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 utilities are computed and used in conflict resolution. We will also look at how these utilities are learned.

6.1 The Utility Theory

Each production has a utility associated with it which can be set directly as we have seen in some of the previous units. Like activations, utilities have noise added to them. The noise 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

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

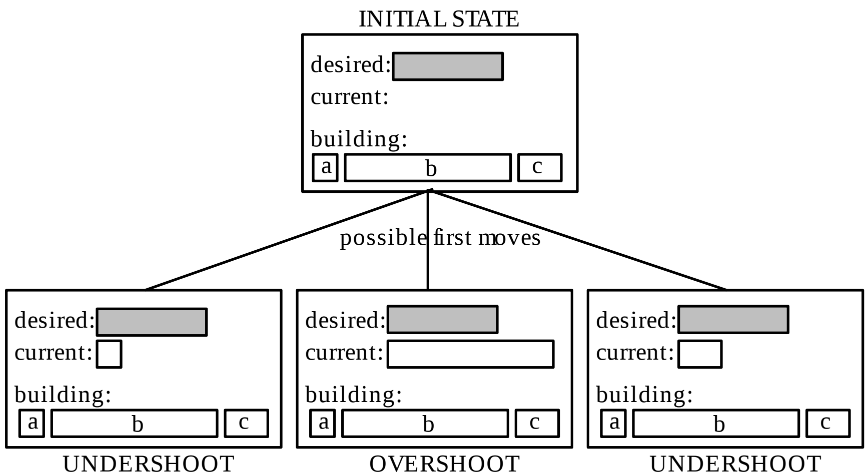
If there are a number of productions competing with expected utility values U_j the probability of choosing production *i* is described by the formula

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

where the summation *j* is over all the productions which currently have their conditions satisfied. Note however that that equation only serves to describe the production selection process. It is not actually computed by the system. The production with the highest utility (after noise is added) will be the one chosen to fire.

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). We will call the first of those the undershoot strategy and the second 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.



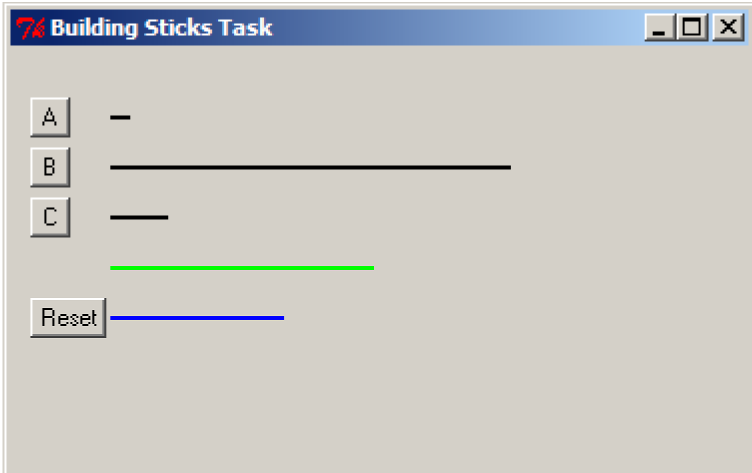
You can go through a version of this task that was written to work with ACT-R models by loading the bst.lisp file in Lisp or importing the bst file in Python. The function bst-test in Lisp or test in the bst module in Python takes one parameter indicating how many sample pairs of problems to present and an optional second parameter which indicates whether a person or model is performing the task. The default is to run the model, so to run yourself through one pair of problems you would use one of these:

```
? (bst-test 1 t)
```

```
>>> bst.test(1,True)
```

The return value will be a list of two numbers indicating how many times you used the overshoot strategy on the first and second problem from the pair respectively.

The experiment will look something like this:



To do the task you will see four lines initially. The top three are black and correspond to the building sticks you have available. The fourth line is green and that is the target length you are attempting to build. The current stick you have built so far will be blue and below the target stick. You will build the current stick by pressing the button to the left of a stick you would like to use next. If your current line is shorter than the target the new stick will be added to the current stick, and if your current line is longer than the target the new stick will be subtracted from the current stick. When you have successfully matched the target length the word "Done" will appear below the current stick and you will progress to the next trial. At any time you can hit the button labeled Reset to clear the current stick and start over.

As it turns out, both of the problems presented in that test set 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 for that problem 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 undershoot for the first about 75% of the time and overshoot for the second about 75% of the time. You can run the model multiple times using the bst-test or test function with the number of pairs to run the model through to see the results for yourself. If you only run the model through one pair it will perform the task with a visible window so that you can watch it, but if you run more than one pair it will use a virtual window to complete the task faster.

