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. In this unit we will describe how those utilities can be learned from experience. 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
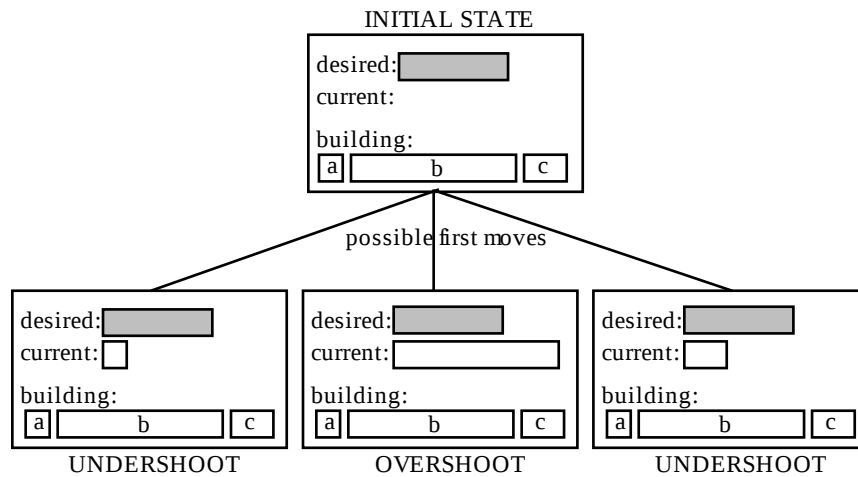
$$\Pr obability(i) = \frac{e^{U_i/\sqrt{2}s}}{\sum_j e^{U_j/\sqrt{2}s}}$$

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.
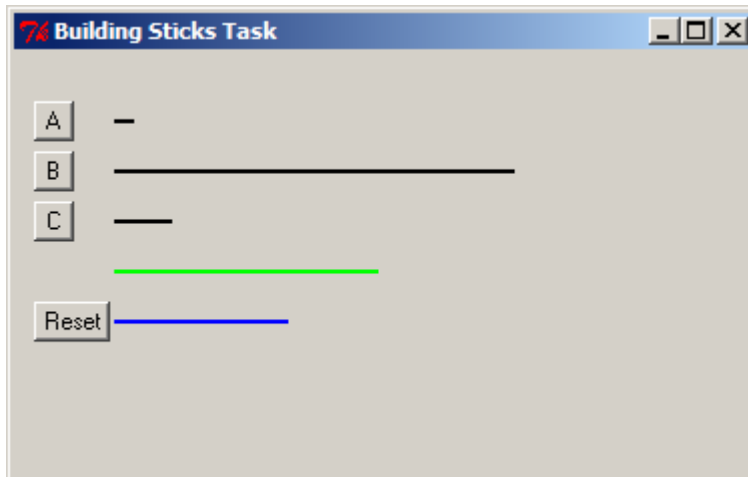
6.2 Building Sticks Example

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). 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.

INITIAL STATE

desired:

current:

building:

a   b   c

possible first moves

desired:	desired:	desired:
current:	current:	current:
building:	building:	building:
a b c	a b c	a b c
UNDERSHOOT	OVERSHOOT	UNDERSHOOT

You can go through a version of this by loading the model **bst**. By calling the command (bst-test 1 'human) you will run through a pair of problems in a version of the task built to run with a model (this is not the original experiment) and it will return a list of two items indicating which strategy, overshoot or undershoot, that you chose first.

The experiment will look something like this:

Building Sticks Task

A

B

C

Reset

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 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 undershoot for the first and overshoot for the second but not always. You can run the model multiple times by calling the function bst-test with one argument which is the number of runs through the two trials. If you only specify one pair of trials then the model will perform the task using a visible window so that you can watch it, but if there is more than one pair of trials specified then the model will use a virtual window. The following is the outcome of 100 trials:

```
> (bst-test 100)
(25 73)
```

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 many productions for encoding the screen and selecting sticks. However, the behavior of the model is really 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
      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
   =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
      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 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 environment, or by using the spp command:

```
> (spp force-over force-under decide-over decide-under)
Parameters for production FORCE-OVER:
 :utility 13.182
 :u  10.000
 :at  0.050
Parameters for production FORCE-UNDER:
 :utility 10.511
 :u  10.000
 :at  0.050
Parameters for production DECIDE-OVER:
 :utility 16.094
 :u  13.000
 :at  0.050
Parameters for production DECIDE-UNDER:
 :utility 19.609
 :u  13.000
 :at  0.050
```

The utility values, u, were set by the following spp commands in the model:

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

The **:u** parameters for the force productions are set to 10 while they are set to a more optimistic 13 for the decide productions. The **:utility** parameter shows the last computed utility value for the production during a conflict-resolution event and includes the utility

5

noise. Thus, we see that even though the true utility for decide-over is 13 it had a utility of 16.094 the last time it was matched in conflict-resolution. Unless a production is explicitly assigned a value for u it is given a default of 0. Therefore, the above four productions are the only ones in the model with non-zero utilities.

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 according to the equation.

