CREATION OF NOVEL FLUID INTELLIGENCE TASKS

By

David May

A capstone project submitted for Graduation with University Honors

May 6, 2021

University Honors
University of California, Riverside

APPROVED

Dr. Aaron Seitz
Department of Psychology

Dr. Richard Cardullo, Howard H Hays Jr. Chair
University Honors

Abstract

Fluid intelligence tests, measuring one's ability to solve novel problems without prior knowledge, are increasingly popular as a measure of far transfer from working memory training tasks. Creating mobile measurements that can be repeatedly administered with different items is crucial to enable more complex investigations of individual differences and cognitive ability training. The present study tests the validity of newly generated variants of existing fluid intelligence problems. The original 23 problem sets we used came from a recently developed mobile test, the University of California Matrix Reasoning Task (UCMRT). We have created a test question tool that can be used to manually generate variants of the existing problems with alterations to colors, shapes, rotations, and orientations. These newly generated variants will be validated to determine whether they are the same difficulty level as the original problems. Following validation of these variants, the question generation tool will be developed further to allow automated creation of new test problems. Implementation of automated problem generation will enable repeated testing of the same individuals to more accurately measure the effects of working memory training tasks over time.

Acknowledgements

Dr. Seitz- for helping me through the process of the Capstone project and keeping me on track for the past year.

Mari Hayashi- for rescuing me whenever I got confused by the codebase.

Randy Mester- for helping me learn C# and Unity, and for introducing me to new concepts like rebasing.

Angie Quagletti, Katerina Christhilf, Rachel Nicole Smith, Anja Pahor, and Susanne Jaeggi- for all the hard work you all did to make this project happen.

Table of Contents

1. Introduction

1.1 Brain training

Brain training refers to the concept that in the same way one can repeatedly perform physical activity to improve their strength over time, certain cognitive tasks can be used to improve particular aspects of mental fitness. One of the aspects of mental fitness is fluid intelligence. Fluid intelligence is the capacity to reason abstractly and solve problems without using knowledge acquired through experience (Cattell, 1963). One of the major difficulties of developing brain training applications is verifying that they are actually leading to improvements in the user's cognition, rather than just making them better at the game they are using to train. For this reason, it is essential that researchers have access to accurate and easy to administer tests that can track improvements in mental cognition. One test, the University of California Matrix Reasoning Task (UCMRT), was developed in response to this need. The present study aims to expand the already existing UCMRT to allow for random generation of new problem variants to evaluate fluid intelligence. Currently, we have developed a program that allows a user to efficiently create new UCMRT problems but have not implemented full random generation.

1.2 Measuring fluid intelligence

One widely used matrix test for measuring fluid intelligence is Raven's Advanced Progressive Matrices (APM) test (Arthur & Day, 1994). APM consists of 48 matrix reasoning problems of increasing difficulty. This test has limitations due to price, difficulty of distribution, and time required to administer. APM is only administered in person via answer sheets and test question booklets and takes around 40-60 minutes to complete, which may cause fatigue in the test taker and lead to less engagement by the test taker (Ackerman & Kanfer, 2009). Since the test only contains 48 problems, it is not conducive to repeated testing of the same individuals. In

an attempt to address this issue, Sandia National Laboratories developed a program that could generate a large quantity of matrix style problems similar to those used in APM (Matzen et al., 2010). Although this software resolved the issue of quantity, it sometimes produced problems that were difficult to discern due to a great number of overlapping shapes (Pahor et al., 2018).

