5_model_for_counting_in_ACTR

April 2, 2021

1 A model for counting in ACTR/`pyactr`

[1]:
```
# uncomment the line below to install pyactr
# !pip3 install pyactr

import pyactr as actr
```

In this section, we present three more (simple) ACT-R models. The models do not add any new concepts to what we have learned so far about ACT-R and `pyactr`. Before we delve into the models, we should point out that none of these models is necessarily cognitively realistic or plausible. We simply present them here to solidify the reader's knowledge of the concepts introduced in this chapter. They also serve as preparation for the more complex linguistic performance models we develop in the remainder of the book.

The first model shows how counting can be simulated in ACT-R. This is a classical, toy example that modelers are often first introduced to when learning about ACT-R.[1] It is a subcomponent of a larger model. The larger model does strive to simulate actual human cognition: it captures how young children learn addition (see **Lebiere, Christian. 1999. The dynamics of cognition: An ACT-R model of cognitive arithmetic.** ***Kognitionswissenschaft*** **8:5–19**). However, our simple model does not have this ambitious goal. The second and third models show how regular grammars and counter automata can be implemented in ACT-R.

The model starts with some number and keeps incrementing it by one until it reaches another, final number. We have two chunk types: (*i*) `countOrder`, used to store the list of natural numbers we are counting over in pairs of successive numbers, and (*ii*) `countFrom`, used to store the current state of the counting process.

[2]:
```
counting = actr.ACTRModel()
actr.chunktype("countOrder", ("first", "second"))
actr.chunktype("countFrom", ("start", "end", "count"))
```

Let's say we want to simulate counting from 2 to 4. We do so by encoding these two parameters in the goal buffer:

[1] It is the first model in the tutorial units available on the official ACT-R website http://act-r.psy.cmu.edu/.

[3]:
```
counting.goal.add(actr.chunkstring(string="""
    isa     countFrom
    start   2
    end     4
"""))
```

Next, we will store counting knowledge in declarative memory. Since counting goes only up to 4 in our toy example, we will only store the first four numbers and their successors:

[4]:
```
dm = counting.decmem
dm.add(actr.chunkstring(string="""
    isa     countOrder
    first   1
    second  2
"""))
dm.add(actr.chunkstring(string="""
    isa     countOrder
    first   2
    second  3
"""))
dm.add(actr.chunkstring(string="""
    isa     countOrder
    first   3
    second  4
"""))
dm.add(actr.chunkstring(string="""
    isa     countOrder
    first   4
    second  5
"""))
```

Finally, our model will have three rules: `"start"`, `"increment"` and `"stop"`. The `"start"` rule is specified below.

[5]:
```
counting.productionstring(name="start", string="""
    =g>
    isa     countFrom
    start   =x
    count   None
    ==>
    =g>
    isa     countFrom
    count   =x
    +retrieval>
    isa countOrder
    first   =x
""")
```

[5]:
```
{'=g': countFrom(count= None, end= , start= =x)}
==>
{'=g': countFrom(count= =x, end= , start= ), '+retrieval': countOrder(first= =x,
second= )}
```

Recall that rules are conditionalized actions and ==> separates preconditions from actions. In this rule, the preconditions simply state that the goal buffer must have a chunk that has no value for the slot `count`. Furthermore, the slot `start` has the value `=x` (since `=x` does not appear anywhere in preconditions, this is trivially satisfied). As for the actions, the rule specifies changes in two buffers: the goal buffer (lines 7–9) and the retrieval buffer (lines 10–12). The ACT-R model will change the value of the slot `count` to the value assigned to the variable `=x`. This means that the value of the `count` slot in the goal buffer will be matched to the value of the `start` slot. Second, we place a retrieval request for a declarative memory chunk that has the value `=x` in the slot `first`. That is, we want to recall the successor of `=x` from memory.

The `"increment"` rule below has preconditions involving the goal and retrieval buffers. It requires the value of `count` in the goal buffer to not match the final, `end` number (lines 4-5). This is achieved by specifying that `count` has the value `=x` and `end` does not have the same value (`~` is negation). Second, the retrieval buffer carries a chunk whose `first` value matches the `count` value in the goal buffer. This condition will be satisfied if the retrieval request placed by the rule `"start"` succeeds. If these preconditions are satisfied, we trigger two actions (lines 11-16). First, the current `count` value will be updated to the value of its successor, which is the value stored in the `second` slot of the chunk in the retrieval buffer (lines 9 and 13). Second, we place a retrieval request for the next increment, i.e., the successor of the updated count (lines 14-16).

