Final Project

Benjamin Sorenson

December 17, 2015

Abstract

As a matter of test security, the state of Pennsylvania has required the use of scrambled test forms for its high-stakes K-12 assessment programs, Pennsylvania System of School Assessment (PSSA) and Keystone. Given the constraints imposed on the scrambling process, the task has been framed as CSP, and two algorithms'(SCRAMBER and backtracking search) performance are compared on the grounds of time to a solution and probability of finding a solution.

1 Introduction

The construction of scrambled test forms is a fairly common practice and is conducted to deter would-be cheaters, and to thwart unskilled cheaters. Conceptually this is a simple task—construct a test and repeatedly shuffle the test questions until you have the desired number of forms. However, in the world of high-stakes, large-scale assessment, it's not so simple. Tests are constructed using strict guidelines dictating everything from the overall difficulty of the test, where difficult test questions can be placed, and the order the content is encountered. If one were to simply shuffle a test form constructed to meet a particular blueprint, it's likely that the scrambled version of the test would violate one or more of the requirements of the blueprint.

Again, this is the world of high-stakes testing—by definition, there are potentially life-altering stakes attached. These stakes can range from the determination of high school graduation eligibility to federal teacher accountability. For this reason it is important that each scrambled version of a test is equivalent to the unscrambled version constructed in accordance with the test blueprint, and that is precisely the aim of the problem I chose for my final project.

It is simple to scramble a single test form in order to come up with one additional form, but it is gets more complex as the number of desired forms increases. Typically, when human beings try this, they will first construct the best scrambled test form they can. This makes for a good set of two forms, but may make it difficult to construct more scrambled forms. As more forms are constructed the number of comparisons increases in complexity at rate of $O(n^2)$ because each form has to be compared to every other form. It doesn't take many forms before these pair-wise comparison become tedious for human beings.

This still wouldn't be too much of a problem if there was just one test to administer, but there are often multiple grades and subjects under a single testing program, and soon the problem of test scrambling, if left to humans could take weeks. It was for this reason that I was asked to write a program to solve the problem scrambling test forms with constraints. After having taken this class, I think the current solution could be re-framed as a constraint satisfaction problem, and improved both in terms of the probability of finding a solution and running time.

2 Background

As a matter of test security, Pennsylvania requires that both its summative Pennsylvania System of School Assessment (PSSA) and graduation requirement Keystone exams are scrambled. This practice was put into place to prevent both student to student cheating and as a deterrent to administrator manipulation of answer documents (e.g., erasing and correcting answers after the fact). The idea was presented to a technical advisory committee (TAC) consisting of educational measurement experts. They suggested that in order to minimize possible item position effects (the phenomenon that test questions can become more or less difficult depending on their position in the exam), tests be broken up into blocks of six or seven items consecutive multiple choice items, and scrambling be done within these blocks according to the following guidelines from the 2014 PSSA Technical Report [1][100-101]

- Items cannot move between blocks.
- DRC and PDE content specialists will work to ensure that the scrambling does not result in making content more difficult than the Master Core item sequence. For example, items of similar cognitive complexity will be swapped rather than random

scrambling.

- A block scramble pattern is only valid if it does not contain an invalid key distribution within the block. Additional checks for an invalid key distribution across blocks must be made when combining block scramble patterns to create forms. For example, scrambling must not create more than three (3) of the same key positions in a row.
- A block scramble pattern is only valid if it does not contain an invalid standard (AA/EC) distribution within a block. Additional checks for standard distribution across blocks must be made when combining block scramble patterns to create forms. An exception was made for one mathematics scramble for each grade which ordered items within block by eligible content per PDE request. Scrambling should not place a difficult item as the first item in a section. The first item in a block that does NOT begin a section may be a difficult item since blocks are invisible to the student.
- For passage-based items, a block scramble pattern is only valid if it does not create dissonance between the items and passage(s).
- Scrambling should not place a difficult item as the first item in a passage set.
- Within a set of items connected to a paired set of passages, an item associated with both passages can be swapped only with another item associated with both passages. (These items must remain at the end of the set of items associated with the passage set.)

Following the guidelines above, test development specialists, divide the Master Core form into 4-9 blocks, and each block is then scrambled into 4 unique scramble patterns. These blocks are then assembled using a software I authored. Some constraints regarding block scramble assembly are handled within the construction of scrambled blocks. For example, the constraint that "Additional checks for standard distribution across blocks must be made when combing block scramble patterns to create forms.", is solved during construction by purposely constructing block scrambles so that no bordering blocks create an invalid standard distribution.

In addition to the above mentioned constraints, there are some practical considerations that further constrain the combination of block scrambles. For example, it is desirable for the purposes of quality assurance that the misapplication of one scrambled answer key for another is easily detectable, and that an unskilled cheater couldn't copy off of a different version of the test, and still get anything but the lowest achievement level possible (this is the equivalent of an "F", but is called "Below Basic" in the Pennsylvania system). For these reasons, the percentage of answer-key overlap between any pair of forms is restricted to something less than the score required for an achievement level above "Below Basic". And, in anticipation that wasting (i.e., neglecting to use) or overusing a block scramble pattern might be undesirable, a further constraint was added to ensure each scramble pattern was used at least once, but not more than a user-specified number of times.

3 Approach

The current approach to solving this problem is by randomized depth-first with back jumping. The problem is represented a matrix where each row in the matrix matrix represents a scrambled form, and each column represents a block number. The value in each cell represents a scramble pattern. For example the matrix

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 2 \\ 2 & 1 & 1 \end{bmatrix}$$

would represent a solution to problem in which there were 2 forms needed in addition to the Master Core, and $a_{21} = 1$ means that form-1 used scramble pattern 1 for block 1. Zeros indicate the use of a Master Core (unscrambled) block. For instance $a_{22} = 0$ indicates that block-2 of form-1 is an unscrambled block. In the current approach, SCRAMBLER (Algorithm- 1), the matrix is built up row-by-row—the initial state is always the Master Core, and every time an assignment is made to a new form that satisfies the given constraints, it is added to the matrix. Once the matrix contains the desired number of rows, the solution is returned.

In this representation, each form is a variable, and each permutation block scrambles is a possible assignment. This means that both space requirement to store the domains for each variable and the cost of constructing the domains is exponential in the number of block scrambles. SCRAMBLER avoids this by randomly generating possible assignments as needed.

Algorithm 1 The SCRAMBLER algorithm

```
function SCRAMBLER($max\_restarts$)
    $more\_steps\_back \leftarrow 0$
    $restarts \leftarrow 0$
    $assignment \leftarrow [(0, 0, \dots, 0)]$
    $forced\_choice \leftarrow 0$                ▷ $forced\_choice$ is set by GETELIGIBLEBLOCKSCRAMBLES
    while $restarts < max\_restarts$ do
        for $i = 1$ to $max\_iterations$ do
            $eligible\_block\_scrambles$ ← GETELIGIBLEBLOCKSCRAMBLES($assignment, forced\_choice$)
            if there are no choices for one or more blocks then break
            else
                $scramble\_pattern \leftarrow$ random choice from each eligible block
            end if
            if $scramble\_pattern$ satisfies constraints then
                add $scramble\_pattern$ to $assignment$
            end if
            if all forms assinged then
                return $assignment$
            end if
        end for
        if $forced\_choice > 0$ then
            if $forced\_choice - more\_steps\_back > 1$ then
                remove all but the first $forced\_choice - more\_steps\_back$ forms from $assignment$
                $more\_steps\_back = more\_steps\_back + 1$
            else
                $assignment \leftarrow [(0, 0, \dots, 0)]$
                $restarts = restarts + 1$
                $more\_steps\_back = 0$
            end if
            $forced\_choice = 0$
        else if $assingment.length - more\_steps\_back > 1$ then
            remove all but the first $assignment.length - more\_steps\_back - 1$ forms from $assignment$
            $more\_steps\_back = more\_steps\_back + 1$
        else
            $assignment \leftarrow [(0, 0, \dots, 0)]$
            $restarts = restarts + 1$
            $more\_steps\_back = 0$
        end if
    end while
    return None
end function
```

