Getting started II

May 19, 2019

1 Getting started II

2 Model 3 - introduction to vision

In the previous two examples we introduced a simple mind that tried to play Memory but instead of that, it just keeps hitting 1 until noone wanted to play with that crazy player anymore.

One problem is that our mind can enforce its wishes on the environment, but it cannot take any information from the environment. But playing Memory requires both.

There is one way pyactr can take information from the environment, through its visual module.

To make it work, we have to initialize the model with an environment.

```
In [1]: import pyactr as actr
        environment = actr.Environment()
        playing_memory = actr.ACTRModel(environment=environment)
```

The environment is an artificial computer screen, our mind cannot see more than that. Let's put something on that screen, say, letter A. We do that by specifying visual stimuli, a list of dictionaries. Every dictionary in the list is one set of stimuli that will appear together on the screen. The dictionary has, as its keys, some mnemonic info by which we can easily identify the stimuli that appear together (we can use letter "A" for our one stimulus) and, as its values, information for the environment about the parameters of each stimulus. The parameters are written as another dictionary. At least two parameters have to be explicit: the x-y position of the stimulus (on a 640x360 screen) and the text that the stimulus has. This leads to the following stimuli (the first dictionary is empty because at the beginning nothing appears on the screen):

```
In [2]: memory = [{}, {"A": {'text': 'A', 'position': (100, 100)}}]
```

We will prepare the model in the same way as before, specifying the initial goal buffer with a chunk that states we play a memory. Notice that our starting chunk is very much like it was before, only two more slots are added to it. One slot, "key" will express what key the model is about to press. The second slot, "object", will express what object was found by that key press. We will use these extra slots in a second.

```
In [3]: goal = playing_memory.set_goal("goal")
        actr.chunktype("playgame", "game, activity, key, object")
        initial_chunk = actr.makechunk(typename="playgame", game="memory", key="1")
        goal.add(initial_chunk)
```

Now, onto the rules. The first two rules did not change significantly. There is only one simplifying change: since we specified already at the initial step what key will be pressed, we don't have to do it again in the "presskey" rule. Rather, we will make that action dependent on the information in the goal buffer. This is done by variable binding: "=k" in the goal buffer binds the value of the slot "key" to the variable "k"; this value is then re-used whenever "=k" appears again, that is, in the key slot of the manual buffer. As a consequence, the key slot in the manual buffer will carry the same value as the key slot in the goal buffer after this rule fires.

```
In [4]: playing_memory.productionstring(name="startplaying", string="""
        =goal>
        isa  playgame
        game memory
        activity None
        ==>
        =goal>
        isa playgame
        activity presskey""") #to be modified later

        playing_memory.productionstring(name="presskey", string="""
        =goal>
        isa  playgame
        game memory
        activity presskey
        key =k
        ==>
        +manual>
        isa _manual
        cmd press_key
        key =k
        =goal>
        isa playgame
        activity attend""")
```

```
Out[4]: {'=goal': playgame(activity= presskey, game= memory, key= =k, object= )}
        ==>
        {'=goal': playgame(activity= attend, game= , key= , object= ), '+manual': _manual(cmd=
```

Apart from this old stuff, we add a new rule. It expresses what happens after a key press is done and, hopefully, some information appeared on the screen. Basically, the rule will move the mind's attention to the object located on the screen (the object itself is first automatically found by the visual_location buffer).

There are two new bits in this rule. The first new bit is that we check on the visual_location buffer in the condition. If some visual_location is present (that is, the model knows that an object is located on the screen), attention is moved to the object. The second new bit is the last four lines of the rule. This sets a peripheral in action. The description of the action is not much different from pressing a key. The action concerns the visual buffer this time: what happens is that the visual attention is moved (command move_attention) to the position specified by the visual_location buffer.

```
In [5]: playing_memory.productionstring(name="attendobject", string="""
        =goal>
        isa  playgame
        game memory
        activity attend
        =visual_location>
        isa _visuallocation
        ?manual>
        state free
        ==>
        =goal>
        isa playgame
        activity storeobject
        +visual>
        isa _visual
        cmd     move_attention
        screen_pos =visual_location""")

Out[5]: {'=visual_location': _visuallocation(color= , screen_x= , screen_y= , value= ), '=goal
        ==>
        {'+visual': _visual(cmd= move_attention, color= , screen_pos= =visual_location, value=
```

