It’s Alive: Emergent ACT-R Voting Model Errors on Full Paper Ballots

Joshua Engels (jae4@rice.edu)

Michael D. Byrne (byrne@rice.edu)

Department of Psychological Sciences, Department of Computer Science

6100 Main St., MS-25, Houston, TX 77005 USA

Abstract

This paper presents an ACT-R model we developed to simulate voting behavior on full size paper ballots. We focus specifically on a case study of a non-standard voting strategy: the strategy votes first from left to right on a ballot and then from top to bottom. We run this model on 2772 randomly generated ballots governed by 3 different structural variables. The findings suggest that our model’s error behavior is emergent and sensitive to ballot structure. These results represent an important step towards our end goal of creating a self-contained piece of software capable of identifying bad ballot design.

**Keywords:** ACT-R; error prediction; voting

# Introduction

Voting is hard. The deliberations and conversations that go into choosing who best represents one’s interests is an important and time-consuming task, one that might be argued to be the very backbone of a democracy. Understandably, many may assume that the subsequent task of correctly voting for one’s chosen candidate is comparatively easy and straightforward. Surely once a voter gets the ballot and can mark whoever they please, the hard part is over.

For the most part, they would be right. When ballots are designed well, any errors voters make are not systematic and will not help or hurt any candidate. When they are designed poorly, they may lead to systematic voting errors, but perhaps this happens rarely or does not matter on a large scale. In the general case, people tend to make relatively few mistakes when marking a ballot with who they want to vote for.

However, in closely contested elections it is not the general case that is important. There have been numerous elections in the past 20 years that have been documented as having been decided by systematic voting errors caused by bad ballot design. While election interference by hacking is a far more flashy and obvious risk, there at least has never been evidence that this has swung an election, unlike for bad ballot design. Ironically, the fear of hacking has led to a return to paper ballots, which with their profusion of races packed onto small sheets of paper makes ballot design even more important.

People routinely make many kinds of voting errors, but the biggest errors resulting from bad ballot design are under and over voting. Under voting is an error that occurs when the voter fails to vote on a race that they intended to, whereas over voting is when a voter votes on a race more than once. The problem of designing even one ballot that will not cause people to systematically under or over vote is challenging. For instance, it might entail running a usability study weeks before the actual election. What makes the problem so difficult is the sheer number of counties in the United States, each of which designs their ballots differently and each of which have hundreds of different iterations of ballots for each precinct they are responsible for. Manually checking each ballot with a usability study is infeasible.

One possible solution to this problem is a piece of software that could automatically check an arbitrary ballot for common design errors. However, since the task is to decide if humans will make a mistake, it is initially difficult to imagine building an automatic system to do such a task. Here is where ACT-R, a cognitive architecture purpose built to simulate human cognitive tasks, comes in. In order to ensure that every mistake was predictable, any mistakes a human would make on a certain ballot would have to be made by the system as well

This is not an original idea. Green (2010) built an ACT-R model that could make the same mistake voters did in a famous ballot caused systematic error. However, these and other attempts have focused on replicating one error behavior. A system that could be used by election officials to identify bad ballots would have to be able to predict all historical voting errors, as well as any new ones.

Thus, Wang, Lindstedt, and Byrne (2019) describe the construction of a model that will simulate a vast array of possible voting errors, with an eventual goal of simulating the entire voting space. In their paper they describe the start of building such a model. The model ran in a voting environment called VoteBox. It was a simulated simple electronic ballot, consisting of just a single race per screen with a next button to navigate.

Nevertheless, within just a simple task was hidden great complexity: Wang describes building and using a total of 40 different voting strategies constructed from differing memory and navigational strategy selections. The voters differing strategies and knowledge led to different rates of error, showing that a voter’s strategy made a difference to whether they were able to vote for their intended candidates. However, this effort did not vary the structure of the ballot and so did not yet have the predictive power necessary to recognize badly designed ballots.

Thus, in this paper, we first describe the extension of this system to handle full simulated paper ballots. Then, we describe the error rates of our new simulated voters on various simulated ballots. This represents a large step towards our end goal of constructing a piece of software to identify bad ballots.

# Method

First, we describe the design of our experiment: the randomly generated full length ballots and our new modular system that votes on them.

## Ballot Design

We built simulated full “paper ballots” for the model which consist of a virtual screen populated with several columns of races. Each race has a title, a list of candidates and their associated parties, and a list of buttons that the model can click to vote for a candidate. (see Figure 1).



Figure 1: Part (top left corner) of a simulated ballot

The resulting simulation is not quite the same as a paper ballot: the model clicks on a button instead of filling in a circle, will never obscure the ballot with its hand, and will never poke a hole in the page among many other differences. However, the ballot is similar enough to cause many of the same errors we expect humans to make.

To help the model navigate, we colored the race header red, the candidates purple, and the parties blue, as ACT-R can make visual location requests based off color.

## Model Design

We built the model itself with one overarching goal in mind: to have the ability to simulate as wide an array of voters as possible.

Our modular system split a simulated voter’s strategy into four different pieces: macronavigation, the process of moving from one race to the next; encoding, the process of determining the race, party, and candidate visual groups for each race; micronavigation, the process of finding the intended candidate to vote for within each race; and clicking, the process of actually clicking on the button corresponding to the chosen candidate. At runtime we select one strategy from each of these categories and combine them together with a declarative memory file to build an ACT-R model. Note that Wang Et Al’s work remains in our architecture as possible micronavigation and declarative memory choices.

## Designing New Strategies

We first built the most obvious strategies for each strategy category because we wanted our initial strategies to lead to a composite voting strategy with no errors. We had to know that our model worked before we could start varying pieces to induce errors.

Our first new strategy was a non-standard macronavigation strategy. Our model’s standard macronavigation strategy was *top to bottom left to right*; that is, the model started in the top left corner and went all the way from the top to the bottom and then went over to the next column to the right and again went top to bottom, repeating until it was finished. This is the most obvious method of macronavigation, and like noted above resulted in no mistaken votes. On the other hand, the first new macronavigation strategy we built was *left to right top to bottom*. This strategy started on the first row of races and proceeded left to right, finishing each row and then continuing to the next one down.

Below is each pattern of movement, where the green arrows represent moving to the next race, the blue arrows a backtrack to the start of a column or row, and the red arrows movement between rows.

Now that we had a working voter, we sought to add an additional macronavigation strategy. Although we could choose any of the 40 possible micronavigation strategies Wang had generated, we decided to hold the micronavigation constant with a so called perfect strategy, one that had a perfect memory of who to vote for and so would never vote for the wrong person. Because of the modular nature of our model, this would ensure that any errors were a result of our macronavigation procedures. We want our final product to be able to simulate a vast array of possible voters, but for development it is useful to hold many of the variables constant

.

## Figures