The idea behind SCRAMBLER is that at the shallower depths, it is easy to find many assignments that will likely satisfy the constraints, but it is also easy to unwittingly paint yourself into a corner along the way. It's difficult to know for sure which variable assignment caused you to fail, but it's not worth the effort to keep checking each possible variable at the current depth and backtrack one level. In this representation of the problem, for a number of desired forms (variables) n_f, number of blocks b, and number of scramble patterns per block s (assuming a equal number of scramble patterns per block), the branching factor is s^b, and the depth of the solution is n_f. So, rather, than search through each s^b at the current depth for the next assignment in the solution solution, SCRAMBLER lets the user set the number of iterations before the algorithm gives up and decides that it's *probably* at a dead-end. Once it has determined that it's likely at a dead-end, it first backs up one level, but it also keeps track of how many times it's had to back up—each time, it backs up one more level than the previous time. Once it has backed up all the way to the root, this is counted as a restart. After a user-specified maximum number of restarts, the algorithm can either return **None** or the current partial assignment—for this project, it was set to return **None**.

While it is not possible to know for sure what assignment likely caused it to run into a dead-end, the function GETELIGIBLEBLOCKSCRAMBLES employs a simple heuristic to help determine how far we should backtrack. When it is called on the current assignment, it counts the number of block scrambles that are still available to use, because of the constraint that all scrambles must be used at least once, it applies the pigeon hole principle to determine if there are any blocks that *must* be used in the remaining variable assignment(s). If there are any, it only returns those as the eligible or legal choices for the next assignment(s), and indicates where this first occurred. If no solution is found, then SCRAMBLER backs up to the point marked by GETELIGIBLESCRAMBLES plus any additional levels indicated by *more_steps_back*.

The purpose of this project was to improve on the current algorithm by employing backtracking search. I chose to use the software aima-python[2]. I changed the representation of the problem so that each (form, block) pair is a single variable and the domain for each is the block scrambles available for the (form, block) combination. For example, the variable (0, 1) is the Master Core form, block-1, and thus has only the value 0 in its domain. The variable (1, 1), however is for form-1, block-1, and thus has the domain [0, 1, 2, 3, 4] (0 for the Master Core, and the integers 1-4 for each of the scrambled version of block-1). The CSP framework in aima-python requires that the variable *neighbors*) is defined. In some sense every variable is a neighbor with every other variable since there are global constraints that apply to the whole assignment, but after some experimentation, neighbors were defined as those (form, block) pairs whose answer keys share a border. For example, for a form f_k, the variable (f_k, b_2) shares a border with (f_k, b_1) and (f_k, b_3).

Based on some preliminary experimentation, I found that the MRV and LCV heuristics as implemented in aima-python only slowed down the search so I implemented a least-used least-overlapping value-ordering heuristic (LULOV) that ordered values by a combination of their usage (less frequently used values in the block were given preference), and their key overlap with other values in the same block in the current assignment (lower overlapping values were given preference). The final ordering was then determined by the sum of the variable's rank by usage and rank by overlap so that equal weight was given to the value's usage and answer-key overlap within the current assignment. Because backtracking search always returns the same solution for the same set of inputs and initial state, and to introduce some randomness, I also experimented with randomly shuffling values.

4 Experimental Design

In this project, I wanted to answer the question "Can the current scrambling algorithm be improved by using backtracking search?" To test the two algorithms, I chose a set of some of the more difficult real-world data, and ran each algorithm under varying conditions.

- The total number of forms was varied between 5 and 20 stepping by 3
- The total amount allowable answer-key overlap varied between 25% and 50% of the total answer-key string stepping by 5%
- For each of the above conditions, backtracking search was run with variable shuffling and the LULOV heuristic —once with neither, once with variable shuffling only, once with LULOV only, and once with both shuffling and LULOV.
- Each algorithm was given maximum of 240 seconds to find a solution.

Each combination of conditions mentioned above was run 4 times, for a total of $4 \times 6 \times 6 \times 4 \times 2 =$ 1152 total runs constituting 4 replications for each combination of conditions for the backtracking search, and 16 replications of each condition for SCRAMBLER as shuffling and LULOV only affect the performance of backtracking search. Due to the randomized approach of SCRAMBLER its performance is best summarized by expected running time over a series of replications since it chooses values randomly.