Finally, we will store the visual information in the goal buffer. This is done again by variable binding: we specify that whatever is the value in the current visual attention is bound to "v" (by setting "value =v"); this v is then transferred as the value of the goal buffer. In the end, the goal buffer carries two pieces of crucial information: what key was pressed and what visual object appeared with that key press.

```
In [6]: playing_memory.productionstring(name="storeobject", string="""
        =goal>
        isa  playgame
        game memory
        activity storeobject
        =visual>
        isa _visual
        value =v
        ==>
        =goal>
        isa playgame
        activity None
        object =v""")

Out[6]: {'=goal': playgame(activity= storeobject, game= memory, key= , object= ), '=visual': _v
        ==>
        {'=goal': playgame(activity= None, game= , key= , object= =v)}
```

We can now run simulation with these four rules.

The simulation is started as in previous cases, with one small modification. We have to specify what environment process it should be tied to. The class Environment specifies one process, which

proceeds by printing one stimulus after another whenever a trigger is pressed, or time expires. We use it here. Following parameters (stimuli and triggers) are parameters supplied to this process.

Finally we also set gui to False to let the environment print its information directly in the trace. If we set it to True, the environment would appear in a separate window. Setting gui to True requires tkinter.

```
In [7]: simulation = playing_memory.simulation(gui=False, environment_process=environment.envi
        simulation.run()
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: startplaying')
****Environment: {}
(0.05, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.05, 'goal', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.1, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.1, 'goal', 'MODIFIED')
(0.1, 'manual', 'COMMAND: press_key')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'NO RULE FOUND')
(0.35, 'manual', 'PREPARATION COMPLETE')
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'NO RULE FOUND')
(0.4, 'manual', 'INITIATION COMPLETE')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'NO RULE FOUND')
(0.5, 'manual', 'KEY PRESSED: 1')
****Environment: {'A': {'text': 'A', 'position': (100, 100)}}
(0.5, 'visual_location', 'ENCODED LOCATION: _visuallocation(color= , screen_x= 100, screen_y= 1
(0.5, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.5, 'PROCEDURAL', 'NO RULE FOUND')
(0.65, 'manual', 'MOVEMENT FINISHED')
(0.65, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.65, 'PROCEDURAL', 'RULE SELECTED: attendobject')
(0.7, 'PROCEDURAL', 'RULE FIRED: attendobject')
(0.7, 'goal', 'MODIFIED')
(0.7, 'visual', 'PREPARATION TO SHIFT VISUAL ATTENTION STARTED')
(0.7, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.7, 'PROCEDURAL', 'NO RULE FOUND')
(0.7121, 'visual', 'CLEARED')
(0.7121, 'visual', "ENCODED VIS OBJECT:'_visual(cmd= move_attention, color= , screen_pos= _visu
(0.7121, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.7121, 'PROCEDURAL', 'RULE SELECTED: storeobject')
(0.7621, 'PROCEDURAL', 'RULE FIRED: storeobject')
(0.7621, 'goal', 'MODIFIED')
(0.7621, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.7621, 'PROCEDURAL', 'RULE SELECTED: startplaying')
```

```
(0.8121, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.8121, 'goal', 'MODIFIED')
(0.8121, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.8121, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.8621, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.8621, 'goal', 'MODIFIED')
(0.8621, 'manual', 'COMMAND: press_key')
(0.8621, 'manual', 'PREPARATION COMPLETE')
(0.8621, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.8621, 'PROCEDURAL', 'NO RULE FOUND')
(0.8701, 'visual', 'PREPARATION TO SHIFT VISUAL ATTENTION COMPLETED')
(0.8701, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.8701, 'PROCEDURAL', 'NO RULE FOUND')
(0.9121, 'manual', 'INITIATION COMPLETE')
(0.9121, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.9121, 'PROCEDURAL', 'NO RULE FOUND')
(0.9315, 'visual', 'SHIFT COMPLETE TO POSITION: [100, 100]')
(0.9315, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.9315, 'PROCEDURAL', 'NO RULE FOUND')
```

Inspecting the goal buffer reveals that we correctly updated the chunk with the information that key "1" corresponds to object "A".

```
In [8]: print(goal)
```

```
{playgame(activity= attend, game= memory, key= 1, object= A)}
```

This might look a bit more like Memory, but there is definitely at least one thing missing. The mind stores only one key-object pair. This would not get us very far in the game.

