models should provide plenty of examples of their use.

The **reset** function initializes ACT-R. It returns the model to the initial state as specified in the model file. It is the programmatic equivalent of pressing the **Reset** button in the environment.

The function **pm-install-device** takes one parameter which should be a window. That parameter tells ACT-R/PM with which window the model will be interacting. Everything in that window can be seen by the model, and all of the model's motor actions (key presses and mouse clicks) will affect that window.

The **pm-proc-display** function is called to make the model "look" at the window. The model only encodes the screen when requested with a call to **pm-proc-display**. Thus, for the model to notice a change to the window **pm-proc-display** must be called after the

change has occurred. This function performs the buffer stuffing of the visual-location buffer if it is empty and triggers the re-encoding if the model is attending an item.

To run the model, use the **pm-run** function. It has one required parameter, the maximum amount of time to run the model, and one optional keyword parameter called :full-time. If :full-time is specified as t then the model will run for the entire time specified. Otherwise, the model will stop immediately when there is nothing more it can do, which may be prior to the end of the specified running time.

In addition to these functions there are the **correlation** and **mean-deviation** functions that you will need to use. These calculate the correlation and mean-deviation between two lists of numbers.

Here is the function that runs the model through the paired associate task from unit 4:

```
(defun do-experiment-model (size trials)
  (let ((result nil)
        (window (open-exp-window "Paired-Associate Experiment" :visible nil)))

    (reset)
    (pm-install-device window)

    (dotimes (i trials)
      (let ((score 0.0)
            (time 0.0)
            (start-time))
        (dolist (x (permute-list (subseq *pairs* (- 20 size))))

          (clear-exp-window)
          (add-text-to-exp-window :text (car x) :x 150 :y 150)

          (setf *response* nil)
          (setf *response-time* nil)
          (setf start-time (pm-get-time))

          (pm-proc-display)
          (pm-run 5.0 :full-time t)

          (when (equal (second x) *response*)
            (incf score 1.0)
            (incf time (- *response-time* start-time)))

          (clear-exp-window)
          (add-text-to-exp-window :text (second x) :x 150 :y 150)

          (pm-proc-display)
          (pm-run 5.0 :full-time t))

        (push (list (/ score size) (and (> score 0) (/ time (* score 1000.0))))
              result)))

    (reverse result)))
```

It is more complicated than the function you will need for this assignment because it is recording response times and averaging the data over multiple runs which your **do-trial-model** function will not be doing. It also calls **reset** which you should not do in your **do-**

trial-model function because you want the model to continue to learn from trial to trial. You should only call **reset** at the start of each pass through the whole experiment. However, it does have a similar sequence of operations. If we ignore the averaging and response times the function opens a window, presents an item of text, runs the model, clears the screen, displays another item of text and then runs the model again (the code highlighted in red). The provided **do-trial-person** function provides the structure for the displaying of the items in the choice task:

```
(defun do-trial-person ()
  (let ((window (open-exp-window "Choice Experiment" :visible t)))

    (add-text-to-exp-window :text "choose" :x 50 :y 100)

    (setf *response* nil)

    (while (null *response*)
           (allow-event-manager window))

    (clear-exp-window)

    (add-text-to-exp-window :text (if (< (random 1.0) .9) "heads" "tails")
                            :x 50 :y 100)

    (sleep 1.0)
    *response*))
```

What you must do is write the **do-trial-model** function that presents the display as **do-trial-person** does, but has the appropriate interaction with ACT-R (the code colored green is not the interaction necessary for ACT-R to do the task).

It is also possible to write the experiment using an event-based style as discussed in the unit4 experiment documentation. That will require a little more work to program because it does not analogize as neatly to an existing model. If you would like to write the experiment in that way you should look at the Zbrodoff model as an example instead of the paired model as described above. In fact, the different ways to write the experiment will actually have an impact on the data fitting for this model because they will have slightly different timing on the events which will affect the efforts experienced by the productions. For the paired associate task the style of the experiment was not an issue because the lengths of the trials were fixed, but in this case, because the trials transition on the response of the model, the event-based experiment will provide a more veridical timing sequence because the events of the experiment are not impacted by components of the model other than its response. However, either solution is acceptable for the assignment.