The default settings were used for SCRAMBER in all cases as these are rarely adjusted in practice:

- Maximum iterations/attempts before backtracking (*max_its*) was set to 1000
- Maximum numer of restarts (*mas_restarts*) before failing was set to 50.

Results were then summarized and analyzed using a combination of NumPy [5], Pandas [**mckinney-proc-scipy-20** and statsmodels [4], using a the time taken to find a solution and probability of finding a solution as a performance metric.

5 Results

Tables 1-3 summarize the results of the experiment. Please note that the tables only present the results as backtracking search with and without LULOV and SCRAMBLER for each of the form and overlap conditions—value shuffling had no noticeable effect on either the probability of success or the amount time to find a solution so those conditions were collapsed into the conditions using backtracking and backtracking with LULOV.

For the most part, backtracking search with LULOV, and backtracking search without LULOV found solutions and failed to find solutions under the same conditions. There were a few exceptions, however. Most notably, in Table 2 and Table 3, we see that when there were 5 forms both backtracking with LULOV and backtracking without LULOV found solutions on each of the eight replications when the allowable overlap was greater than chance (25%), and both took a similar amount of time to find a solution. However, when using LULOV, backtracking search was able to find a solution to 2 of the 8 replications with 5 forms and a total key overlap no greater than chance. With only 8 replications it's difficult to say whether this was luck or due to the heuristic itself.

Observing the results from other conditions in Table 2 and Table 3, we see that LULOV had little to no observable effect. For example, with 8 forms, backtracking search failed both with LULOV and without, and both were able to find a solution when the overlap was less than 40%. At an overlap

		Solution Found	Total Time (s)				
N-Forms	P-Overlap	P-Success	N-Rep.	Mean	Min.	Max.	Std. Dev.
5	0.25	0.50	16	15.94	2.18	26.96	8.91
	0.30	1.00	16	0.56	0.00	3.88	0.99
	0.35	1.00	16	0.19	0.00	1.61	0.45
	0.40	1.00	16	0.06	0.00	0.97	0.24
	0.45	1.00	16	0.20	0.00	0.83	0.28
	0.50	1.00	16	0.22	0.00	1.23	0.39
8	0.25	0.00	16	78.60	43.98	93.37	19.64
	0.30	0.00	16	118.54	62.10	132.78	22.28
	0.35	1.00	16	1.13	0.06	5.15	1.35
	0.40	1.00	16	0.23	0.01	1.27	0.44
	0.45	1.00	16	0.12	0.01	1.79	0.45
	0.50	1.00	16	0.07	0.01	0.51	0.17
11	0.25	0.00	16	88.02	81.77	97.74	4.10
	0.30	0.00	16	114.09	107.94	125.77	4.98
	0.35	0.00	16	162.70	152.71	170.28	5.19
	0.40	1.00	16	0.91	0.15	4.65	1.08
	0.45	1.00	16	0.09	0.03	0.24	0.05
	0.50	1.00	16	0.04	0.02	0.08	0.02
14	0.25	0.00	16	84.17	79.04	89.22	2.55
	0.30	0.00	16	114.12	106.48	122.46	3.90
	0.35	0.00	16	161.89	155.45	168.40	3.86
	0.40	0.00	16	240.00	240.00	240.00	0.00
	0.45	1.00	16	1.13	0.09	2.74	0.91
	0.50	1.00	16	0.47	0.03	1.66	0.61
17	0.25	0.00	16	84.36	81.30	90.54	2.48
	0.30	0.00	16	118.22	112.19	125.47	4.54
	0.35	0.00	16	171.89	156.47	240.00	18.89
	0.40	0.00	16	242.97	240.00	251.32	4.18
	0.45	1.00	16	35.76	1.51	86.20	29.44
	0.50	1.00	16	2.28	0.24	8.69	2.39
20	0.25	0.00	16	86.28	82.21	92.09	3.13
	0.30	0.00	16	116.27	111.47	122.35	2.99
	0.35	0.00	16	164.82	159.52	169.72	2.70
	0.40	0.00	16	241.38	240.00	250.92	3.23
	0.45	0.00	16	240.00	240.00	240.00	0.00
	0.50	0.00	16	240.00	240.00	240.00	0.00

