In D. Reitter & F. E. Ritter (Eds.), Proceedings of the 14th International Conference on Cognitive Modeling (ICCM 2016). University Park, PA: Penn State.

Using a cognitive architecture in educational and recreational games: How to incorporate a model in your App

Niels A. Taatgen (n.a.taatgen@rug.nl) and Harmen de Weerd (h.a.de.weerd@rug.nl)
Institute of Artificial Intelligence, Nijenborgh 9
9747 AG Groningen, The Netherlands

Abstract

We present a Swift re-implementation of the ACT-R cognitive architecture, which can be used to quickly build iOS Apps that incorporate an ACT-R model as a core feature. We discuss how this implementation can be used in an example model, and explore the breadth of possibilities by presenting six Apps resulting from a newly developed course in which students make use of Swift ACT-R to combine cognitive models with mobile applications.

Keywords: ACT-R, mobile apps, game design

Introduction

Cognitive models have proven to be a valuable research tool in advancing our understanding of human cognition. Because of their ability to model human behavior, cognitive models also have a great potential for use outside of research, such as in educational or recreational settings. In this role, the model is not used to explain human data, but to act as a simulated human agent. In previous cases, such as the ACT-R model that played SET (Taatgen et al., 2003), and DISTRACT-R (Salvucci et al., 2005), the model was implemented directly in the target programming language. In this paper, we present an ACT-R re-implementation that can be used as a component in an iOS App. The implementation makes it possible to quickly build Apps with ACT-R inside. As a demonstration, we present a Rock-Paper-Scissors App that the first author built in just one-and-a-half hour. We further look at the results of a course that we taught using the implementation, and the six Apps that came out of that course.

Swift ACT-R

The re-implementation of ACT-R uses the new Swift programming language. Swift is an object-oriented programming language similar to Java and C++. The Swift implementation of ACT-R consists of a set of classes that implement the different components of ACT-R, such as Chunks, Declarative Memory, Procedural Memory, and the overarching Model class.

The simplest way to use Swift ACT-R is to write a text-file with a regular ACT-R model (with some limitations). The next step is to build a controller for the App, that responds to button presses and other actions the user can take. This controller creates an instance of the model class, and loads the ACT-R model into that instance:

```
model = Model()
model.loadModel("example")
```

The model can then be run using the `run` method:

```
model.run()
```

The model communicates with the App through the action buffer (a new buffer that takes the role of standard perception and action buffers). Whenever a production rule takes a `+action>` action, the model stops, and hands control back to the main program. The main program can then read out the contents of the action buffer, make appropriate changes to the interface, wait for user input, place information back into the action buffer, and then run the model again.

There are several alternatives to using ACT-R code, for example, it is also possible to access declarative memory directly, or even to have no explicit ACT-R model, but instead use declarative memory directly. The ACT-R code can be downloaded from:
https://github.com/ntaatgen/ACT-R
It has two example models, both of the prisoner's dilemma (Lebiere, Wallach & West, 2000 and Stevens, Taatgen, & Cnossen, 2016).

Example model: Rock - Paper - Scissors

Lebiere and West (1999) built an ACT-R model that can play Rock-Paper-Scissors, and adapts itself to its opponent by trying to predict the next move based on previous experiences. The lag 1 model of Lebiere et al. stores sequences of two consecutive moves of the opponent in declarative memory and uses these to predict the opponent's next move. For example, the model has the following chunks for sequences that start with rock:

```
(RR isa decision step1 rock step2 rock)
(RP isa decision step1 rock step2 paper)
(RS isa decision step1 rock step2 scissors)
```

Each time the opponent plays rock twice in a row, the RR chunk is strengthened, each time rock is followed by paper, the RP chunk is strengthened, and each time rock is followed by scissors, the RS chunk is strengthened. When the model needs to decide what to do in the turn after the opponent has played rock, it retrieves the most active chunk with rock in step1. The value in step2 is then the model's prediction for the next move of the opponent. It only needs to decide what move to counter that with. The whole model consists of only four production rules (actually, five: one more rule to play the first game, when there is no previous decision). Figure 1 lists these productions.

```
(p retrieve-decision
    =goal>
        isa goal
        state start
        playerlast =last
==>
    =goal>
        state retrieve
    +retrieval>
        isa decision
        step1 =last)

 (p retrieve-beats
    =goal>
        isa goal
        state retrieve
    =retrieval>
        isa decision
        step2 =prediction
==>
    =goal>
        state retrieve-beats
    +retrieval>
        isa beats
        slot1 =prediction)

(p make-decision
   =goal>
     isa goal
     state retrieve-beats
   =retrieval>
     isa beats
     slot2 =decision
==>
   =goal>
     state decide
   +action>
     isa move
     choice =decision)

(p restart-after-action
  =goal>
    isa goal
    state decide
    playerlast =last
  =action>
    isa move
    opponent =decision
==>
  +goal>
     isa goal
     state start
     playerlast =decision
  +imaginal>
    isa decision
    step1 =last
    step2 =decision
  -action>)
```

