Brief Introduction to ACT-R for Soarers: Soar and ACT-R Still Have Much to Learn from Each Other

Richard M Young

Psychology Department
University of Hertfordshire

Talk presented at 19th Soar Workshop
University of Michigan, Ann Arbor
21st-23rd May 1999

Overview and Background

- This talk is in two parts
  1 A v short introduction to ACT-R (John L suggested) (definitely non-standard: specifically for Soarers)
  2 A partial comparison between Soar and ACT-R, arguing that they both still have crucial things to gain from the other

- How I found myself in this position …
  — this past semester, I've taught a new option on Cognitive Modelling to final-year Cog Sci u/gs
  — decided to base it heavily on ACT-R (brief look at other architectures and issues too)
  — I've had to learn ACT-R as we go along (a good forcing function, but I wouldn't really recommend it)
  — I'll be attending the ACT-R summer school this year

Introduction to ACT-R

- A long history of books and architectures (Anderson 1976, 1983, 1990, 1993)
  — some “implemented”, some not
- A new book: J R Anderson & C Lebière (1998), *The Atomic Components of Thought*. Erlbaum.
  — very different flavour to previous versions of ACT
  — 1998 book:ACT + UTC book: Soar
- There’s
  — a web site <http://act.psy.cmu.edu>
  — a good on-line tutorial (many of the models overlap with those in the book)
  — the book models are themselves available on-line
  — there’s an annual 2-week summer school, etc.
- In this talk, by “ACT” I mean ACT-R 4.0, as described in the book
  — the actual software can be made less constrained, more flexible, more “standard production system”

Characteristics of ACT-R

- ACT has a *symbolic* aspect (symbolic memory and production system), realised over a *subsymbolic* mechanism.
  - — "subsymbolic" means "squishy": real-valued quantities and continuous maths; networky; a bit connectionist-like; activations, strengths, etc.
- There is an underlying theory, called *rational analysis* (Anderson, 1990, The Adaptive Character of Thought)
  - — provides a non-arbitrary basis for design decisions about the subsymbolic mechanisms.

Rational analysis involves

1 Making evidence-based assumptions about the statistical structure of the environment.

2 Deriving — mathematically, lots of Bayesian statistics — the optimal strategies for dealing with such an environment.

3 Assuming that those optimal strategies describe approximately what the human cognitive system does.

Structure of the ACT Architecture

This page shows Figure 1.2

from Anderson & Lebière (1998)

Memories

- Has two (main) memories: *procedural* and *declarative*
- **Procedural** memory is long-term, and holds *productions* (i.e. production rules)
- **Declarative** memory is long-term, and holds *chunks*
  - — an ACT chunk is very similar to a Soar object
- Note that these are *both* static, quasi-permanent memories:
  - — can be added to only by learning, not by RHS actions
  - — items once there tend to remain (unless decay)
- There is also a **goal-stack**
  - — which also holds chunks (encoding goals)
  - — only the top (current) goal is visible
  - — its slots are highly dynamic: writeable by RHS actions
  - — an admitted weakness of ACT is that currently the goal stack is effectively outside of the theory
- The correspondence is something like this:

ACT	+?	Soar
proc mem	♠	PM
decl mem	♠	PM

goal stack ♠ WM

Productions

- Productions have conditions and actions, as we'd expect, but the behaviour is more restrictive than Soar's
  - — grainsize ≈ Soar operator (default .05 sec)
- The main actions on the RHS are to
  - — write (change) a slot (= attribute) of the current goal
  - — manipulate the goal stack (push, pop, change)
- *First* condition of every rule is a test against current goal.
- The (zero or more) further conditions consist of *retrievals* against decl memory:

  goal-test & retrieval1 & retrieval2 … --> action & action …
- Conflict resolution (which we'll describe in a moment) selects a single rule to (try to) fire, *based solely on the goal test*. For the chosen rule, the process is:
  - — match goal test to current goal (bind vbls, etc.)
  - — attempt to match retrieval1 condition against decl mem; if fail, then rule fails (no backtracking)
  - — attempt to match retrieval2 condition … etc.
  - — perform RHS actions

- If the rule fails, try the next rule in conflict resolution order. If all fail, then pop the current goal with failure.

Conflict Resolution

- Conflict resolution is handled subsymbolically, and puts rules (NB not instantiations) in order of preference, then picks the top one
- For each rule, compute the *expected gain*

$$\boxed{E = PG - C}$$

P	probability that will eventually attain goal if fire this production
G	value of the goal (usually in units of time)
C	total cost of reaching the goal by a route that includes firing this production (same units as G)

Quantities Underlying Expected Gain

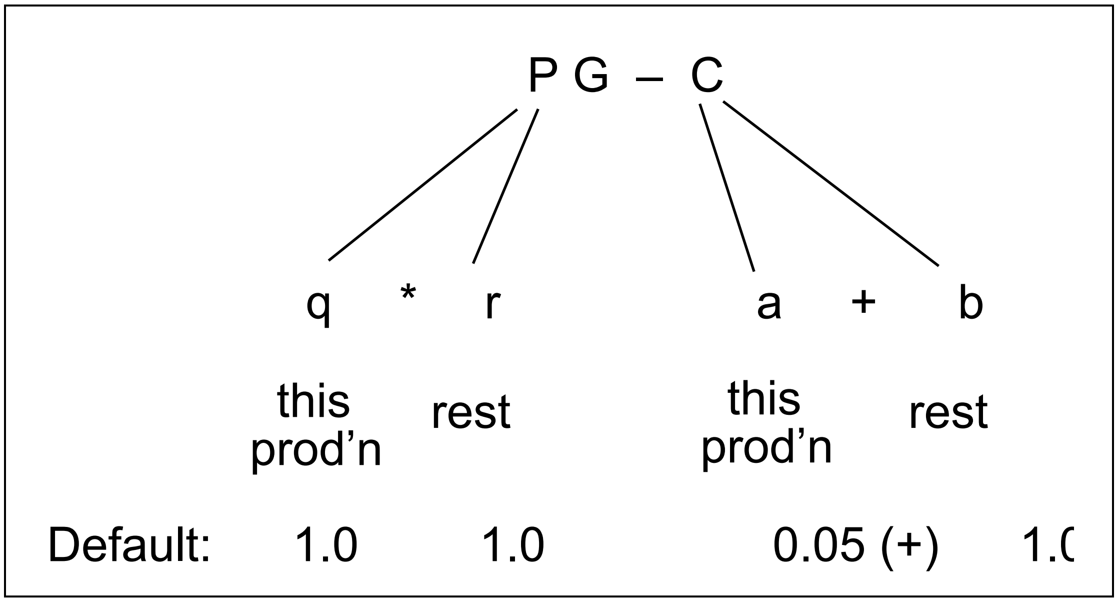
$$\boxed{E = PG - C}$$

$P = q\,r$

- q probability that this production will fire successfully
- r probability that will be able to complete the *rest* of the route to the goal

$C = a + b$

- a cost of firing this production (+ achieving any subgoals + carrying out any external action)
- b cost of the *rest* of the route to the goal



- Noise is added, both for tie-breaking, but also for deeper reasons based on rational analysis (cf. “masking”)

Learning Declarative Chunks and Productions

- Learning “chunks” (items) in decl memory is simple & uniform: each time the goal stack is popped with success, the popped goal becomes a chunk in DM
  — or boosts activation of an existing chunk

- Perceptual encoding is the only other source of declarative knowledge.

- Learning productions is neither simple nor uniform.

  Basically, you form a special kind of object called a “dependency”, which effectively *describes* the rule you want to learn. You push that dependency as a subgoal, and when you pop it, the rule is acquired in PM.
  — cf. the *build* command in some earlier prod’n sys

  Soarers would have things to say about that mechanism, not all of it complimentary. (But more on that later.)