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

In the model, the parameter *s* is set at 3. 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, they 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

$$U_i(n) = U_i(n-1) + \alpha[R_i(n) - U_i(n-1)]$$ **Difference Learning Equation**

where α is the learning rate and is typically set at .2 (this can be changed by adjusting the :alpha parameter with the sgp command). 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. 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 the productions which have been selected between the current reward and the previous reward.

It is possible to provide rewards to a model at any time by calling the trigger-reward command, and it is also possible to attach rewards to specific productions which are applied when those productions fire.   Attaching rewards to productions is often a convenient way to provide rewards to a model. For instance, in the building sticks task there is one production that fires when an action has been successful and another which fires when it has not:

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

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

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 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 after the sequence of problems this rises to 53% for the last problem.

The same bst model used to run the tests above can also perform this experiment, and will show performance similar to the data if the utility learning mechanism is enabled by setting the :ul parameter to t with the **sgp** command. The bst-experiment function will enable utility learning for the model and run it through the experiment multiple times averaging the results. The following is the performance of the model on a 100 simulation run:

```
> (bst-experiment 100)
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 over these 100 runs. 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 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:

```
     5.163    PROCEDURAL                 PRODUCTION-FIRED READ-DONE
     5.163    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) = 14.837 [20.0 - 5.163 seconds since selection]
   U(n) = 2.9674
  Updating utility of production FIND-NEXT-LINE
   U(n-1) = 0.0   R(n) = 14.887 [20.0 - 5.113 seconds since selection]
   U(n) = 2.9774
  Updating utility of production ATTEND-LINE
   U(n-1) = 0.0   R(n) = 14.937 [20.0 - 5.063 seconds since selection]
   U(n) = 2.9874
  Updating utility of production ENCODE-LINE-A
   U(n-1) = 0.0   R(n) = 15.0720005 [20.0 - 4.928 seconds since selection]
   U(n) = 3.0144002
  Updating utility of production FIND-NEXT-LINE
   U(n-1) = 2.9774   R(n) = 15.122 [20.0 - 4.878 seconds since selection]
   U(n) = 5.40632
```

```
Updating utility of production ATTEND-LINE
 U(n-1) = 2.9874   R(n) = 15.172 [20.0 - 4.828 seconds since selection]
 U(n) = 5.42432
Updating utility of production ENCODE-LINE-B
 U(n-1) = 0.0   R(n) = 15.3220005 [20.0 - 4.678 seconds since selection]
 U(n) = 3.0644002
Updating utility of production FIND-NEXT-LINE
 U(n-1) = 5.40632   R(n) = 15.372 [20.0 - 4.628 seconds since selection]
 U(n) = 7.399456
Updating utility of production ATTEND-LINE
 U(n-1) = 5.42432   R(n) = 15.422 [20.0 - 4.578 seconds since selection]
 U(n) = 7.4238563
Updating utility of production ENCODE-LINE-C
 U(n-1) = 0.0   R(n) = 15.557 [20.0 - 4.443 seconds since selection]
 U(n) = 3.1114001
Updating utility of production FIND-NEXT-LINE
 U(n-1) = 7.399456   R(n) = 15.607 [20.0 - 4.393 seconds since selection]
 U(n) = 9.040965
Updating utility of production ATTEND-LINE
 U(n-1) = 7.4238563   R(n) = 15.657 [20.0 - 4.343 seconds since selection]
 U(n) = 9.070485
Updating utility of production ENCODE-LINE-GOAL
 U(n-1) = 0.0   R(n) = 15.792 [20.0 - 4.208 seconds since selection]
 U(n) = 3.1584
Updating utility of production ENCODE-UNDER
 U(n-1) = 0.0   R(n) = 15.927 [20.0 - 4.073 seconds since selection]
 U(n) = 3.1854
Updating utility of production ENCODE-OVER
 U(n-1) = 0.0   R(n) = 16.062 [20.0 - 3.938 seconds since selection]
 U(n) = 3.2124002
Updating utility of production DECIDE-UNDER
 U(n-1) = 13.0   R(n) = 16.112 [20.0 - 3.888 seconds since selection]
 U(n) = 13.6224
Updating utility of production MOVE-MOUSE
 U(n-1) = 0.0   R(n) = 16.162 [20.0 - 3.838 seconds since selection]
 U(n) = 3.2324002
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) = 9.070485   R(n) = 17.112999 [20.0 - 2.887 seconds since selection]
 U(n) = 10.6789875
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) = 3.2324002   R(n) = 17.618 [20.0 - 2.382 seconds since selection]
 U(n) = 6.10952
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) = 10.6789875   R(n) = 17.918 [20.0 - 2.082 seconds since selection]
 U(n) = 12.12679
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) = 6.10952   R(n) = 18.558 [20.0 - 1.442 seconds since selection]
 U(n) = 8.599216
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) = 12.12679   R(n) = 19.445 [20.0 - 0.555 seconds since selection]
 U(n) = 13.590432
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 declaring a chunk-type with an isa:

```
(p encode-line-goal
```

```
   ...
  ==>
   =imaginal>
      goal-loc    =pos
      length      =length
   =goal>
      state       encode-under
   +visual>
      isa         move-attention
      screen-pos =c)


(p click-mouse
   ...
  ==>
   =goal>
      state  wait-for-click
   +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, but if that is true how are those requests doing the right thing since we've 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 symbol for the slot name. Here are some example chunk-types some of which have default values for slots:

```
(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 don't, we can see how the default values are filled in for the chunks that don't 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
```

The same process applies to conditions and actions in a production. The chunk-types move-attention and click-mouse are created by the visual and motor modules and include default values for the cmd slot essentially like this (there's also an additional component 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 printing out the representation of the production which the procedural module has for the encode-line-goal production 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 the isa information is not part of the request to the visual module in the actual representation of the production but the cmd slot has been included in the request. This would also result in the same request, but is a little bit redundant:

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

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

```
> (pp click-mouse)
(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 the conditions in 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
  ==>
   ...)

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

Previously in the tutorial it was stated that the chunks placed into the **visual** buffer are created with slots from the chunk-type 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 several other chunk-types for the objects which it can create. 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 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. This example creates 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 the ones above:

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

However, those are different because while the resulting chunk-types in both cases do have the same sets of slots, the chunk-types in the second case are not actually subtypes of the other types, they just happen to have slots with the same names. The difference is that with the true chunk-type 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 a default value of t that has the same name as the type. 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 print out the internal representation of them:

```
> (pp encode-line-goal read-done)
(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
)
```

Thus, this specification from the encode-line-goal production:

```
   =visual>
     isa          line
     screen-pos =pos
     width        =length
```

results in these conditions for the chunk in the buffer:

```
   =visual>
     screen-pos =pos
     width        =length
     line         t
```

which could have been specified directly, but declaring the specific type can be more readable and is consistent with the way chunk-types have been used in ACT-R prior to version 6.1. One other possible way to specify that condition would be to declare the parent type visual-object and specify the line slot explicitly:

```
   =visual>
     isa          visual-object
     screen-pos =pos
     width        =length
     line         t
```

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). 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. When the key is pressed the screen is cleared and presents as feedback the correct answer, either "Heads" or "Tails".
4. That feedback stays there 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 each block of 12 trials. These proportions are 0.66, 0.78, 0.80, and 0.82. 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 based on the feedback from the experiment it must adjust its probabilities 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:

```
> (collect-data 100)
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 in the **sgp** command) and the rewards associated with successful and unsuccessful responses.

The starting model you are given for this task, **choice**, contains only the functions which are able to run a person through one trial and to collect a key press response using the "trial at a time" experiment writing style as discussed in the unit 4 experiment code document. The rpm-window-key-event-handler method provided is very similar to those from other units and will record a key press from either a person or the model by setting the variable ***response*** to the string representing that key. The function **do-choice-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-choice-model**. You also need to write a function called **choice-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. The **choice-data** function does not have to be able to run a person through the task. It only needs to be able to run the model. You also must write the model for the task that fits the data.

My suggestion would be to first write the **do-choice-model** function and a model that does the task (without trying to fit the data), and make sure that works correctly. An important issue here is to make sure that it correctly represents the experiment described, including the timing. 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 **choice-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 previous units' experiment description texts. Those functions will be described again here, and the models 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 environment.

The function **install-device** takes one parameter which should be a window. That parameter tells ACT-R 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 **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 **proc-display**. Thus, for the model to notice a change to the window **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 visual re-encoding if the model is attending an item.

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, the maximum amount of time to run the model.

The **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 addition to those functions you will also need to use the **correlation** and **mean-deviation** functions. Those 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 which should serve as a useful example of presenting a task:

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

    (reset)

    (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 (get-time))

          (proc-display)
          (run-full-time 5)

          (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)

          (proc-display)
          (run-full-time 5))

        (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-choice-model** function will not be doing. It also calls **reset** which you should not do in your **do-choice-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. Ignoring those complications, it performs a similar sequence of operations to those necessary to do this experiment: opening a window, presenting an item of text, running the model, clearing the screen, displaying another item of text and then running the model again (the code highlighted in red). Note however that the timing for the choice task is not the same as the timing of the paired associate experiment thus you will have to do things somewhat differently to accurately replicate the choice task described.

In the choice file provided the **do-choice-person** function provides the structure for the displaying of the items in the choice task:

```
(defun do-choice-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 (< (act-r-random 1.0) .9) "heads" "tails")
                            :x 50 :y 100)

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

What you must do is write the **do-choice-model** function that presents the display similar to the way that **do-choice-person** does, but has the appropriate interaction with ACT-R (the code colored green handles the interaction for a person doing the task and should **not** appear in your **do-choice-model** function). The exact placement of the choose prompt and the feedback of heads and tails is not important for the task, and the 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 choose and the feedback occur including situations where they are not both in the same location.

It is also possible to write the experiment using an event-based style as discussed in the unit 4 experiment code text. That 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 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 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. In this case, because the trials transition on the response of the model, an event-based 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, either solution is acceptable for the assignment.

References

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.