Summer A 2014

Table of Contents

Software for Analyzing and Generating Fixed Expanding Retrieval Schedules ..................... 2-16
Nicole Mirea, Faculty Mentor: Phil Kellman, Ph.D.

**These research papers are written by undergraduate students as part of the capstone requirement for the Cognitive Science major.**
An expanding retrieval schedule is a schedule of retrieval practice in which the spacing between each subsequent presentation of a given item increases by a predetermined factor. Although many other researchers have studied these types of fixed schedules, we had previously been unable to find a suitable algorithm for automatically generating them. In past research by the lab, these schedules were generated by hand—a tedious and regrettably often imprecise process. The primary challenge in designing such an algorithm was maintaining a constant rate of spacing expansion. To illustrate, let’s suppose we wish to increase the spacing between presentations of an item by two with each presentation, with an initial spacing of one item between each presentation. The beginning of the sequence would look something like this: 1, 2, 1, 2, 3, 4. But what comes next? Both 1 and 3 are viable candidates.

Initially, I thought of solving this problem through the use of filler items. For every “spot” in the entire schedule, the following JavaScript function runs through all available items to find the one whose presentation would be ideal. If none of the items fit in the current spot, the function adds a filler item instead.

```javascript
function makeSequence(items, expansion, maxlength) {

    // generate and shuffle list of items
    var item_pool = shuffle(_.range(1, items+1));

    // minimum spacing between two consecutive numbers
    var min_interval = 1;
```
var first_interval = min_interval + 1;

// leeway factor permitted by the algorithm
var wiggle = 4;

// text to use for filler in display
var filler_text = "F";

var this_interval = 0, prev_interval = 0, add_item = 0, prev_instances = 0;
var test = true;
var test_seqlength = false;

var seq = [];
for(var i = 0; i < maxlength; i++) { //noprotect
  // pick first
  if (seq.length === 0) {
    add_item = item_pool[getRandomInt(0,item_pool.length-1)];
    test = true;
  }
  else {
    for (var j = 0; j < item_pool.length; j++) {
      add_item = item_pool[j];
      test = true;
    }
  }
  //pick first
  if (seq.length === 0) {
    add_item = item_pool[getRandomInt(0,item_pool.length-1)];
    test = true;
  }
  else {
    for (var j = 0; j < item_pool.length; j++) {
      add_item = item_pool[j];
      test = true;
    }
  }
}
prev_instances = getInstances(add_item, seq);

if (prev_instances.length > 0) {

    // current interval: difference between current index and previous appearance of add_item
    this_interval = i - prev_instances[prev_instances.length-1];

    if (prev_instances.length > 1) {

        // calculate length of previous interval
        prev_interval = prev_instances[prev_instances.length-1] - prev_instances[prev_instances.length-2];

        // test whether current interval is long enough (but not longer than the 'wiggle' factor allows)
        if (this_interval <= (prev_interval + expansion) || this_interval >= ((prev_interval + expansion) + wiggle))
            test = false;

    }

    // tests whether first interval meets minimum interval length requirements
    else if (prev_instances.length === 1 && (this_interval < first_interval || this_interval >= (first_interval + wiggle))) {
        test = false;
    }
}

if (test === true) {

}
break;

}

}

// no items from the pool match the conditions
if (test === false)
    add_item = filler_text;

}

// add item to sequence
seq.push(add_item);

}

// replace all "one-shots" with filler
for (var n = 0; n < seq.length; n++) {
    if (getInstances(seq[n], seq).length == 1)
        seq[n] = filler_text;

}

return seq;

This algorithm relies heavily on filler items. However, for the purposes of our laboratory’s research, it is not always advisable to use filler items; in some experiments, their introduction would pose a confound. Furthermore, some items were never introduced using this algorithm. To resolve these problems, we settled on the function below, which assigns each item a priority score based on the difference from the expected spacing and the actual number of items between the current “slot” and the previous occurrence of the item. The item with the highest priority score is then picked to appear next. (If two items have identical priority scores, one is picked at random from those.)

```javascript
function makeSequence(numitems,numtrials,ppi){

    /// Params ///

    //number of items
```
var n = _.range(numitems);
//number of trials
var t = _.range(1,numtrials);
//length of item schedule
var l = _.range(ppi);
//trial spacing
var s = l.map(function(guy){
    return (guy-1)*4 + 5;
});
console.log(s);

///// Calc /////

var itemHist = n.map(function(){
    return [];
});

//calculate each trial
var sched = t.map(function(trial){
    //find highest priority item
    //console.log("trial: "+trial);
    //score = abs|trials since last presentation minus expected spacing|
    var score = n.map(function(item){
        var len = itemHist[item].length;
        var len = itemHist[item].length;
if (len!==0){
    if (len == 1 && trial-itemHist[item].slice(-1)[0]-s[len-1] == 1){
        return 100;
    } else if (len == 2 && trial-itemHist[item].slice(-1)[0]-s[len-1] == 5){
        return 90;
    } else {
        return trial-itemHist[item].slice(-1)[0]-s[len-1];
    }
}

} else {
    return 0.5;
}

});

//console.log('trial: '+trial+' score: ['+score.join(',')+']');
var min = _.max(score);
var instances = score.reduce(function(a, e, i) {
    if (e === min){
        a.push(i);
    }
    return a;
}, []);