The model for the task involves many productions for encoding the screen and selecting sticks. However, the critical behavior of the model is controlled by four productions that make the decision as to whether to apply the overshoot or the undershoot strategy.

```
(p decide-over
   =goal>
      isa       try-strategy
      state     choose-strategy
      strategy  nil
   =imaginal>
      isa       encoding
      under     =under
      over      =over
   !eval! (< =over (- =under 25))
  ==>
   =imaginal>
   =goal>
      state     prepare-mouse
      strategy  over
   +visual-location>
      isa       visual-location
      kind      oval
      value     "b")


(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
      value     "b")



(p decide-under
```

```
   =goal>
      isa       try-strategy
      state     choose-strategy
      strategy  nil
   =imaginal>
      isa       encoding
      over      =over
      under     =under
   !eval! (< =under (- =over 25))
  ==>
   =imaginal>
   =goal>
      state     prepare-mouse
      strategy  under
   +visual-location>
      isa       visual-location
      kind      oval
      value     "c")


(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
      value     "c")
```

The key information is in the over and under slots of the chunk in the **imaginal** buffer. The over slot encodes the pixel difference between stick b and the target stick, and the under slot encodes the difference between the target stick and stick c. These values have been computed by prior productions that encode the problem. If one of these differences appears to get the model much closer to the target (more than 25 pixels closer than the other) then the decide-under or decide-over productions 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 target stick 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 productions is determined by their relative utilities which we can see using the Procedural tool in the ACT-R Environment, or by using the spp command:

```
? (spp force-over force-under decide-over decide-under)
```

```
>>> actr.spp('force-over','force-under','decide-over','decide-under')
```

The output from that will look like this if the model has not yet performed the task:

```
Parameters for production FORCE-OVER:
 :utility    NIL
 :u  10.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production FORCE-UNDER:
 :utility    NIL
 :u  10.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production DECIDE-OVER:
 :utility    NIL
 :u  13.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production DECIDE-UNDER:
 :utility    NIL
 :u  13.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
```

The productions' current utility values, labeled :u, are set in the model using the spp command:

```
(spp decide-over :u 13)
(spp decide-under :u 13)
(spp force-over :u 10)
(spp force-under :u 10)
```

The :u parameters are set to 10 for the force productions and to 13 for the decide productions since making the decision based on which one looks closer should be preferred to just guessing, at least initially. The :utility value shown in the output from spp indicates the last computed utility value for the production during a conflict-resolution event and includes the utility noise. The value of nil in the output above before the model runs indicates that the production has not yet been used. If you run the model through the task and then check the parameters you will see something like this which shows the noisy utility values for the productions which matched the state and could possibly have been selected during conflict-resolution:

```
Parameters for production FORCE-OVER:
 :utility  8.893
 :u  10.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production FORCE-UNDER:
```

```
 :utility 15.080
 :u  10.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production DECIDE-OVER:
 :utility 15.434
 :u  13.372
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production DECIDE-UNDER:
 :utility    NIL
 :u  13.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
```

Let us 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 for both problems. Let us consider the probability of choosing each production using the equation shown above, and the fact that the noise parameter, *s*, is set to 3 in the model.

First, consider the probability of the decide production:

$$\Pr obability(decide) = \frac{e^{13/4.24}}{e^{13/4.24} + e^{10/4.24} + e^{10/4.24}}$$
$$= \frac{e^{3/4.24}}{e^{3/4.24} + e^{0} + e^{0}} = .504$$

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 Utility Learning

So far we have only considered the situation where the production parameters are static. The utilities of productions can also be learned as the model runs based on rewards that are received by the model. When utility learning is enabled, the productions' utilities are updated according to a simple integrator model (e.g. see Bush & Mosteller, 1955). If U_i(n-1) is the utility of a production *i* after its n-1st application and R_i(n) is the reward the production receives for its nth application, then its utility U_i(n) after its nth application will be:

$$U_i(n) = U_i(n-1) + \alpha\left[R_i(n) - U_i(n-1)\right]$$

where α is the learning rate and is typically set at .2 (this can be changed by adjusting the

:alpha parameter in the model). This is also basically the Rescorla-Wagner learning rule (Rescorla & Wagner, 1972). According to this equation the utility of a production will be gradually adjusted until it matches the average reward that the production receives.

There are a couple of things to mention about the rewards. The rewards can occur at any time, and are not necessarily associated with any particular production. Also, a number of productions may have fired before a reward is delivered. The reward R_i(n) that production *i* will receive will be the external reward received minus the time from production *i*'s selection to the reward. This serves to give less reward to more distant productions. This is like the temporal discounting in reinforcement learning but proves to be more robust within the ACT-R architecture (not suggesting it is generally more robust). This reinforcement goes back to all of the productions which have been selected between the current reward and the previous reward.

There are two ways to provide rewards to a model: at any time the trigger-reward command can be used to provide a reward or rewards can be attached to productions and those rewards will be applied after the corresponding production fires. Attaching rewards to productions can be the more convenient way to provide rewards to a model when they correspond to situations which the model will explicitly process. For instance, in the building sticks task the rewards are provided when the model successfully completes a problem and when it has to reset and start over, and there are productions which handle those situations: read-done detects that it has completed the problem and pick-another-strategy is responsible for choosing again after resetting. One can associate rewards with these outcomes by setting the reward values of those productions:

```
(spp read-done :reward 20)
(spp pick-another-strategy :reward 0)
```

When read-done fires it will propagate a reward of 20 back to the previous productions which have been fired. Of course, productions earlier in the chain will receive smaller values because the time to the reward is subtracted from the reward. If pick-another-strategy fires, a reward of 0 will be propagated back – which means that previous productions will actually receive a negative reward because of the time that passed. Consider what happens when a sequence of productions leads to a dead end, pick-another-strategy fires, another sequence of productions fire that leads to a solution, and then read-done fires. The reward associated with read-done will propagate back only to the production which fired after pick-another-strategy and no further because the reward only goes back as far as the last reward. Note that the production read-done will receive its own reward, but pick-another-strategy will not receive any of read-done's reward since it will have received the reward from its own firing.

6.4 Learning in the Building Sticks Task

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

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 when presented with the same problem at the end of the experiment 53% use overshoot.

The model which ran the test trials above can also perform the whole experiment, and will show performance similar to the data through the utility learning mechanism which is enabled in the model by setting the :ul parameter to t. The bst-experiment function in Lisp and the experiment function in the bst module for Python will run the model through the experiment multiple times averaging the results. The following shows the performance of the model averaged over 100 iterations of the experiment:

```
CORRELATION:  0.803
MEAN DEVIATION: 17.129

Trial 1    2    3    4    5    6    7    8    9    10   11   12   13   14   15
     23.0 60.0 59.0 70.0 91.0 42.0 80.0 86.0 59.0 34.0 33.0 22.0 54.0 72.0 56.0

DECIDE-OVER : 13.1506
DECIDE-UNDER: 11.1510
FORCE-OVER  : 12.1525
FORCE-UNDER : 6.5943
```

Also printed out are the average values of the utility parameters for the critical productions after each run through the experiment. As can be seen, the two over productions have increased their utility while the under productions have had a drop off. On average, the **force-over** production has a slightly higher value than the **decide-under** production. It is this change in utility values that creates the increased tendency to choose the overshoot strategy.