- However, it is capable of learning all 6 kinds of rules:
  {modify goal | not} ∞ {push | pop | neither}

Learning Declarative Parameters

- The subsymbolic parameters associated with decl chunks are
  - — base-level activations
  - — strength of associations
- They are changed by *usage*, but not by *success/failure*.
- Each chunk$_i$ has an activation level, A_i, given by

$$A_i = B_i + \sum_j W_i S_{ji}$$

- Activation is interpreted as a measure (“log posterior odds”) that the chunk will be used to match to a production in the next cycle. “Learning … comes down to using past experience to estimate the quantities these parameters are supposed to reflect.”
- B is the chunk’s *base-level activation*, and reflects “how recently and frequently it is accessed”.
- Based on rational analysis, we use

$$B = \log\left(\sum_k t_k^{-d}\right)$$

  where the t_k are the times since the chunk was encountered, and d is a decay parameter, default 0.5.

Learning Decl’ve Parameters, cont.

- W_j is the *source activation* of the *j*th filled slot of the current goal.

  If the goal has n filled slots, the source activation of each is taken to be 1/n. This is not subject to learning.

- The S_{ji} is the *strength of association* between the *j*th goal slot and the chunk. It is “a measure of how often the chunk has been needed when j was an element of the goal”. It “can be thought of as an estimation of the likelihood of j being a source present if the chunk is retrieved”.

- In other words, roughly, it measures the extent to which goal slot j and chunk i “go together”.

- When a chunk is created, it is given default values for S_{ji}. With experience as the model runs, these S_{ji} are adjusted towards an estimate of their true values.

- To complete the story on rule firing: With multiple matches, the chunk which gets retrieved is the one with the highest activation, A_i
  - — With noise, the choice is partly probabilistic — a “soft max” — similar to that for conflict resolution

Learning of Procedural Parameters

- q is the probability that a particular production will successfully apply. Suppose we know how often it applies successfully and how often it fails, then we have

$$q = \frac{\text{Successes}}{\text{Successes + Failures}}$$

- ACT takes the view that the number of successes, say, includes both successes *before* now (*prior* successes) and successes met during the running of the model (*experienced* successes). So the formula becomes

$$q = \frac{\text{prior successes + experienced successes}}{\text{prior suc + exp suc + prior fail + exp fail}}$$

- A similar argument and formula for r, except of course that successes are the "eventual successes" and the failures are the "eventual failures", where *eventual* refers to what happens *after* the firing of this rule, with regard to achieving the goal (or not).

- Similarly for estimating the *cost* parameters, a and b.

$$a = \frac{\sum\text{efforts}}{\text{no. of firings}} = \frac{\sum\text{efforts}}{\text{successes+failures}}$$

- A time-based decay version of these formulas can be used, as for base-level activation.

ACT Model of Menu-Scanning Experiment

- Anderson, Matessa & Lebière (1997) model of an experiment by Nilsen (1991), also modelled in EPIC by Kieras & Meyer (1997) in same issue of journal.
- S puts mouse on menu header: click, and a vertical menu appears, digits 1-9 in random order. With a target item specified , have to move mouse to it and click it.
- Because of random order, have to scan the menu visually to find target. Ss tend to scan vertically downwards, and tend to move the mouse along with the scanning. So, once found, there is little variation in mouse movement time. The variation in time comes from how far need to scan through the menu.
- Assumptions of the ACT model are in line with those of an earlier model dealing with letters and numbers. Letters and numbers are assumed to be composed of a number of *features*. McClelland & Rumelhart (1981), for a famous model of “interactive activation”, proposed a definite set of features for upper-case letters and numbers, and this is what the ACT model uses too.

ACT Model: Doing the Task

- The ACT model of how the task gets done is very simple.
  1) Choose a random feature of the target digit.
  2) Shift attention downwards to next occurrence of that feature.
  3) If that item is the target letter, then move mouse to it and click.