//console.log('instances: ['+instances.join(',')+']');
var item = 0;
item = _.shuffle(instances)[0];
itemHist[item].push(trial);
return item;
//}
});

return sched;
}

Using the same parameters as in the previous example yields a vastly different output: 1, 2, 1, 2, 3, 4, 3, 4, 1, 2, 5, 6, 5, 6, 3, 4, 7, 8, 7, 8, 5, 1, 2, 6, 3, 7, 8, 4, 9, 10, 9, 10, 5, 6, 11, 1, 11, 7, 8, 2, 9, 10, 3, 4, 11, 12, 5, 12, 6, 13, 9, 13, 7, 10, 8, 12, 1, 11, 2, 13, 3, 4, 14, 15, 14, 15, 9, 5, 12, 6, 10, 13, 7, 14, 11, 15, 8, 16, 1, 16, 2, 17, 12, 17, 3, 14, 4, 9, 15, 16, 13, 10, 17, 5, 6, 11, 7, 18, 8, 18, 16, 14, 12, 15, 17, 1, 18, 2, 13, 9, 3, 19, 4, 19, 10, 16, 18, 11, 17, 14, 5, 19, 6, 15, 7, 8, 12, 20, 21, 20, 21, 18, 13, 19, 16, 9, 1, 20, 21, 17, 2, 3, 10, 14, 4, 11, 15, 20, 19, 21, 18, 5, 6, 12, 7, 8, 16, 22, 13, 22, 23, 20, 23, 17, 21, 22, 9, 19, 23, 14, 1, 24, 15, 24, 18, 10, 2, 3, 22, 11, 23, 4, 24, 20, 16, 21, 12, 5, 6, 13, 7, 17, 19, 8, 22, 24, 23, 25, 26, 25, 26, 14, 18, 9, 15, 20, 25, 26, 21, 1, 24, 10, 22, 11, 16, 23, 2, 3, 25, 26, 19, 4, 12, 17, 13, 5, 6, 27, 24, 27, 7, 8, 20, 25, 18, 26, 22, 21, 14.

Here, all items are introduced (provided the schedule is long enough), and no filler items are required. The only limitation of this algorithm is that the spacing delays cannot reliably expand once all items are introduced, but this is more a general mathematical property of expanding sequences rather than a failure of the algorithm.

Eventually, I also translated the function above into another programming language, PHP, while modifying it to return only the next item in the sequence, given a list of all items that have been presented so far and a set of available items. This is so that items may be eliminated
from the available set once the learner has reached a certain criterion (for example, answered the item correctly three times in a row). The source code for this version is reproduced below:

```php
function next_item($history, $unretired) {
    $result = NULL;
    // don't return anything if all items retired
    if (sizeof($unretired) > 0) {
        // create list of priority scores
        $scores = array();
        foreach ($unretired as $itemid) {
            $scores[$itemid] = score($history,$itemid);
        }
    }
}
```
// find highest score, create array of items w/ that score

$hiscore = max($scores);

$instances = array_keys($scores,$hiscore);

$randomind = mt_rand(0,count($instances) - 1);

$result = $instances[$randomind];

} /*

Calculates priority score for a given item.

@param $history is a list of item identifiers presented thus far
@param $item is the identifier corresponding to the current item
@return a priority score that determines how soon the item should be presented */

function score($history, $item) {
    $result = 0.5;

    $itemhist = array_keys($history, $item);
    $len = sizeof($itemhist);
    $trial = sizeof($history);
    $delaydiff = $trial - end($itemhist) - calc_previous_delay($len - 1);
if ($len > 0) {

    if ($len == 1 && ($delaydiff == 1))
        $result = 100;
    elseif ($len == 2 && ($delaydiff == 5))
        $result = 90;
    else
        $result = $delaydiff;

}

return $result;
}

/**************************************************************************
* Calculates expected spacing between the (n+1)th and nth presentation of an item.
**************************************************************************/

function calc_previous_delay($n) {
    $expansion = 4; // factor by which the sequence expands

    return ($n-1)*$expansion + $expansion + 1;
}
To facilitate the analysis of these sequences—check that all items are introduced, easily calculate the spacing delays between items, etc.—I also created a JavaScript tool that calculates the number of presentations of each item, the pattern of spacing delays, and the average delay for each item in a schedule from a list of every item in the schedule. This program also calculates averages and standard deviations of the previous measures across items. Output is both in the form of a downloadable CSV file (for use in Excel and other spreadsheet software) and a neatly-formatted table, an example of which is printed on the page following the source code for this function below.

```javascript
function (seq_text, delimiter) {
    var seq = seq_text.split(delimiter);
    var uniques = getUniques(seq).naturalSort();
    var delays = [], prescounts = [], avgdelays = [];
    var all_info = [['Value', 'Presentation Count', 'Delay Pattern', 'Average Delay']];
    var totals = [['Averages'], ['Standard Deviations']];

    for (var i = 0; i < uniques.length; i++) {
        delays.push(getDelays(uniques[i], seq));
    }

    return $result;
}
```
prescounts.push(getInstances(uniques[i],seq).length);

avgdelays.push(average(delays[i]));

all_info.push(
    uniques[i],
    prescounts[i],
    delays[i],
    avgdelays[i]
);

// total averages
totals[0].push(average(prescounts));
totals[0].push(across_arrays(delays,average));
totals[0].push(average(avgdelays));

// total standard deviations
totals[1].push(stdev(prescounts));
totals[1].push(across_arrays(delays,stdev));
totals[1].push(stdev(avgdelays));

// putting it all together
all_info = all_info.concat(totals);

// output time

"#output div#results".html(arrays_to_table(all_info));
// generate csv link

csv_link = document.createElement("a");
csv_link.setAttribute("href", arrays_to_csv(all_info));
csv_link.setAttribute("download", "my_data.csv");
$("#output div#results").prepend(csv_link);
$("#output div#results a").html("Download CSV");
**Figure.** Interface of Expanding Sequence Analyzer. A short sequence is shown for illustrative purposes. Editable fields have a light blue background.