This model also turns on the utility learning trace, the :ult parameter, which works similar to the activation trace shown in the previous unit. If you enable the trace in the model by setting the :v parameter to **t** then every time there is a reward given to the model the trace will show the utility changes for all of the productions affected by that reward. Here is an example from a run showing the positive reward for successfully completing a trial:

```
     5.062   UTILITY                PROPAGATE-REWARD 20
 Utility updates with Reward = 20.0   alpha = 0.2
  Updating utility of production START-TRIAL
   U(n-1) = 0.0   R(n) = 14.938 [20.0 - 5.062 seconds since selection]
   U(n) = 2.9876
  Updating utility of production FIND-NEXT-LINE
   U(n-1) = 0.0   R(n) = 14.988 [20.0 - 5.012 seconds since selection]
   U(n) = 2.9976
  Updating utility of production ATTEND-LINE
   U(n-1) = 0.0   R(n) = 15.038 [20.0 - 4.962 seconds since selection]
   U(n) = 3.0076
  Updating utility of production ENCODE-LINE-A
   U(n-1) = 0.0   R(n) = 15.173 [20.0 - 4.827 seconds since selection]
   U(n) = 3.0346
  Updating utility of production FIND-NEXT-LINE
   U(n-1) = 2.9976   R(n) = 15.223 [20.0 - 4.777 seconds since selection]
   U(n) = 5.44268
  Updating utility of production ATTEND-LINE
   U(n-1) = 3.0076   R(n) = 15.273 [20.0 - 4.727 seconds since selection]
   U(n) = 5.46068
  Updating utility of production ENCODE-LINE-B
   U(n-1) = 0.0   R(n) = 15.423 [20.0 - 4.577 seconds since selection]
   U(n) = 3.0846002
  Updating utility of production FIND-NEXT-LINE
   U(n-1) = 5.44268   R(n) = 15.473 [20.0 - 4.527 seconds since selection]
   U(n) = 7.448744
  Updating utility of production ATTEND-LINE
   U(n-1) = 5.46068   R(n) = 15.523 [20.0 - 4.477 seconds since selection]
   U(n) = 7.473144
  Updating utility of production ENCODE-LINE-C
   U(n-1) = 0.0   R(n) = 15.658 [20.0 - 4.342 seconds since selection]
   U(n) = 3.1316001
  Updating utility of production FIND-NEXT-LINE
   U(n-1) = 7.448744   R(n) = 15.708 [20.0 - 4.292 seconds since selection]
   U(n) = 9.100595
  Updating utility of production ATTEND-LINE
   U(n-1) = 7.473144   R(n) = 15.757999 [20.0 - 4.242 seconds since selection]
   U(n) = 9.1301155
  Updating utility of production ENCODE-LINE-GOAL
   U(n-1) = 0.0   R(n) = 15.893 [20.0 - 4.107 seconds since selection]
   U(n) = 3.1786
  Updating utility of production ENCODE-UNDER
   U(n-1) = 0.0   R(n) = 16.028 [20.0 - 3.972 seconds since selection]
   U(n) = 3.2056
  Updating utility of production ENCODE-OVER
   U(n-1) = 0.0   R(n) = 16.163 [20.0 - 3.837 seconds since selection]
   U(n) = 3.2326
  Updating utility of production FORCE-UNDER
   U(n-1) = 10.0   R(n) = 16.213 [20.0 - 3.787 seconds since selection]
   U(n) = 11.2425995
  Updating utility of production MOVE-MOUSE
   U(n-1) = 0.0   R(n) = 16.263 [20.0 - 3.737 seconds since selection]
   U(n) = 3.2526002
  Updating utility of production CLICK-MOUSE
   U(n-1) = 0.0   R(n) = 16.713 [20.0 - 3.287 seconds since selection]
```

```
 U(n) = 3.3425999
Updating utility of production LOOK-FOR-CURRENT
 U(n-1) = 0.0   R(n) = 17.063 [20.0 - 2.937 seconds since selection]
 U(n) = 3.4126
Updating utility of production ATTEND-LINE
 U(n-1) = 9.1301155   R(n) = 17.112999 [20.0 - 2.887 seconds since selection]
 U(n) = 10.726692
Updating utility of production ENCODE-LINE-CURRENT
 U(n-1) = 0.0   R(n) = 17.248 [20.0 - 2.752 seconds since selection]
 U(n) = 3.4496
Updating utility of production CALCULATE-DIFFERENCE
 U(n-1) = 0.0   R(n) = 17.383 [20.0 - 2.617 seconds since selection]
 U(n) = 3.4766
Updating utility of production CONSIDER-C
 U(n-1) = 0.0   R(n) = 17.433 [20.0 - 2.567 seconds since selection]
 U(n) = 3.4866002
Updating utility of production CHOOSE-C
 U(n-1) = 0.0   R(n) = 17.568 [20.0 - 2.432 seconds since selection]
 U(n) = 3.5136
Updating utility of production MOVE-MOUSE
 U(n-1) = 3.2526002   R(n) = 17.618 [20.0 - 2.382 seconds since selection]
 U(n) = 6.12568
Updating utility of production CLICK-MOUSE
 U(n-1) = 3.3425999   R(n) = 17.668 [20.0 - 2.332 seconds since selection]
 U(n) = 6.2076797
Updating utility of production LOOK-FOR-CURRENT
 U(n-1) = 3.4126   R(n) = 17.868 [20.0 - 2.132 seconds since selection]
 U(n) = 6.3036804
Updating utility of production ATTEND-LINE
 U(n-1) = 10.726692   R(n) = 17.918 [20.0 - 2.082 seconds since selection]
 U(n) = 12.164953
Updating utility of production ENCODE-LINE-CURRENT
 U(n-1) = 3.4496   R(n) = 18.053 [20.0 - 1.947 seconds since selection]
 U(n) = 6.37028
Updating utility of production CALCULATE-DIFFERENCE
 U(n-1) = 3.4766   R(n) = 18.188 [20.0 - 1.812 seconds since selection]
 U(n) = 6.41888
Updating utility of production CONSIDER-C
 U(n-1) = 3.4866002   R(n) = 18.238 [20.0 - 1.762 seconds since selection]
 U(n) = 6.43688
Updating utility of production CONSIDER-A
 U(n-1) = 0.0   R(n) = 18.373 [20.0 - 1.627 seconds since selection]
 U(n) = 3.6746
Updating utility of production CHOOSE-A
 U(n-1) = 0.0   R(n) = 18.508 [20.0 - 1.492 seconds since selection]
 U(n) = 3.7015998
Updating utility of production MOVE-MOUSE
 U(n-1) = 6.12568   R(n) = 18.558 [20.0 - 1.442 seconds since selection]
 U(n) = 8.612144
Updating utility of production CLICK-MOUSE
 U(n-1) = 6.2076797   R(n) = 19.045 [20.0 - 0.955 seconds since selection]
 U(n) = 8.775144
Updating utility of production LOOK-FOR-CURRENT
 U(n-1) = 6.3036804   R(n) = 19.395 [20.0 - 0.605 seconds since selection]
 U(n) = 8.921945
Updating utility of production ATTEND-LINE
 U(n-1) = 12.164953   R(n) = 19.445 [20.0 - 0.555 seconds since selection]
 U(n) = 13.620962
Updating utility of production ENCODE-LINE-CURRENT
 U(n-1) = 6.37028   R(n) = 19.58 [20.0 - 0.42 seconds since selection]
 U(n) = 9.012224
Updating utility of production CALCULATE-DIFFERENCE
```

```
   U(n-1) = 6.41888   R(n) = 19.715 [20.0 - 0.285 seconds since selection]
   U(n) = 9.078104
  Updating utility of production CHECK-FOR-DONE
   U(n-1) = 0.0   R(n) = 19.765 [20.0 - 0.235 seconds since selection]
   U(n) = 3.9529998
  Updating utility of production FIND-DONE
   U(n-1) = 0.0   R(n) = 19.815 [20.0 - 0.185 seconds since selection]
   U(n) = 3.963
  Updating utility of production READ-DONE
   U(n-1) = 0.0   R(n) = 19.95 [20.0 - 0.05 seconds since selection]
   U(n) = 3.9900002
```

6.5 Additional Chunk-type Capabilities

Before discussing the assignment task for this unit we will look at a few of the productions in this model which appear to be doing things differently than previous units.

6.5.1 Default chunk-type slot values

If we look at the actions of the productions encode-line-goal and click-mouse we see that they seem to be missing the cmd slot in the requests to the **visual** and **manual** buffers which we have seen previously, and instead are only declaring a chunk-type with an isa:

```
(p encode-line-goal
...
  ==>
...
   +visual>
      isa        move-attention
      screen-pos =c)


(p click-mouse
...
  ==>
...
   +manual>
      isa    click-mouse)
```

Up to this point it has been stated that the isa declarations are optional and not a part of a request or condition. If that is true then how are those requests doing the right thing since we've also said that **visual** and **manual** requests require the cmd slot be specified to indicate the action to perform?

The answer to that question is that there is an additional process which happens when declaring a chunk-type with isa both in creating chunks and in specifying the conditions and actions of a production. That additional process relies on the ability to indicate default values for a slot when creating a chunk-type. Up until now when we have created chunk-types we have only specified the set of slots which it can include, but in addition to that one can specify default initial values for specific slots. If a chunk-type indicates that a slot should have a value by default, then when that chunk-type is declared any slots with default values that are not specified in the chunk definition or production statement will automatically be included with their default values.

To specify a default value for a slot in a chunk-type one needs to specify a list of two items for the slot where the first item is the slot name and the second is the default value, instead of just a slot name. Here are some example chunk-types for reference:

```
(chunk-type a slot)
(chunk-type b (slot 1))
(chunk-type c slot (slot2 t))
```

The chunk-type b has a default value of 1 for the slot named slot and chunk-type c has a default value of t for the slot named slot2. If we create some chunks specifying those chunk-types, some of which include a value for the slots with default values and others which do not, we can see how the default values are filled in for the chunks that do not specify those slots:

```
? (define-chunks (c1 isa a)
                 (c2 isa b)
                 (c3 isa b slot 3)
                 (c4 isa c)
                 (c5 isa c slot2 nil))


? (pprint-chunks C1 C2 C3 C4 C5)
C1

C2
   SLOT  1

C3
   SLOT  3

C4
   SLOT2  T

C5
```

For the slots with default values the chunks which specify the slot get the value specified, whereas the ones that do not get the default value. The same process applies to conditions and actions in a production. The chunk-types move-attention and click-mouse are created by the vision and motor modules and include default values for the cmd slot essentially like this (there is more to the real chunk-type specifications which will be discussed in the next section):

```
(chunk-type move-attention (cmd move-attention) screen-pos)
```

```
(chunk-type click-mouse (cmd click-mouse))
```

Therefore, this request:

```
   +visual>
      isa        move-attention
      screen-pos =c
```

is equivalent to this:

```
   +visual>
      cmd        move-attention
      screen-pos =c
```

and we can see that by looking at the actual representation of the production which the procedural module has for encode-line-goal using the Procedural viewer in the ACT-R Environment or by using the pp (print production) command:

```
? (pp encode-line-goal)
(P ENCODE-LINE-GOAL
   =GOAL>
       STATE ATTENDING
   =IMAGINAL>
       C-LOC =C
       GOAL-LOC NIL
   =VISUAL>
       SCREEN-POS =POS
       WIDTH =LENGTH
       LINE T
   ?VISUAL>
       STATE FREE
 ==>
   =IMAGINAL>
       GOAL-LOC =POS
       LENGTH =LENGTH
   =GOAL>
       STATE ENCODE-UNDER
   +VISUAL>
       SCREEN-POS =C
       CMD MOVE-ATTENTION
)
```

Notice how none of the isa declarations from the original definition are part of the actual production representation but the cmd slot has been included in the request to the **visual** buffer.

The same thing holds for the manual request in the click-mouse production:

```
(P CLICK-MOUSE
   =GOAL>
       STATE MOVE-MOUSE
   ?MANUAL>
       STATE FREE
 ==>
   =GOAL>
       STATE WAIT-FOR-CLICK
   +MANUAL>
       CMD CLICK-MOUSE
)
```

6.5.2 Chunk-type hierarchy

Now we will look at some of the isa declarations on the LHSs of the encode-line-goal and read-done productions.

```
(p encode-line-goal
   =goal>
      isa        try-strategy
      state      attending
   =imaginal>
      isa        encoding
      c-loc      =c
      goal-loc   nil
   =visual>
      isa        line
      screen-pos =pos
      width      =length
   ?visual>
      state      free
  ==>
   =imaginal>
      goal-loc   =pos
      length     =length
   =goal>
      state      encode-under
   +visual>
      isa        move-attention
      screen-pos =c)

(p read-done
   =goal>
```

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

Previously in the tutorial it was stated that the chunks placed into the **visual** buffer are created with slots from the chunk-type named visual-object. However, if we look at the conditions in those productions they are specifying chunk-types of line and text respectively in the **visual** buffer conditions.

In addition to the chunk-type visual-object the vision module defines some other chunk-types for the objects which are used by the AGI experiment windows. Those other chunk-types, like line and text, are actually subtypes of the type visual-object. A subtype contains all of the slots that its parent chunk-type contains, but may also contain additional slots or different default slot values for the slots which it has. When the vision module encodes a feature from an AGI window into a chunk it actually uses the slots of one of the specific types, like line or text to create the chunk.

In general one can create an arbitrary hierarchy of chunk-types with each subtype inheriting the slots and default values of its parent chunk-type (or even multiple parent chunk-types). A hierarchy of chunk-types can be helpful to the modeler in specifying chunks and productions, but other than through the inclusion of default slot values, such a hierarchy has no effect on the actual chunks or productions in the model.

To create a chunk-type which is a subtype of another chunk-type that parent type must be specified in the definition of the subtype. That is done by using a list to specify the name of the subtype which includes a list which has the keyword :include and the name of a parent chunk-type for each parent type to be included. These chunk-type definitions create chunk-types a and b, and then a chunk-type named c which is a subtype of chunk-type a and a chunk-type d which is a subtype of both a and b:

```
(chunk-type a slot1)
(chunk-type b slot2)
(chunk-type (c (:include a)) slot3)
(chunk-type (d (:include a) (:include b)) slot4)
```

One might think these definitions would be equivalent to those shown above:

```
(chunk-type a slot1)
(chunk-type b slot2)
(chunk-type c slot1 slot3)
```

```
(chunk-type d slot1 slot2 slot4)
```

However they are not because the subtyping mechanism is also extending the slots which are valid for the parent types as well. In addition to a subtype inheriting the slots from its parent types, the parent types also gain access to all of the slots from their children. Thus, with the definitions using the hierarchy it is acceptable to specify a parent type and use slots defined in its subtypes, but that is not valid for chunks which just happen to have the same slot names. Thus, with the first set of chunk-type definitions this is acceptable:

```
(define-chunks (isa a slot3 t slot4 10))
```

because chunk-types c and d are subtypes of chunk-type a and have slots named slot3 and slot4, but with the second set that would result in warnings for invalid slots in the chunk definition.

Now, for the line and text chunk-types used in these productions here are the relevant chunk-type definitions which the vision module has:

```
(chunk-type visual-object screen-pos value status color height width)
(chunk-type (text (:include visual-object)) (text t))
(chunk-type (line (:include visual-object)) (line t) end1-x end1-y
            end2-x end2-y)
```

The text and line chunk-types are subtypes of the visual-object type and each includes a new slot with the same name as the type and a default value of t. Because of that default slot value, declaring the buffer tests with the types text and line in these productions adds that additional condition to the productions as we can see when we look at the actual representation of them:

```
(P ENCODE-LINE-GOAL
   =GOAL>
       STATE ATTENDING
   =IMAGINAL>
       C-LOC =C
       GOAL-LOC NIL
   =VISUAL>
       SCREEN-POS =POS
       WIDTH =LENGTH
       LINE T
   ?VISUAL>
       STATE FREE
==>
   =IMAGINAL>
       GOAL-LOC =POS
       LENGTH =LENGTH
```

```
   =GOAL>
       STATE ENCODE-UNDER
   +VISUAL>
       SCREEN-POS =C
       CMD MOVE-ATTENTION
)
(P READ-DONE
   =GOAL>
       STATE READ-DONE
   =VISUAL>
       VALUE "done"
       TEXT T
 ==>
   +GOAL>
       STATE START
)
```

Those slot tests could have been specified directly, but declaring the type can be more readable and is consistent with the way chunk-types were used in older versions of ACT-R (those prior to version 6.1).

In the default slot section above it showed chunk-type definitions for the move-attention and click-mouse actions to indicate they had default slots, but the actual definitions of those chunk-types also include parent types which could be used in the requests as well:

```
(chunk-type vision-command cmd)
(chunk-type (move-attention (:include vision-command))
            (cmd move-attention) screen-pos scale)

(chunk-type motor-command (cmd "motor action"))
(chunk-type (click-mouse (:include motor-command)) (cmd click-mouse))
```

6.6 Learning in a Probability Choice Experiment

Your assignment is to develop a model for a "probability matching" experiment run by Friedman et al (1964). However, unlike the assignments for previous units, you are not provided with the code that implements the experiment this time. Therefore you will need to first write the experiment, and then develop the model, which more closely represents the typical modeling situation. The experiment to be implemented is very simple. Here is the basic procedure which is repeated for 48 trials:

1. The participant is presented with a screen saying "Choose"

2. The participant either presses the ‘h’ key for heads or the ‘t’ key for tails

3. When the key is pressed the screen is cleared and the feedback indicating the correct answer, either "Heads" or "Tails", is displayed.