Table 1: Scrambler

		Solution Found	Total Time (s)				
N-Forms	P-Overlap	P-Success	N-Rep.	Mean	Min.	Max.	Std. Dev.
5	0.25	0.00	8	240.00	240.00	240.00	0.00
	0.30	1.00	8	1.02	0.10	7.27	2.52
	0.35	1.00	8	0.13	0.09	0.20	0.04
	0.40	1.00	8	0.11	0.08	0.11	0.01
	0.45	1.00	8	0.10	0.07	0.12	0.01
	0.50	1.00	8	0.11	0.09	0.13	0.01
8	0.25	0.00	8	240.00	240.00	240.00	0.00
	0.30	0.00	8	240.00	240.00	240.00	0.00
	0.35	0.00	8	240.00	240.00	240.00	0.00
	0.40	1.00	8	1.80	0.57	5.61	2.08
	0.45	1.00	8	0.44	0.38	0.53	0.06
	0.50	1.00	8	0.43	0.39	0.45	0.02
11	0.25	0.00	8	240.00	240.00	240.00	0.00
	0.30	0.00	8	240.00	240.00	240.00	0.00
	0.35	0.00	8	240.00	240.00	240.00	0.00
	0.40	1.00	8	6.61	3.71	11.73	3.64
	0.45	1.00	8	1.30	1.03	1.64	0.27
	0.50	1.00	8	1.05	0.85	1.20	0.12
14	0.25	0.00	8	240.00	240.00	240.00	0.00
	0.30	0.00	8	240.00	240.00	240.00	0.00
	0.35	0.00	8	240.00	240.00	240.00	0.00
	0.40	0.00	8	240.00	240.00	240.00	0.00
	0.45	1.00	8	2.83	1.90	3.87	0.60
	0.50	1.00	8	2.41	1.81	2.92	0.42
17	0.25	0.00	8	240.00	240.00	240.00	0.00
	0.30	0.00	8	240.00	240.00	240.00	0.00
	0.35	0.00	8	240.00	240.00	240.00	0.00
	0.40	0.00	8	240.00	240.00	240.00	0.00
	0.45	1.00	8	6.62	5.58	8.19	0.90
	0.50	1.00	8	5.84	4.09	6.75	0.96
20	0.25	0.00	8	240.00	240.00	240.00	0.00
	0.30	0.00	8	240.00	240.00	240.00	0.00
	0.35	0.00	8	240.00	240.00	240.00	0.00
	0.40	0.00	8	240.00	240.00	240.00	0.00
	0.45	0.00	8	240.00	240.00	240.00	0.00
	0.50	0.12	8	231.51	172.05	240.00	24.02

Table 2: Backtracking with no Heuristic

		Solution Found	Total Time (s)				
N-Forms	P-Overlap	P-Success	N-Rep.	Mean	Min.	Max.	Std. Dev.
5	0.25	0.25	8	182.04	7.34	240.00	107.31
	0.30	1.00	8	1.68	0.07	12.59	4.41
	0.35	1.00	8	0.11	0.08	0.15	0.02
	0.40	1.00	8	0.11	0.07	0.16	0.03
	0.45	1.00	8	0.11	0.08	0.13	0.02
	0.50	1.00	8	0.11	0.08	0.12	0.01
8	0.25	0.00	8	240.00	240.00	240.00	0.00
	0.30	0.00	8	240.00	240.00	240.00	0.00
	0.35	0.00	8	240.00	240.00	240.00	0.00
	0.40	1.00	8	1.26	0.54	2.74	0.97
	0.45	1.00	8	0.46	0.37	0.71	0.11
	0.50	1.00	8	0.45	0.37	0.54	0.05
11	0.25	0.00	8	240.00	240.00	240.00	0.00
	0.30	0.00	8	240.00	240.00	240.00	0.00
	0.35	0.00	8	240.00	240.00	240.00	0.00
	0.40	1.00	8	4.87	2.30	10.48	2.46
	0.45	1.00	8	1.51	1.11	1.80	0.23
	0.50	1.00	8	1.07	0.90	1.27	0.13
14	0.25	0.00	8	240.00	240.00	240.00	0.00
	0.30	0.00	8	240.00	240.00	240.00	0.00
	0.35	0.00	8	240.00	240.00	240.00	0.00
	0.40	0.00	8	240.00	240.00	240.00	0.00
	0.45	1.00	8	2.67	2.08	3.21	0.39
	0.50	1.00	8	2.39	1.91	2.79	0.38
17	0.25	0.00	8	240.00	240.00	240.00	0.00
	0.30	0.00	8	240.00	240.00	240.00	0.00
	0.35	0.00	8	240.00	240.00	240.00	0.00
	0.40	0.00	8	240.00	240.00	240.00	0.00
	0.45	1.00	8	6.62	4.47	8.63	1.19
	0.50	1.00	8	5.50	4.25	6.42	0.90
20	0.25	0.00	8	240.00	240.00	240.00	0.00
	0.30	0.00	8	240.00	240.00	240.00	0.00
	0.35	0.00	8	240.00	240.00	240.00	0.00
	0.40	0.00	8	240.00	240.00	240.00	0.00
	0.45	0.00	8	240.00	240.00	240.00	0.00
	0.50	0.12	8	230.45	163.58	240.00	27.02