In a nutshell, the problem is that our model is memoryless. In fact, any past knowledge is irrelevant. A future action is decided just based on the current contents of the buffers. We will remove the limitation in the following model.

3 Model 4 - introduction to declarative memory

When the mind achieves a goal, the chunk in the goal buffer does not just disappear. It is flushed into the declarative memory. Similarly, when the visual buffer has achieved whatever it should have achieved, its chunk is flushed into the declarative memory. Any chunk can in principle be later recalled (retrieved) from that memory.

We will now implement this idea.

First, we will modify our "startplaying" rule by requiring that this rule only fires in the first round (before any object is recalled).

```
In [9]: playing_memory.productionstring(name="startplaying", string="""
        =goal>
        isa  playgame
```

```
        game memory
        activity None
        object None
        ==>
        =goal>
        isa playgame
        activity presskey""")
```

```
Out[9]: {'=goal': playgame(activity= None, game= memory, key= , object= None)}
        ==>
        {'=goal': playgame(activity= presskey, game= , key= , object= )}
```

In contrast to that rule, "continueplaying" will only fire when the goal buffer has some value in the slot "object". This is required by setting "object ~None" (~ negates a value that follows).

If this is so, we let the model proceed to recalling.

```
In [10]: playing_memory.productionstring(name="continueplaying", string="""
         =goal>
         isa  playgame
         game memory
         activity None
         object ~None
         ==>
         =goal>
         isa playgame
         game memory
         activity recall""")
```

```
Out[10]: {'=goal': playgame(activity= None, game= memory, key= , object= ~None)}
         ==>
         {'=goal': playgame(activity= recall, game= memory, key= , object= )}
```

The following three rules are the meat of this model. They work as follows:

1. The rule "recallvalue" tries to recall a chunk with particular values.
2. The rule "recallsuccessful" will terminate the simulation if some chunk was found. If this is so, the goal buffer will signal what key should be pressed next (we will assume that the game stops when one successful pair is found).
3. The rule "recallfailed" will set the model back into the game if nothing was recalled.

The rule "recallvalue" is specified below. Recalling is done by using another buffer, "retrieval". The buffer will search the declarative memory for any chunk that matches its requirements. In our case, the chunk must have, as its object, whatever value was in the slot "object" of the goal buffer. Now, recall that "object" carries the information about the visual element that was just inspected. Recalling a chunk that has the same value in the slot "object" is only possible if a pair was found.

```
In [11]: playing_memory.productionstring(name="recallvalue", string="""
         =goal>
         isa  playgame
```

```
game memory
activity recall
object =val
==>
=goal>
isa  playgame
game memory
activity checkrecalled
+retrieval>
isa playgame
object =val""")
```

```
Out[11]: {'=goal': playgame(activity= recall, game= memory, key= , object= =val)}
         ==>
         {'+retrieval': playgame(activity= , game= , key= , object= =val), '=goal': playgame(a
```

The rule "recallsuccessful" fires only if something was retrieved. This is done by querying whether the retrieval buffer is full. If this is so, we will signal that we are done ("activity done") but that the final key must be identical to the one that let us found the first member in the pair.

```
In [12]: playing_memory.productionstring(name="recallsuccessful", string="""
         =goal>
         isa  playgame
         game memory
         activity checkrecalled
         object =val
         ?retrieval>
         buffer full
         =retrieval>
         isa playgame
         key =k
         ==>
         =goal>
         isa playgame
         key =k
         activity done""")
```

```
Out[12]: {'?retrieval': {'buffer': 'full'}, '=retrieval': playgame(activity= , game= , key= =k
         ==>
         {'=goal': playgame(activity= done, game= , key= =k, object= )}
```

Finally, the last rule states that when the retrieval failed (querying on the retrieval and requiring "state" error"), we should move on:

1. In the action, the goal buffer is cleared (~goal>)
2. A new chunk is put in the goal buffer which requires that we are to press the next key in the line (key 2).

```
In [13]: playing_memory.productionstring(name="recallfailed", string="""
         =goal>
```

```
        isa  playgame
        game memory
        activity checkrecalled
        key 1
        object =val
        ?retrieval>
        state error
        ==>
        ~goal>
        +goal>
        isa playgame
        game memory
        activity presskey
        key 2""")
```

```
Out[13]: {'?retrieval': {'state': 'error'}, '=goal': playgame(activity= checkrecalled, game= me
         ==>
         {'~goal': None, '+goal': playgame(activity= presskey, game= memory, key= 2, object= )}
```

This is all. We now only have to make a new set of stimuli for the game of Memory and let the model run. There are three stimuli present, the empty screen at the start, object "A" (which will appear when the model presses 1) and object "A" as the last one (this will appear when the model presses 2).

The simulation is run for 2 seconds.

```
In [14]: memory = [{}, {"A": {'text': 'A', 'position': (100, 100)}}, {"A": {'text': 'A', 'posit
```

```
In [15]: goal.add(initial_chunk)
         simulation = playing_memory.simulation(gui=False, environment_process=environment.envi
         simulation.run(max_time=2)
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: startplaying')
****Environment: {}
(0.05, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.05, 'goal', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.1, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.1, 'goal', 'MODIFIED')
(0.1, 'manual', 'COMMAND: press_key')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'NO RULE FOUND')
(0.35, 'manual', 'PREPARATION COMPLETE')
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'NO RULE FOUND')
(0.4, 'manual', 'INITIATION COMPLETE')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'NO RULE FOUND')
```

```
(0.5, 'manual', 'KEY PRESSED: 1')
****Environment: {'A': {'text': 'A', 'position': (100, 100)}}
(0.5, 'visual_location', 'ENCODED LOCATION: _visuallocation(color= , screen_x= 100, screen_y= 1
(0.5, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.5, 'PROCEDURAL', 'NO RULE FOUND')
(0.5096, 'visual', 'AUTOMATIC BUFFERING: _visual(cmd= , color= , screen_pos= _visuallocation(co
(0.5096, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.5096, 'PROCEDURAL', 'NO RULE FOUND')
(0.65, 'manual', 'MOVEMENT FINISHED')
(0.65, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.65, 'PROCEDURAL', 'RULE SELECTED: attendobject')
(0.7, 'PROCEDURAL', 'RULE FIRED: attendobject')
(0.7, 'goal', 'MODIFIED')
(0.7, 'visual', 'PREPARATION TO SHIFT VISUAL ATTENTION STARTED')
(0.7, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.7, 'PROCEDURAL', 'NO RULE FOUND')
(0.7086, 'visual', 'CLEARED')
(0.7086, 'visual', "ENCODED VIS OBJECT:'_visual(cmd= move_attention, color= , screen_pos= _visu
(0.7086, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.7086, 'PROCEDURAL', 'RULE SELECTED: storeobject')
(0.7586, 'PROCEDURAL', 'RULE FIRED: storeobject')
(0.7586, 'goal', 'MODIFIED')
(0.7586, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.7586, 'PROCEDURAL', 'RULE SELECTED: continueplaying')
(0.8086, 'PROCEDURAL', 'RULE FIRED: continueplaying')
(0.8086, 'goal', 'MODIFIED')
(0.8086, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.8086, 'PROCEDURAL', 'RULE SELECTED: recallvalue')
(0.8586, 'PROCEDURAL', 'RULE FIRED: recallvalue')
(0.8586, 'goal', 'MODIFIED')
(0.8586, 'retrieval', 'START RETRIEVAL')
(0.8586, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.8586, 'PROCEDURAL', 'NO RULE FOUND')
(0.8682, 'visual', 'PREPARATION TO SHIFT VISUAL ATTENTION COMPLETED')
(0.8682, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.8682, 'PROCEDURAL', 'NO RULE FOUND')
(0.9086, 'retrieval', 'RETRIEVED: None')
(0.9086, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.9086, 'PROCEDURAL', 'RULE SELECTED: recallfailed')
(0.9586, 'PROCEDURAL', 'RULE FIRED: recallfailed')
(0.9586, 'goal', 'CLEARED')
(0.9586, 'goal', 'CLEARED')
(0.9586, 'goal', 'CREATED A CHUNK: playgame(activity= presskey, game= memory, key= 2, object= )
(0.9586, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.9586, 'PROCEDURAL', 'RULE SELECTED: presskey')
(1.0028, 'visual', 'SHIFT COMPLETE TO POSITION: [100, 100]')
(1.0086, 'PROCEDURAL', 'RULE FIRED: presskey')
(1.0086, 'goal', 'MODIFIED')
```

```
(1.0086, 'manual', 'COMMAND: press_key')
(1.0086, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1.0086, 'PROCEDURAL', 'NO RULE FOUND')
(1.1586, 'manual', 'PREPARATION COMPLETE')
(1.1586, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1.1586, 'PROCEDURAL', 'NO RULE FOUND')
(1.2086, 'manual', 'INITIATION COMPLETE')
(1.2086, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1.2086, 'PROCEDURAL', 'NO RULE FOUND')
(1.3086, 'manual', 'KEY PRESSED: 2')
****Environment: {'A': {'text': 'A', 'position': (100, 100)}}
(1.3086, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1.3086, 'PROCEDURAL', 'NO RULE FOUND')
(1.3211, 'visual', 'AUTOMATIC BUFFERING: _visual(cmd= , color= , screen_pos= _visuallocation(c
(1.3211, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1.3211, 'PROCEDURAL', 'NO RULE FOUND')
(1.4586, 'manual', 'MOVEMENT FINISHED')
(1.4586, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1.4586, 'PROCEDURAL', 'RULE SELECTED: attendobject')
(1.5086, 'PROCEDURAL', 'RULE FIRED: attendobject')
(1.5086, 'goal', 'MODIFIED')
(1.5086, 'visual', 'PREPARATION TO SHIFT VISUAL ATTENTION STARTED')
(1.5086, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1.5086, 'PROCEDURAL', 'NO RULE FOUND')
(1.5165, 'visual', 'CLEARED')
(1.5165, 'visual', "ENCODED VIS OBJECT:'_visual(cmd= move_attention, color= , screen_pos= _visu
(1.5165, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1.5165, 'PROCEDURAL', 'RULE SELECTED: storeobject')
(1.5665, 'PROCEDURAL', 'RULE FIRED: storeobject')
(1.5665, 'goal', 'MODIFIED')
(1.5665, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1.5665, 'PROCEDURAL', 'RULE SELECTED: continueplaying')
(1.6155, 'visual', 'PREPARATION TO SHIFT VISUAL ATTENTION COMPLETED')
(1.6165, 'PROCEDURAL', 'RULE FIRED: continueplaying')
(1.6165, 'goal', 'MODIFIED')
(1.6165, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1.6165, 'PROCEDURAL', 'RULE SELECTED: recallvalue')
(1.6665, 'PROCEDURAL', 'RULE FIRED: recallvalue')
(1.6665, 'goal', 'MODIFIED')
(1.6665, 'retrieval', 'START RETRIEVAL')
(1.6665, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1.6665, 'PROCEDURAL', 'NO RULE FOUND')
(1.7165, 'retrieval', 'CLEARED')
(1.7165, 'retrieval', 'RETRIEVED: playgame(activity= checkrecalled, game= memory, key= 1, obje
(1.7165, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1.7165, 'PROCEDURAL', 'RULE SELECTED: recallsuccessful')
(1.724, 'visual', 'SHIFT COMPLETE TO POSITION: [100, 100]')
(1.7665, 'PROCEDURAL', 'RULE FIRED: recallsuccessful')
```

```
(1.7665, 'goal', 'MODIFIED')
(1.7665, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1.7665, 'PROCEDURAL', 'NO RULE FOUND')
```

Everything looks correct, but to be sure we can check that the goal buffer is done at the end and that it states that the other member of the pair "A"-"A" appeared when 1 was pressed.

```
In [16]: print(goal)

{playgame(activity= done, game= memory, key= 1, object= A)}
```

And as we are at it, we can also check what our declarative memory looks like. This will print all chunks that are in the memory, along with the time stamps at which they were introduced. Note that chunks from the visual module appears here, as well.

```
In [17]: print(playing_memory.decmem)

{_visual(cmd= move_attention, color= None, screen_pos= _visuallocation(color= , screen_x= 100,
```

This is ok. Now, we mentioned that our game of Memory should ideally consist of more cards, revealed by pressing 1,2,...,9.

To finalize this game, we have to do create 9 versions of "recallfailed", depending on the key that should be pressed.

We can make all the rules in one loop, as follows:

```
In [18]: for i in range(1, 9):
             playing_memory.productionstring(name="recallfailed" + str(i), string="""
             =goal>
             isa  playgame
             game memory
             activity checkrecalled
             key """+ str(i) +"""
             object =val
             ?retrieval>
             state error
             ==>
             ~goal>
             +goal>
             isa playgame
             game memory
             activity presskey
             key """ + str((i+1)%10))

         playing_memory.productionstring(name="recallfailed9", string="""
         =goal>
         isa  playgame
```

```
        game memory
        activity checkrecalled
        key 0
        object =val
        ?retrieval>
        state error
        ==>
        ~goal>""")
```

```
Out[18]: {'?retrieval': {'state': 'error'}, '=goal': playgame(activity= checkrecalled, game= me
         ==>
         {'~goal': None}
```

Notice that these rules will be called "recallfailed1"..."recallfailed9". They will increment the number that should be pressed whenever the retrieval failed.

Now, we only need to have a big enough list of stimuli and prepare the simulation.

```
In [19]: memory = [{}] + [{i: {'text': i, 'position': (100, 100)}} for i in "BCDEFGHDIJ"]
         goal.add(initial_chunk)
         playing_memory.retrieval.pop()
         simulation = playing_memory.simulation(environment_process=environment.environment_pro
```

We could run the simulation but it would take a lot of space. So instead of that, we will proceed step-wise. We will run a loop. At every turn of the loop, one step of the simulation is taken. (One step corresponds to one line in the trace of the model.) We can then inspect whether the retrieval was successful. We do so by checking whether the current action in the simulation is the rule "recallsuccessful". If it is, we break.

```
In [20]: while True:
             simulation.step()
             if simulation.current_event.action == "RULE FIRED: recallsuccessful":
                 break
```

The final piece is to inspect the resulting goal buffer. Notice that based on what we specified in our stimuli (memory), we expect that the model should find that there is one match, "D", and the first member of the pair appeared when the key "3" was pressed.

```
In [21]: print(goal)
```

```
{playgame(activity= done, game= memory, key= 3, object= D)}
```

As a final bit, let us check how much time the whole memory game took.

```
In [22]: print(simulation.show_time())
```

```
6.7147
```

4 Final model - introduction to sub-symbolic system

The models that we considered here are rather simple and boring from the perspective of AI, Machine Learning etc. So why do people use ACT-R?

One reason is that ACT-R strikes a nice balance between the level of abstractness and the level of precision. The models are abstract enough so that modelers can quite quickly draw a simulation of a task they are interested in. At the same time, the models are no hand-waving, quite a lot of details have to be specified. Due to this, the models make precise predictions re behavioral data, in particular, how much time a task is predicted to take, how often people fail in the task etc. The main reason to use ACT-R is to have a precise model at hand that can simulate such behavioral data in enough detail.

An ACT-R model is linked to quantitative measures through the sub-symbolic system. Let us switch it on and run the last simulation again.

We switch it on (along with other parameters) by changing a value in the dictionary model_parameters.

```
In [23]: playing_memory.model_parameters["subsymbolic"] = True
```

```
In [24]: goal.add(initial_chunk)
         playing_memory.retrieval.pop()
         for i in playing_memory.decmem.copy():
             playing_memory.decmem.pop(i)
         simulation = playing_memory.simulation(environment_process=environment.environment_pr
```

```
In [25]: while True:
             simulation.step()
             if simulation.current_event.action == "RULE FIRED: recallsuccessful":
                 break
             if simulation.show_time() > 10: #10 seconds is enough time to go through the card
                 print("Nothing found, breaking")
                 break
```

```
Nothing found, breaking
```

What happened here? Why was nothing found?

Here is one way to put it. The goal of the last model (and any model in ACT-R) is not to fulfill some task, find the right answer etc. The goal is to fulfill some task *in the same way as humans would do*. Now, it is possible that people would have hard time to remember the first D at the time they see the second D on the screen, and if they do, so should the model. Whether they do or not is an empirical question. But whatever the findings, the model should mimic the behavior of humans - say, by taking as much time to go through the task as people do, by getting the answer right as often as people do, among other things.

5 Where to go from here?

A theoretical introduction to ACT-R can be found in the paper Anderson et al., 2004, An Integrated Theory of the Mind, Psychological Review.

An even better, but much longer introduction is in the book Anderson & Lebiere, The Atomic Components of Thought.

A hands-on introduction is present in the tutorials written for LISP and available here: http://act-r.psy.cmu.edu/software/ Almost all the code from the tutorials was translated into pyactr. Check the folder tutorials here on github for it.

Some more information about ACT-R and pyactr is present in the folder docs/ on github.