4. That feedback stays on the screen for exactly 1 second before the next trial is presented.

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 long 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 blocks of 12 trials. These proportions are 0.664, 0.778, 0.804, and 0.818. This is the data that your model is to fit. It is important to note that this is the **proportion of choice for heads**, not the proportion of correct responses – the correctness of the response does not matter.

Your model must begin with a 50% chance of saying heads, then based on the feedback from the experiment it must adjust its choice through utility learning so that it averages responding heads close to 66% over the first block of 12 trials, and increases to about 82% 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:

```
CORRELATION:  0.994
MEAN DEVIATION:  0.016
 Original     Current
   0.664       0.646
   0.778       0.771
   0.804       0.826
   0.818       0.830
```

In achieving this, the parameters I worked with were the noise in the utilities (set by the :egs parameter) and the rewards associated with successful and unsuccessful responses.

The starting model for this task, found in the choice-model.lisp file of the unit, only contains the calls to clear ACT-R to its initial state and create a model named choice which has no content. You will need to write the whole model.

The initial code for this task can be found in the choice.lisp and choice.py files. It contains the call to load the starting model, specifies the experimental data, a global variable for collecting a response, and includes two functions. The first function is used as a monitor for the output-key action and just sets a global variable to record the response (as was done in many of the previous tasks). The other function, called choice-person in the Lisp version and person in the Python version, is a correct implementation of the task described for running a person through a single trial, and it returns the key which was pressed.

You should write a similar function to run the model through one trial, which should be named **choice-model** (in Lisp) or **model** (in Python). You will also need to write a function that takes one parameter and runs the whole 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. That function should be named **choice-data** (in Lisp) and **data** (in Python). That function does not have to be able to run a person through the task. It only needs to be able to run the model.

My suggestion would be to first write the single trial function making sure that it correctly represents the experiment described, including the timing. Then write a model that is able to perform that task correctly. Next write a function to run a block of 12 trials and test that to make sure the model works correctly when going from trial to trial. 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 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 you will need to use some of the ACT-R commands that were discussed in the previous units' code texts in addition to those already used in the function for running a person. The necessary commands will be described again here briefly, and the experiments you have seen up to this point 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 ACT-R Environment.

The **run** function can be used to run the model until either it has nothing to do or the specified amount of time has passed. It has one required parameter, which is the maximum amount of time to run the model in seconds.

The **run-full-time/run_full_time** function can be used to run the model for a specific amount of time. It takes one parameter which is the amount of time to run the model in seconds.

The **install-device/install_device** function takes one parameter which specifies a device the model will interact with and the value returned from opening the experiment window is the device of interest for this task.

In addition to those functions you will also want to use the **correlation** and **mean-deviation/mean_deviation** functions. Those calculate the correlation and mean-deviation between two lists of numbers.

6.6.1 Example experiment functions

The paired associate task from unit 4 is a good example to look at for creating your experiment, and the do-experiment function in the Lisp version and the do_experiment function in the Python version have a very similar structure to what you will need for presenting a trial of this choice experiment. The paired associate functions are a little more complicated than the function you will need for this assignment because they can run either a person or the model and are also recording response times and averaging the data over multiple runs which your single trial function will not need to do. They also call **reset** which you should not do in your single trial 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. Ignoring those complications, it performs a similar sequence of operations to those necessary to do this experiment: open a window, tell the model to interact with that window, present an item of text, run the model, clear the screen, display another item of text, and then run the model again. Those functions are copied here and the relevant operations are highlighted in green.

Lisp

```
(defun do-experiment (size trials human)

 (if (and human (not (visible-virtuals-available?)))
     (print-warning "Cannot run the task as a person without a visible window available.")
   (progn
     (reset)

     (let* ((result nil)
            (model (not human))
            (window (open-exp-window "Paired-Associate Experiment" :visible human)))

       (when model
         (install-device window))

       (dotimes (i trials)
         (let ((score 0.0)
               (time 0.0))

           (dolist (x (permute-list (subseq *pairs* (- 20 size))))

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

             (setf *response* nil)
             (let ((start (get-time model)))

               (if model
                   (run-full-time 5)
                 (while (< (- (get-time nil) start) 5000)
                   (process-events)))

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

               (clear-exp-window window)
               (add-text-to-exp-window window (second x) :x 150 :y 150)
               (setf start (get-time model))

               (if model
```

```
              (run-full-time 5)
            (while (< (- (get-time nil) start) 5000)
              (process-events)))))

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

  (reverse result)))))
```