Table 3: Backtracking with LULOV

of 40%, backtracking with LULOV had a lower average running time than backtracking without, but again, with only eight replications, it's doubtful that these results will generalize. Considering also that the non-LULOV version had a maximum running time of 5.61s, the hypothesis that the heuristic made any significant impact to the performance is doubtful.

Under the condition when 11 were forms required, backtracking search with LULOV shows the greatest difference in terms of running time compared to backtracking search with no heuristic. Backtracking with LULOV beat backtracking search by more than two seconds on average, had a smaller standard deviation, lower minimum and lower maximum than the non-heuristic version for the case when 40% overlap was allowed. However, we see the oposite is true when we allow 45% overlap.

When we compare the results of SCRAMBLER (Table 1) to the results of backtracking search (both with and without LULOV), we see that in almost every condition where backtracking search was able to find a solution, SCRAMBLER also finds a solution. And, in the condition with 8 forms and a maximum overlap of 35%, SCRAMBLER was able to find a solution in all 16 replications. By contrast, backtracking search (again, either with or without LULOV) was not able to find one in any of the 16 replications. There was one instance (20 forms with 50% answer-key overlap) where backtracking search was able to find a solution, and SCRAMLER was not.

Based on this informal analysis, it appears that SCRAMBLER is the superior algorithm under the conditions tested, and that backtracking search with LULOV constitutes a minimal improvement, if any, over backtracking with no heuristic. To isolate the effect each algorithm and heuristic choice had on the probability of finding a solution, a logistic regression was run using the Python [3] package statsmodels [4] using the indicator variable $mathitsolution\_found$ as the dependent variable, and regressing on the number of forms, percent allowable overlap, the algorithm used, and the interaction between the algorithm used and the use of LULOV. From table 4 we see that all else equal, choosing SCRAMBLER over backtracking was significant ($p = .02$), and had a positive coefficient of 2.37 which translates to 2.79 times greater (see table 5) odds of finding a solution under the conditions tested when compared to backtracking search, all else being equal. Though it's not statistically significant, it is worth noting that the coefficient on that the interaction of backtracking search with a LULOV has a positive coefficient, corresponding to a 23% increase in the odds of finding a solution, all else being equal. Because it's far from statistically significant

	Coef.	Std.Err.	z	$P > \|z\|$	[0.025	0.975]
Intercept	-17.48	1.57	-11.11	0.00	-20.56	-14.39
Scrambler	1.03	0.44	2.33	0.02	0.16	1.89
BT:LULOV	0.20	0.44	0.44	0.66	-0.67	1.06
Scrambler:LULOV	-0.00	0.42	-0.00	1.00	-0.82	0.82
% Overlap	83.06	6.94	11.98	0.00	69.47	96.65
N-Forms	-1.25	0.11	-11.94	0.00	-1.46	-1.05

Table 4: Logistics regression results

	Coefs.	[0.025	0.975]
Intercept	0.00	0.00	0.00
Scrambler	2.79	1.18	6.64
BT:LULOV	1.22	0.51	2.90
Scrambler:LULOV	1.00	0.44	2.26
% Overlap	1.2e+36	1.5e+30	9.5e+41
N-Forms	0.29	0.23	0.35

Table 5: exp Coefs. from logistic regression

($p = 0.66$), it's entirely possible that this is just noise, and says nothing about the quality of the heuristic.