Figure 1: Productions in the Rock-Paper-Scissors model

The model makes a decision in three steps. It first retrieves its prediction for what the opponent will do next based on their previous move. Based on that prediction, it will retrieve from memory the move that will beat the predicted action (e.g. rock beats scissors). It will then put this action into the action buffer. Control is then returned to the main program, which waits until the human player takes an action by pushing one of three buttons in the interface (Figure 2).

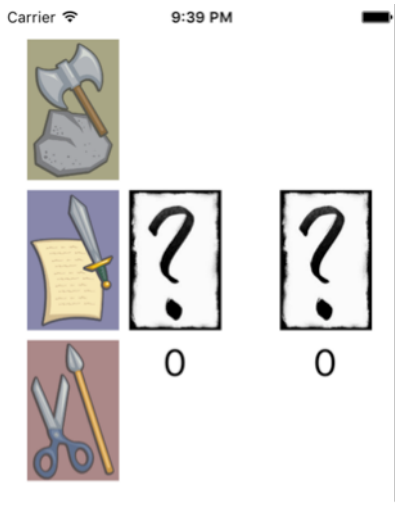


Figure 2. The Rock-Paper-Scissors game on an iPhone

The program itself is straightforward. The function that is called when the App is started already loads in the model and carries out a first run. The model has therefore made its decision, and now waits for the player to tap one of the buttons. Once the player has made a decision, the code checks who has won, and adjust the scores. Figure 3 shows all the basic code that is necessary. Some additional code is needed to update the display with appropriate feedback, and show the scores.

To explore the breadth of possibilities of constructing Apps with a built-in model, we made this the goal of an advanced cognitive modeling course.

Course outline

The course 'Cognitive Modeling: Complex Behaviour' is part of the Master Human-Machine Communication and the Master Artificial Intelligence at the University of Groningen. It has been set up as a so-called *learning community*. For the purpose of the course, students had access to a lab room with several workstations to develop apps on, as well as a number of iPads and iPhones for testing. The lab room was available to the students for the full duration of the ten-week course. In line with the concept of a learning community, the focus is on letting the students present their work for open discussion among themselves, rather than on formal lectures.

The course followed the plan outlined in Table 1. At the first meeting, students divided themselves into three-person

```
 override func viewDidLoad() { // This function is called when the App starts up
    super.viewDidLoad()
    model.loadModel("rps")
    model.run()
}

 // The following function is called when the player pushed one of the buttons
 @IBAction func gameAction(sender: UIButton) {
    // The player action is the title of the button that was pressed
    let playerAction = sender.currentTitle!
    // The model action is in the choice slot in the action buffer
    let modelAction = model.lastAction("choice")!
    // Determine the outcome of the game
    switch (playerAction,modelAction) {
    case ("Rock","rock"),("Paper","paper"),("Scissors","scissors"):
        // Tie
        break
    case ("Rock","scissors"),("Paper","rock"),("Scissors","paper"):
        // Players wins
        pScore += 1
        mScore -= 1
    default:
        // Model wins
        pScore -= 1
        mScore += 1
    }
    // Communicate the player's action back to the model by setting a slot
    // in the action buffer
    model.modifyLastAction("opponent", value: playerAction.lowercaseString)
    // And run the model again for the next trial
    model.run()
 }
```

Figure 3. Code in the App to handle the interaction between the player and the model.

project teams, and were encouraged to immediately start developing a project proposal. A project proposal was subject to two conditions: (1) the App had to be developed in Swift, and (2) the core of the App should be the Swift implementation of the ACT-R cognitive architecture. No further requirements were given, although project proposals had to be approved before a team could start. In particular, students were free to choose to build a game, an educational app, or different applications using an ACT-R model. In addition, students were free to make their App for iPad, iPhone, Apple Watch, or any combination of the three.

Each team consisted of three people, with one member being responsible for graphical user interface (GUI) design, one for cognitive model design, and for programming and coordination. The first two weeks were meant for students to familiarize themselves with the Swift programming language and the Swift ACT-R implementation. Each of these topics included a short lecture and a small, ungraded assignment.

Students presented their finalized project proposals in the third week. Over the following five weeks, students gave weekly progress reports on the status of their project, either privately with one of the lecturers, or as a presentation to fellow students to encourage discussion of common problems and solutions.

Final presentations and demonstrations of the App were due in week 8 and 9, which left the students one additional week to write a final report on their App. Mirroring the structure of the student projects, the final report was required to discuss the graphical user interface, the cognitive model, and general programming.

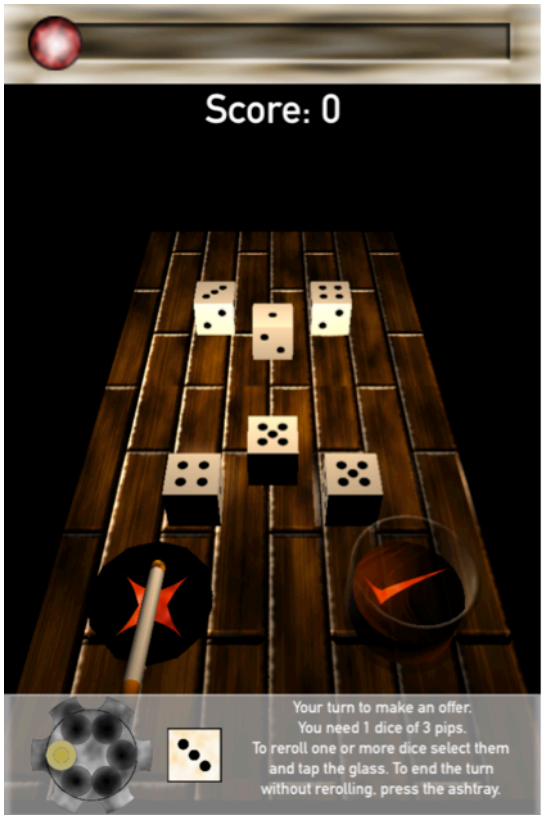
Table 1: Course plan for 'Cognitive Modeling: Complex Behavior'.

Week	Activity
1	Introductory lecture on Swift Creating project teams Assignment: Build simple calculator app
2	Introductory lecture on Swift ACT-R Assignment: Build rock-paper-scissors opponent using Swift ACT-R
3	Presentation final project proposals
4-7	Progress reports
8	Final presentation
9	Demonstration of the App and election of the best App
10	Deadline final report

Description of developed mobile apps

Six projects were developed during the course, each with a corresponding App. As mentioned in the course outline, students were free to choose the topic of their application, as long as it included Swift ACT-R as a core mechanism. The six projects included three recreational games, two educational games, and one other application. In this section, we describe each of these apps in more detail.

Six-Dice game



The six-dice game is a recreational game of incomplete information played over a number of rounds. Two players control three dice each. At the start of each round, each player is given a goal that involves a certain number of dice that should show a given number of pips. For example, the human player in the screenshot above has the goal to have at least one of the six dice show a 3, while the cognitive agent may have the goal to have one die to have rolled a 6. Note that these goals are private information. That is, neither player knows the other player's goal.

Once the goals are revealed, all dice are rolled and revealed. Next, one of the players may offer to reroll any subset of their own three dice. The other player must decide whether or not to accept this proposal. If the proposal is rejected, the round ends and each player who has achieved his or her goal gains one point. If the second player accepts the proposal, the dice selected by the proposing player are rerolled, but the second player also has to select the same number of their own dice to reroll. Note that the deciding player controls which dice are rerolled.

At the end of the game, the player with the highest score wins. However, when the combined score of both players is below a certain threshold, the game ends without a winner. The game is therefore a game of mixed motives. Especially near the end of the game, it may be in the best interest of a player to allow the opponent to reach their goal.

The ACT-R model is used to assess the opponent's trustworthiness. Each game, the model would assess the outcome of the game, and assign it a trust value between 1 and 10, and add this as a chunk to declarative memory. When it later had to make a decision in which trust of the other player played a role, it would perform a blended retrieval of the trust value.

Memory

Memory (also known as Concentration) is a recreational card game played by placing a number of cards face down on a surface. Players take turns revealing two of the cards. If these two cards form a pair, the cards are removed from the game and the player scores a point. Otherwise, the cards are placed back face down. The game continues until no cards are left, at which point the player with the most points wins.

Note that for a computer player, the game of Memory is a trivial one. After all, turning the cards face down after revealing only presents a challenge for players without a perfect memory. The goal of incorporating a cognitive agent in Memory is therefore to create a fun and challenging competitor, rather than to make an agent that follows an optimal strategy in playing Memory.

The ACT-R Memory player makes use of declarative memory to store card information such as the identities and positions of previously revealed cards. When the ACT-R player believes to have found a pair of cards, it tries to retrieve the locations and claim the pair. To simulate human-like errors, the model adds noise to the stored positions of cards. This causes cards on the edges and corners of the surface to be remembered better than interior cards.

Pyramid game

The pyramid game is a recreational game played with a standard deck of 52 cards. Ten cards are placed face down as a pyramid (see screenshot above). In addition, each player receives four facedown cards. At the start of the game, each player is allowed to look at their own four cards. A player cannot look at the cards of the other player, and once the game starts, a player is no longer allowed to review their own cards either.

The game is divided into multiple rounds. In each round, a face down pyramid card is turned over, starting at the bottom left and moving slowly up the pyramid. The value of a card depends on its position in the pyramid. Cards on the bottom row are worth 1 point, while the top card is worth 4 points. Once a card is revealed, both players decide whether or not to claim that one of their four cards has the same face value. If a player decides to do so, they select one of their four cards. The other player may then choose to challenge this claim by turning over the selected card and check its face value.

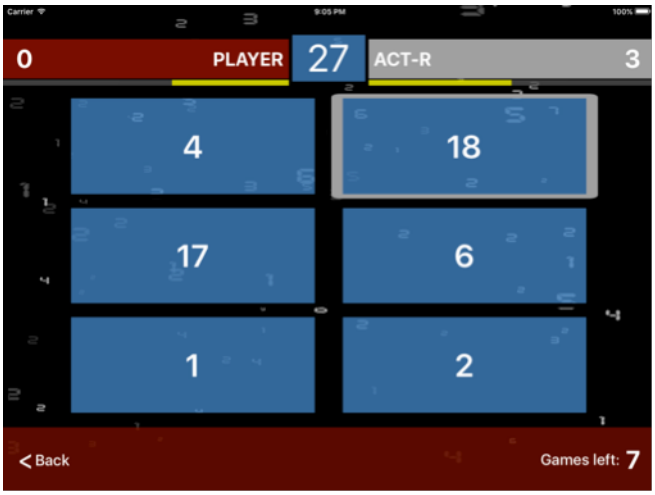
The players' scores change based on the outcome of a round. If no claim is made, scores remain unchanged. When a claim remains unchallenged, the claimant adds the value of the pyramid card to their score. If the claim is challenged and found to be false, the challenger adds twice the value of the pyramid card to their score. Finally, if a claim is challenged and found to be true, the claimant adds twice the value of the pyramid card to his score.

Independent of whether a claim was challenged, any card that was used to claim points are replaced with new cards from the deck. Only the owner of a new card is allowed to inspect it.

For the purpose of the app, the human player always starts by deciding to make a claim. Once this claim is resolved, the computer player takes its turn and the game continues to the next pyramid card. When there are no cards left on the pyramid to turn over, the player with the highest score wins.

The ACT-R opponent makes use of declarative memory to remember its own cards, and regularly rehearses these cards to avoid forgetting. In addition, the ACT-R player tries to model the behavior of the human player in terms of the likelihood that the human player is bluffing and the likelihood that the human player will challenge a claim of the ACT-R player.

Mathgician game



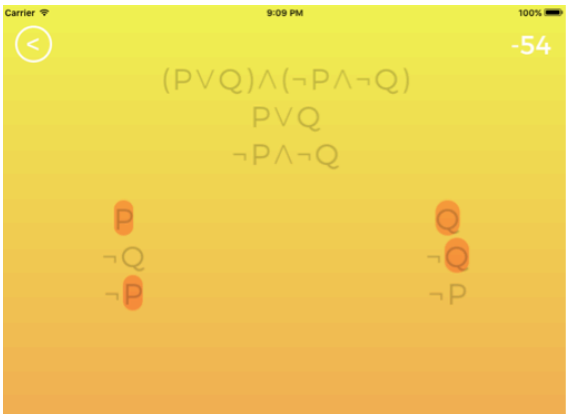
Mathgician is an educational App aimed at training addition to children in the form of a competitive game. In the game, players are presented with a goal number and six tiles. Each tile has a number printed on it. Players have to select tiles such that the numbers on the selected tiles add up to the goal number. For example, to get the goal of 27, as in the screenshot above, players could select the tiles with the numbers 17, 6, and 4, but also the tiles with the numbers 18, 6, 2, and 1.

The game is played against an ACT-R opponent, which follows a human-like greedy strategy, in which it tries to get to the goal number with high numbers first. If this fails, the model tries lower numbers. In addition, the App has the option to play against an adaptive opponent, which tries to match the search speed of the ACT-R model with the behavioral data of the human player.