Python

```
def do_experiment(size, trials, human):

    if human and not(actr.visible_virtuals_available()):
        actr.print_warning("Cannot run the task as a person without a visible window.")
    else:

        actr.reset()

        result = []
        model = not(human)
        window = actr.open_exp_window("Paired-Associate Experiment",visible=human)

        if model:
            actr.install_device(window)

        for i in range(trials):
            score = 0
            time = 0

            for prompt,associate in actr.permute_list(pairs[20 - size:]):

                actr.clear_exp_window(window)
                actr.add_text_to_exp_window (window, prompt, x=150 , y=150)

                global response
                response = ''
                start = actr.get_time(model)

                if model:
                    actr.run_full_time(5)
                else:
                    while (actr.get_time(False) - start) < 5000:
                        actr.process_events()

                if response == associate:
                    score += 1
                    time += response_time - start

                actr.clear_exp_window(window)
                actr.add_text_to_exp_window (window, associate, x=150 , y=150)
                start = actr.get_time(model)

                if model:
                    actr.run_full_time(5)
                else:
                    while (actr.get_time(False) - start) < 5000:
                        actr.process_events()

            if score > 0:
                average_time = time / score / 1000.0
            else:
```

```
                average_time = 0

            result.append((score/size,average_time))

        return result
```

In the choice files provided, the function for running a person provides the general structure for performing a trial in this task: it opens a window, creates a monitor for recording the key press, adds the choose prompt to the screen, clears that prompt, and then displays the feedback. However, instead of running a model it waits for a person to press the key and waits for 1 second of real time to pass.

Lisp

```
(defun choice-person ()
  (when (visible-virtuals-available?)
    (let ((window (open-exp-window "Choice Experiment" :visible t)))

      (add-act-r-command "choice-response" 'respond-to-key-press "Choice task key response")
      (monitor-act-r-command "output-key" "choice-response")

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

      (setf *response* nil)

      (while (null *response*)
        (process-events))

      (clear-exp-window window)

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

      (let ((start (get-time nil)))

        (while (< (- (get-time nil) start) 1000)
          (process-events)))

      (remove-act-r-command-monitor "output-key" "choice-response")
      (remove-act-r-command "choice-response")

      *response*)))
```

Python

```
def person():
    global response

    if actr.visible_virtuals_available():
        window = actr.open_exp_window("Choice Experiment",visible=True)

        actr.add_command("choice-response",respond_to_key_press,"Choice task key response")
        actr.monitor_command("output-key","choice-response")

        actr.add_text_to_exp_window (window, 'choose', x=50, y=100)
```

```
response = ''

while response == '':
    actr.process_events()

actr.clear_exp_window(window)

if actr.random(1.0) < .9:
    answer = 'heads'
else:
    answer = 'tails'

actr.add_text_to_exp_window (window, answer, x=50, y=100)

start = actr.get_time(False)

while (actr.get_time(False) - start) < 1000:
    actr.process_events()

actr.remove_command_monitor("output-key","choice-response")
actr.remove_command("choice-response")

return response
```

What you must do is write the corresponding function that has the appropriate interaction for the ACT-R model to perform the task. The code with a line through it is only a safety test for running a person and should **not** be included in your model running version. The code colored red above handles the interaction for a person doing the task, and that is **not** the same as what will be needed to run a model. It should be **replaced** with the appropriate code for the model to interact with the task, which will be similar to the green code from the paired example above (however the timing for the paired task is not the same as it is in this task so you will have to adjust that appropriately).

Something else to think about is that the exact placement of the choose prompt and the feedback of heads and tails is not specified in the description of the task. Therefore, your model should not assume anything about their locations i.e. your model should still be able to do the task regardless of where on the screen the choose prompt and the feedback occur. In the given function for running a person the answer is presented in the same location as the word "choose", but your model should also be able to perform the task if they are presented in different locations.

One final thing to note is that the paired associate task is an example of the "trial at a time" approach to building an experiment as discussed in the unit 4 code documentation, and that is probably the easiest way to approach this task. However, it is also possible to write this experiment using the "event-driven" style which was also discussed in the unit 4 code documentation. If you want to use that approach it will require a little more work to program because it does not analogize as neatly to one of the previous units' tasks. If you would like to try to write the experiment in that way you should look at the zbrodoff experiment as an example instead of the paired associate experiment. In fact, the different ways to write the experiment can actually have an effect on the data fitting for this model because they will likely have slightly different timing on the events which will affect the rewards received 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 are supposed to transition when the key press occurs, an event-driven experiment will provide a more veridical timing sequence because the events of the experiment will not be affected by components of the model other than its response. However, since you will be fitting the parameters in your model to the task you have written, that difference should not really matter, and either approach is acceptable for the assignment.

References

Bush, R. R., & Mosteller, F. (1955). *Stochastic Models for Learning*. New York: Wiley.

Friedman, M. P., Burke, C. J., Cole, M., Keller, L., Millward, R. B., & Estes, W. K., (1964). Two-choice behavior under extended training with shifting probabilities of reinforcement. In R. C. Atkinson (Ed.), *Studies in mathematical psychology* (pp. 250-316). Stanford, CA: Stanford University Press.

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.

Rescorla, R. A., & Wagner, A. R. (1972). A theory of Pavlovian conditioning: Variations on the effectiveness of reinforcement and nonreinforcement. In A. H. Black & W. R. Prokasy (eds.), *Classical Conditioning: II. Current Research and Theory* (pp. 64-99). New York: Appleton-Century-Crofts.