From tables 1-3, we can see that in all cases where both Scrambler and backtracking search were able to find a solution, Scrambler was able to find it in less time. To isolate the effect the choice of algorithm and heuristic had on the time to find a solution, an ordinary least squares (OLS) regression was run with the same regressors as before, but this time with $log(mathittime)$ as the dependent variable, and unsuccessful observations removed. Once again, the choice of algorithm was significant, but the interaction between the choice of algorithm and the use of the LULOV heuristic was not. Table 6. summarizes the results of the regression, and table shows the impact

	Coef.	Std.Err.	t	$P > \|t\|$	[0.025	0.975]
Intercept	1.29	0.49	2.61	0.01	0.32	2.25
Scrambler	-1.94	0.19	-10.05	0.00	-2.32	-1.56
BT:LULOV	0.01	0.20	0.04	0.97	-0.38	0.39
Scrambler:LULOV	-0.04	0.19	-0.22	0.83	-0.41	0.33
% Overlap	-13.51	1.21	-11.15	0.00	-15.89	-11.12
N-Forms	0.46	0.02	24.69	0.00	0.42	0.49

Table 6: OLS Regression Results

	Coefs.	[0.025	0.975]
Intercept	3.62	1.37	9.53
Scrambler	0.14	0.10	0.21
BT:LULOV	1.01	0.69	1.48
Scrambler:LULOV	0.96	0.66	1.39
% Overlap	0.00	0.00	0.00
N-Forms	1.58	1.52	1.64

Table 7: exp Coefs. from OLS regression results

on running time. From this we can see that under the conditions tested, all else equal, when Scrambler is able to find a solution, it does so in approximately 14% time it takes backtracking search to find a solution. We also see that LULOV with backtracking search neither a statistically significant ($p = 0.97$) nor practically significant impact on the time to solution for backtracking search.

6 Conclusions

Under the conditions tested, Scrambler was the superior algorithm in terms of both the probability of finding a solution, and in the amount time taken to find a solution. Backtracking search, although it was able to find a solution under nearly all the conditions Scrambler did, it was significantly slower. This is less an indictment of backtracking search, and more an indictment of LULOV as regression results indicate that it had no effect either the running time or the probability of finding a solution.

I think the MRV and LRV heuristics are likely superior to LULOV for this problem, but their default implementations in aima-python combined with the representation used, could have hurt their performance. It would be interesting to to see a future comparison between Scrambler and backtracking search with more efficient implementations of MRV and LRV

For the typical use case, Scrambler appears to be the superior algorithm when it comes for scrambling test forms by blocks. The typical use case for this algorithm is on 8 forms, and it was in this case that Scrambler was the clear victor. However, backtracking showed promise when the number of forms exceeded use case. Here, the playing field was level because the time limit came before the algorithm hit its maximum number of restarts. It would be interesting how these two algorithms performed when as the number of required forms gets large.

References

[1] Data Recognition Corporation. *Technical Report for the 2014 Pennsylvania System of School Assessment*. Tech. rep. Data Recognition Corporation, 2014.

[2] Peter Norvig et al. *aima-python*. Version 2015.2.8.5. 2015. URL: `https://github.com/hobson/aima`.

[3] Guido van Rossum. *Python*. Version 2.7.11. 2015. URL: `http://www.python.org`.

[4] Skipper Seabold and Josef Perktold. "Statsmodels: Econometric and statistical modeling with python". In: *9th Python in Science Conference*. 2010.

[5] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30. DOI: `http://dx.doi.org/10.1109/MCSE.2011.37`. URL: `http://scitation.aip.org/content/aip/journal/cise/13/2/10.1109/MCSE.2011.37`.