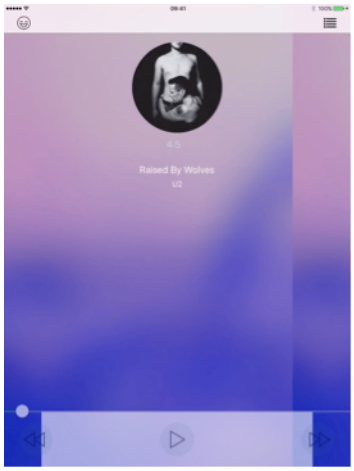
OMGLogic game

OMGLogic is an educational App that is intended to help players learn how to construct semantic tableaux. In the app, players are presented with formulas from propositional logic. They are asked to construct a semantic tableau to show that the formula cannot be satisfied. To do so, players have to select part of the formula and decide how it should be handled to continue the tableau. Whenever a player takes a correct action their score increases, while incorrect answers decrease a player's score.

The ACT-R model presents a competitor that attempts to gain points for itself while constructing the semantic tableau. The ACT-R model attempts to match the skill level of the human player in solving the formulas. If the model successfully retrieves a correct course of action before the human player, the step is executed and the human player loses points.



LagMusic app



LagMusic is a music player that makes use of ACT-R to predict when a listener wants to listen to a previously heard song again, given a user-specified mood. The model uses the actions of the user as feedback. Skipping a song is considered to be an indication that the user is unwilling to listen to the song, while a user that listens to a song for the full duration is considered to be happy with the model's selection. Finally, a user can indicate that it likes a song, but does not want to listen to it at the current moment by shaking the device.

The ACT-R model uses activation of a song chunk to determine whether, given the user's current mood, a song has sufficiently faded from memory for the user to appreciate hearing it again, as well as whether or not the user appreciates the song at all in the current mood. Whenever a song is played in full, the positive feedback increases activation. Skipping a song provides negative feedback by decreasing a song's activity. In addition, the model updates the retrieval threshold, making it more likely for the model to retrieve songs with higher activation. The neutral feedback, which indicates that the song is appreciated but played too soon, adjusts the retrieval threshold in the other direction, making it more likely for the model to retrieve songs with lower activation.

Conclusion

Cognitive models are not only useful to build theories of human cognition, but they can also be applied to build simulated humans. In this paper we explored possible ways in which the ACT-R architecture can be used in the context of a mobile application. One of the conclusions we can draw at this stage is that for most purposes declarative memory is the most useful component in ACT-R. It can model how we remember and forget information, and can also model decision making through instances-based learning. Further potential of incorporating a model in an App is that the App can gather its own information to train the model. Again, declarative learning is the lowest hanging fruit here, but procedural learning is potentially very powerful as well.

Acknowledgments

The underlying research project is funded by the Metalogue project; a Seventh Framework Programme collaboration funded by the European Commission, grant agreement number: 611073.

We want to thank the following students for their efforts in bringing the ACT-R Apps to life: Harmke Alkemade, Roberto De Cecilio De Carlos, Andrija Curganov, Thomas Derksen, Annemarie Galetzka, Joram Koiter, Michael LeKander, Milena Mandic, Rick van der Mark, Hugo van Plateringen, Tom Renkema, Jordi Top, Teun van Tuijl, Olaf Visker, Alex de Vries, Evert van der Weit, Marco Wirthlin, and Maikel Withagen.

References

Anderson, J.R. (2007). *How Can the Human Mind Occur in the Physical Universe?* Oxford, USA: Oxford University Press.

Lebiere, C., Wallach, D., & West, R. (2000). A memory-based account of the prisoner's dilemma and other 2x2 games. In N. Taatgen, & J. Aasman (Eds.), *Proceedings of the Third International Conference on Cognitive Modeling* (pp. 185–193). Veenendaal: Universal Press.

Lebiere, C., & West, R.L. (1999). Using ACT-R to model the dynamic properties of simple games. In *Proceedings of the Twenty-first Conference of the Cognitive Science Society* (pp. 296-301).

Salvucci, D.D., Zuber, M., Beregovaia, E., & Markley, D. (2005). Distract-R: Rapid prototyping and evaluation of in-vehicle interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 581-589).

Stevens, C.A., Taatgen, N.A., & Cnossen, F. (2016). Instance-based models of metacognition in the prisoner's dilemma. *Topics in Cognitive Science, 8(1)*, 322-334.

Taatgen, N.A., van Oploo, M., Braaksma, J. & Niemantsverdriet, J. (2003). How to construct a believable opponent using cognitive modeling in the game of Set. In *Proceedings of the Fifth International Conference on Cognitive Modeling* (pp. 201-206).