[6]:
```
counting.productionstring(name="increment", string="""
    =g>
    isa     countFrom
    count   =x
    end     ~=x
    =retrieval>
    isa     countOrder
    first   =x
    second  =y
    ==>
    =g>
    isa     countFrom
    count   =y
    +retrieval>
    isa     countOrder
    first   =y
""")
```

[6]:
```
{'=g': countFrom(count= =x, end= ~=x, start= ), '=retrieval': countOrder(first=
=x, second= =y)}
==>
{'=g': countFrom(count= =y, end= , start= ), '+retrieval': countOrder(first= =y,
second= )}
```

Finally, if the current count matches the final number (specified in the slot `end`), the `"stop"` rule clears the goal buffer, indicating that the counting goal has been achieved.

[7]:
```
counting.productionstring(name="stop", string="""
    =g>
    isa     countFrom
    count   =x
    end     =x
    ==>
    ~g>
""")
```

[7]:
```
{'=g': countFrom(count= =x, end= =x, start= )}
==>
{'~g': None}
```

We can now run the counting model:

[8]:
```
counting_sim = counting.simulation()
counting_sim.run()
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: start')
(0.05, 'PROCEDURAL', 'RULE FIRED: start')
(0.05, 'g', 'MODIFIED')
(0.05, 'retrieval', 'START RETRIEVAL')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'NO RULE FOUND')
(0.1, 'retrieval', 'CLEARED')
(0.1, 'retrieval', 'RETRIEVED: countOrder(first= 2, second= 3)')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: increment')
(0.15, 'PROCEDURAL', 'RULE FIRED: increment')
(0.15, 'g', 'MODIFIED')
(0.15, 'retrieval', 'START RETRIEVAL')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'NO RULE FOUND')
(0.2, 'retrieval', 'CLEARED')
(0.2, 'retrieval', 'RETRIEVED: countOrder(first= 3, second= 4)')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'RULE SELECTED: increment')
(0.25, 'PROCEDURAL', 'RULE FIRED: increment')
(0.25, 'g', 'MODIFIED')
(0.25, 'retrieval', 'START RETRIEVAL')
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.25, 'PROCEDURAL', 'RULE SELECTED: stop')
(0.3, 'retrieval', 'CLEARED')
(0.3, 'PROCEDURAL', 'RULE FIRED: stop')
```

```
(0.3, 'retrieval', 'RETRIEVED: countOrder(first= 4, second= 5)')
(0.3, 'g', 'CLEARED')
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3, 'PROCEDURAL', 'NO RULE FOUND')
```

The counting process unfolds in the expected way. The model starts at number 2: rule `"start"` is selected at 0 ms and fires 50 ms later (lines 4-5). The retrieval request for the successor of 2 is placed at the 50 ms point (line 7) and is completed successfully at the 100 ms point (line 11).

At this point, the preconditions of the `"increment"` rule are satisfied, so the rule is selected at 100 ms and fires at 150 ms. The current count is updated to 3 (the g buffer is modified on line 15) and a retrieval request for the successor of 3 is placed.

The retrieval is completed at 200 ms (line 20), at which point the `"increment"` rule is selected again and fires at 250 ms. Yet again, the current count is updated (line 24), reaching the end goal of 4, and a retrieval request is placed (line 25). The retrieval request is not needed but it is still placed as part of the actions triggered by the `"increment"` rule.

However, at the same time (that is, we're still at 250 ms) the preconditions of the `"stop"` rule are satisfied, since the current count matches the end number. The `"stop"` rule is therefore selected (line 27) and fires 50 ms later (line 29). We are now at 300 ms. The retrieval request for the successor of 4 was successful (line 30), but the counting process is over and the g (goal) buffer is cleared (line 31).

In sum, the model simulates basic counting by successor finding, i.e., incrementing by one. Obviously, this is too trivial compared to how adults actually count, but children arguably learn counting by incrementing by one and only later generalize this procedure. At the same time, children memorize particularly frequent (hence, useful) cases of counting. For more details about ACT-R modeling of arithmetic learning, see:

- **Lebiere, Christian. 1999. The dynamics of cognition: An ACT-R model of cognitive arithmetic. *Kognitionswissenschaft* 8:5–19**

[ ]: