Optimising the ACT-R paired associate model using HTCondor

David Peebles

6th May 2016

Introduction

This document describes the files in this repository that will allow you to run multiple copies of an ACT-R model using a High Throughput Computing (HTC) environment managed by HTCondor. It describes the required changes to the model code to allocate random values to the parameters and the key commands in the HTCondor *submit description file* to run the model on a network of 64-bit MS Windows computers.

This repository contains four key files: *paired.lisp* which contains the ACT-R model code taken from Unit 4 of the ACT-R tutorials, *actr-s-64.exe* the standalone ACT-R executable for MS Windows, *paired.job* which is the HTCondor submit description file, and *new-random-state.lisp* which creates a random module to address certain issues with random number generation (discussed further below).

A detailed description of the paired associate task and the ACT-R model can be found in the documents *unit4.pdf* and *unit4_exp.pdf* from the tutorial unit (available in the repository). The paired associate model is an ideal example to use because it is relatively simple but requires the four parameters shown in Table 1 to be estimated (although, as with most models, the base-level learning parameter remains unchanged from its recommended value of 0.5).

Table 1: Optimal parameter values for ACT-R paired associate model

Parameter	Label	Value
Retrieval threshold	:rt	-2
Latency factor	:lf	0.4
Instantaneous activation noise	:ans	0.5
Base-level learning rate	:bll	0.5

Required changes to the model code

The aim of this exercise is to use HTCondor to explore cognitive model parameter spaces by running multiple copies of the model, each with a different set of randomly generated parameter values, and then analysing the returned outputs to identify promising candidates.

The values for each parameter are typically constrained between maximum and minimum bounds so the *bounded-random* function returns a real-valued random number between max and min (ensuring that the random function is passed a floating point to avoid integer only results). This function can be found in the file *new-random-state.lisp*.

```
(defun bounded-random (min max)
  (let ((delta (- max min)))
    (+ min (if (integerp delta)
               (act-r-random (* 1.0 delta))
             (act-r-random delta)))))
```

To allocate the values, the parameter assignments in the model definition are commented out in the *sgp* ACT-R command and replaced by the *sgp-fct* function. In addition the command that sets the random seed is commented out as this has the effect of ensuring the same random numbers are created for every run of the original model.

```
(sgp :v nil
     :esc t
  ;; :rt -2
  ;; :lf 0.4
  ;; :ans 0.5
     :bll 0.5
     :act nil
     :ncnar nil)

 ;; (sgp :seed (200 4))

(sgp-fct (list
          :rt (bounded-random -5.0 -0.01)    ;; retrieval threshold
          :lf (bounded-random 0.01 0.8)      ;; latency factor
          :ans (bounded-random 0.01 0.8)))   ;; activation noise
```

Finally, to identify the set of parameter values associated with the model run output, the following ACT-R command is added to the *output-data* function in the model file which outputs the parameter values to stdout[1].

```
(sgp :rt :lf :ans)
```

Random number generation

A potential problem arises with the generation of random numbers related to how various Lisp implementations set the seed for the pseudorandom number generator (PRNG). In particular, it is crucial that each instance of a model is initialised with a unique seed. Many Lisps (and the ACT-R random module) use the computer's clock to generate the starting seed but this may cause a problem if different instances are given the same clock time when they are loaded or run.

One way to reduce the likelihood of this occurring is to always use the ACT-R PRNG and create a version which, rather than using the system clock, uses a random number generated from Clozure Common Lisp (the Lisp implementation used to create the MS Windows standalone ACT-R executable, which doesn't use the system clock anyway) to seed ACT-R's PRNG. This method is employed in a new ACT-R *create-random-module* function (in the file *new-random-state.lisp*) which is loaded prior to loading the model file.

```
(defun create-random-module (ignore)
  (declare (ignore ignore))
  (let* ((state (make-mersenne-twister))
         (vec (coerce (ccl::random.mrg31k3p-state (make-random-state t)) 'simple-vector)))
    (init_by_array vec state)
    (make-act-r-random-module :state state)))
```

Creating a submit description file

To submit jobs for execution under HTCondor, one must create a *submit description file*. The file *paired.job* in the repository is a submit description file which contains a minimum set of commands to run 100 instances of the paired model on a network of on a network of 64-bit MS Windows computers[2].

To run this submit description file will require finding out the particular features of ones own local HTC environment and probably some appropriate modification but I discuss the commands specific to running the ACT-R model below.

[1] All changes to the ACT-R model file can be found by searching for 'HTC'

[2] All commands can be found in the HTCondor manual

The four lines below specify the MS Windows standalone ACT-R executable as the executable to be run and that it should always be transferred to the computers in the network. Line 3 specifies the argument to the executable, a string containing the sequence of files to load (-l) and then functions to execute (-e). The first of these runs the model while the second exits the Lisp environment. Finally, the fourth line specifies the code files to be transferred to the network computers.

```
executable = act-r-s-64.exe
transfer_executable = ALWAYS
arguments = "-l 'new-random-state.lisp' -l 'paired.lisp' -e '(paired-experiment 100)' -e '(quit)'"
transfer_input_files = new-random-state.lisp, paired.lisp
```

The next four lines below specify the file names of the output, error and log files that will be generated by each model run. Each model run in a job will be created with the same job 'cluster' number and a unique 'process' number.

```
output = out.stdout.$(Cluster).$(Process)
error = out.err.$(Cluster).$(Process)
log = out.clog.$(Cluster).$(Process)
```

Finally the command below submits 100 instances of the model to the queue as one job.

```
queue 100
```