- The model has just two interesting rules:

```
Hunt-Feature
IF the goal is to find a target
   that has feature F
   AND there is an unattended
   object below the current
   location with feature F
THEN move attention to closest
   such object
```

```
Found-Target
IF  the goal is to find a target
   AND the target is at loc’n L
THEN move mouse to L and click
```

Matching the Data

- The empirical data are that total reaction time varies as a linear function of target position, with a slope of 103 msec per position.
- The number features are such that with p = .53, a randomly chosen feature of one number will occur as a feature in another, given, number. Using the figure from the Sperling experiment of 185 msec to switch attention, the ACT model will predict a linear function with a slope of

  $$185 \ \infty \ .53 = 98 \text{ msec per position.}$$

  (Compare with 103 msec in data.)

Stressing the Model: Distractors

- If the model is right, it should also predict the effect of using different *distractors*. Instead of searching for a ‘6’, say, in a background of other digits, Ss could be asked to find a *letter* in a background of digits.
- The probability of a random letter feature being shared with a given digit is .42.
- So, the ACT model predicts a slope of

  185 ∞ .42 = 78 msec per position.

  (Actual slope is 80 msec.)

NOTE

- There is a distinctive *style* of using the ACT model:
  - — The production system model is (ridiculously) simple, and so is the task.
  - — So, no great achievement to say “Look, an ACT model can do the task”.
  - — The model serves as basis for a simple quantitative analysis.
  - — The model is to be assessed on its quantitative fit to the empirical data.

Soar and ACT-R

- In this second half of the talk, I want to argue that each of Soar and ACT-R still have much to gain from the lessons of the other.

- I'll play it mostly as "Here's an example where Soar can learn from ACT" and "Here's one where …"
  — but I also want to emphasise that in many cases the situation is more complex (and more interesting) than "Soar good, ACT bad" or vice versa

HEALTH WARNINGS

- My own personal views, etc. …

- In this talk, "constraint" means a positive term, a virtue
  — see discussion in *UTC* or, e.g., Howes & Young 1997

WARNING: R18 certificate

Some of what follows may be controversial

- Warning: I'll probably several times say "Soar" when I mean "ACT"
  — what kind of cognitive model or architecture would

explain that kind of persistent error?

ACT ❷ Soar: Fine-Grain Data

1 ACT allows us to write quick & simple models for quick & simple tasks

- — e.g. the menu scanning task: v simple prod'n system, but basis for interesting predictions, with close contact to empirical data
- — in class I used a 1-rule model, "think of a name for a dog": get simple but non-trival effects, e.g. of learning

2 ACT models fit to actual performance times and error rates

- — track small shifts with experimental conditions
- — fit fine-grain effects on latencies

• Mostly, with Soar this has not been done. Closest:

• Wiesmeyer, visual attention. Impressive integration across experimental paradigms with single 50 msec constant. But

- — weak on graded effects (& never published properly!)

• Miller & Laird, SCA: graded effects, and human-like curves. But

— not aim at actual quantitative fit (go for surprise value)
— serious problems (e.g. saturation)

Soar ❷ ACT: Rule Learning

3 Soar's (rule) learning mechanism is elegant, simple, ubiquitous, automatic (architectural), and rational.

- ACT's rule learning is reflective (post-event) only, too deliberative, arbitrary (lack of constraint), knowledge-based instead of architectural (self-programming)
- Anderson & Lebière raise various objections to Soar's learning mechanism ("[data chunking] a problematic aspect of Soar"; "too many productions"; "excessive production rule formation")
- But they don't stand up
  - — Soar's data chunking is an important source of constraint
  - — supports learning as a side-effect of performance
  - — linked to rational aspects of cognition (Young & Lewis, 1999)
  - — not even clear that a (non-arbitrary) ACT wouldn't be subject to same phenomenon
  - — what on earth is "too many" productions?
  - — a similar mechanism is anyway used for ACT's declarative learning

Soar   ACT: Modelling a Subject

4 Anderson & Lebière say [approximately, I can’t find the quote] “An ACT model is always a model of a human S in an experimental situation”

- This is a crucially important point, but decidedly 2-edged

ACT ❷ Soar

- Means that an ACT model always grounded in empirical data, and can be put into contact with those data.

Soar ❷ ACT

- But also a limitation of ACT. Never gets to deal with human as an autonomous agent with full range of capabilities
  - — only some of which get tapped, selectively, in setting of a psychological experiment
  - — basically, an “ecological validity” point
- Has negative impact on aspects where Soar strong
  - — functionality
  - — integration across different aspect of cognition
  - — capability-based accounts of cognitive performance

Soar ? ACT: Constraints

5 Because of its focus on constraint, Soarers can play the game of “Listening to the architecture”
  - not “how do I think people do this task?”, but “given the task, how would Soar do it?”

- Until recently, couldn’t consider doing this with ACT
  - too much choice of mechanism & parameters
  - too little constraint
- But with ACT-R 4.0 now becoming possible, and to some extent being done, i.e. write simple model “to do” task, see what ACT predicts about performance & learning
- ACT now arguably more constrained than Soar
- Productions very constrained
  - conflict resolution only on goal test
  - no backtracking in retrieval; etc.
- Severe constraints on dynamic task-related information
  - decl mem is basically a *permanent* store: info there subject to activation & decay effects
  - (soft) constraints on goal slots: total activation divided among # filled goal slots, so if too many,

may not retrieve intended items from memory

Soar ?❷ ACT: Lean Architecture

6 Soar clearly wins on having a lean, fixed architecture
  — no choice as to what mechanisms to invoke, or what to appeal to for a particular task or data set

- Might argue, therefore, that Soar wins on constraint
  — however, I'm not entirely convinced

- Problem seems to be that Soar's home territory is *so* distant from many tasks of interest, that lots of ingenuity is needed in order to see how it would do the task at all e.g. SCA on concept learning

- Consequently, leaves room for lots of different routes, depending upon imagination of analyst
  — hence, not effectively constrained

- View "all cognition as problem solving in a problem space"
  — leads to difficulty where, to be honest, view doesn't fit

Soar ❷ ACT: Depth of Explanation

7 ACT lends itself to “data modelling”, which provides little by way of “explanation”

— details of many ACT set by what look like convenient ways to match the data, rather than by arguments based upon functionality

— can be subtle (e.g. Anderson & Liebière, p.330)

— history of too much *ad hoc* parameter tuning (though now much improved)

- Soar (at least, when used carefully & well) lends itself to deeper questioning about “How does it come to be doing the task like that?” E.g.

  — attempts at performing task based on instructions

  — the data-chunking story, and its relation to learning from instructions

  — the NL story

- This is an important issue for *integration*

ACT ❷ Soar: Easier to Learn

8 We've been told several times that Soar is too hard to learn and that ACT is easier

— was one reason why I picked ACT for my u/gs

— also, good support from textbook and web tutorial

• I'm not entirely convinced. Story goes something like this:

• There's *much* more to learn about ACT

— far more variety of mechanism

— rules, chunks, conflict resolution, activations, associative strengths, noise, rule strengths (haven't mentioned that), decay, probability estimates, …

— book (pp.99-100) 33 standard numerical parameters; (pp.432-3) 20 basic equations

• However, the crucial difference (I think) is that ACT doesn't have any brick walls and vertical cliffs on its learning curve, whereas Soar clearly does.

You can know just a little about ACT, and write simple models. Then learn some more, expand your repertoire.

• Seems not to be true of Soar

— though recent tutorials (e.g. Eaters) are trying hard to move in that direction

ACT ❷ Soar: Simple Maths Models

9 ACT lends itself well to a variety of approximate mathematical models

- Ranging from simple arithmetic
  — menu scanning model
  to sophisticated Bayesian analysis
- Getting an analytical grip on Soar models (beyond just “50 msec per operator”) has proved elusive.