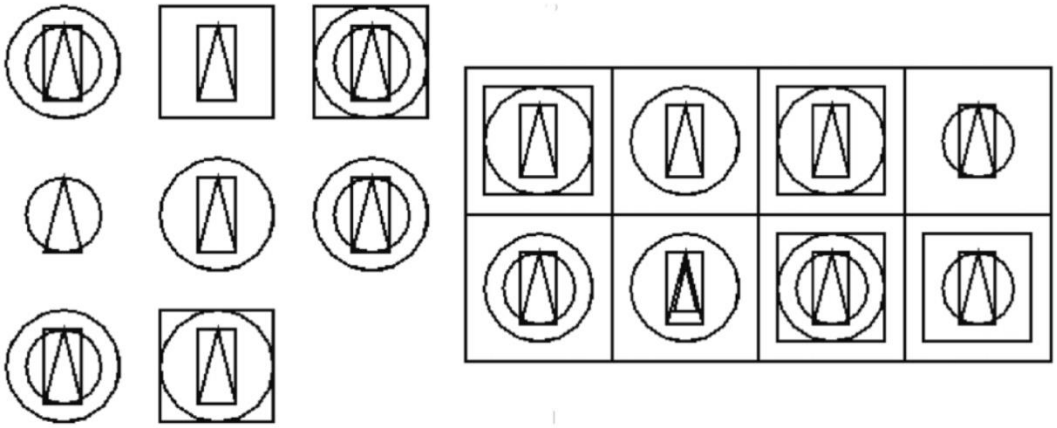

Figure 1.1: A matrix problem produced by the Sandia software.

UCMRT is a more recently developed test, based on the same matrix style problems as APM and Sandia, created to address some of their limitations. The improvements that UCMRT made over previous matrix tests include its ability to be administered on tablets and its reduced test time of around 15 minutes. It can also be administered without an administrator present. Additionally, the problems in UCMRT are more user friendly because although they use the same problem type and structure, they are not only grayscale and they do not feature overlapping shapes. To further improve UCMRT, this project led to the development of a new tool that enables a user to efficiently create new problem variants.

1.3 Why randomly generate problems?

Currently, we have replicated the original UCMRT problems using our newly developed application. Once these new versions of the original problems are validated to determine their similarity in accuracy to the original test, the application will be extended to allow random

generation of new problem sets. The randomly generated problems will be constrained in features such as colors and positioning of shapes to ensure accessibility and to maintain accuracy. Random generation of problems is desirable because previous matrix reasoning tasks have been limited in the quantity of their problem sets. With a limited problem set, researchers may be unable to effectively test the same individuals many times because they could develop recognition of the problems. Since fluid intelligence pertains to problem solving without using knowledge from past experiences, this recognition may lead to less accurate measures of fluid intelligence. Randomly generating problems allows us to create a nearly unlimited number of problem variants that could be used for repeated testing over time.

2. Methodology

2.1 Types of matrix problems

UCMRT problems can be broken down into 6 categories. These include one-relation, two-relation, logic, and three-relation with one, two, or three transformations (Matzen et al., 2010). The relations correspond to one or more rules that define the pattern found within a problem. The rules used are shape, color, quantity, size, rotation and position. Figure 2.1 shows a three-relation problem from the original UCMRT problem set. The 8 squares on the left side of the screen show the pattern that the user needs to recognize, with the 9^{th} square in the bottom right being the one they need to fill in to complete the pattern. The 8 squares on the right side of the screen represent the user's answer choices. The three relations in this particular problem are shape, color, and rotation. There are also three transformations in this problem. The change in rotation propagates from the top left square to the bottom right. Specifically, the trapezoid on the top left is not rotated at all, the next diagonal row down is rotated by 45° to the right, and the middle diagonal row is rotated 90°. These rotation increments of 45° continue down to the

bottom right square. The shape changes in this problem can also be split up into diagonal rows, but this time propagating out from the top right square. Finally, it can be seen that like shape colors are organized into diagonal rows. Figure 2.2 shows an "OR" logic problem. Logic problems can be solved by looking at the third square in any given row or column and determining its' relation to the two squares that came before it in that row or column. Looking first at the top right square, we can observe that it contains the green circle from the top left and top middle squares, and it contains the purple rectangle from the top left square. With this information, we can already determine that this is an "OR" problem. If the problem were an "XOR" type, the green circle would not be included in the top right square because it is found in both of the squares to the left of it. If it were instead an "AND" problem, the purple rectangle and two blue triangles would not be included in the top right square because they weren't present in both of the first two squares. Patterns like the ones described in this section make up the basis for all UCMRT matrix problems, as well as Raven's APM problems (Pahor et al., 2018).

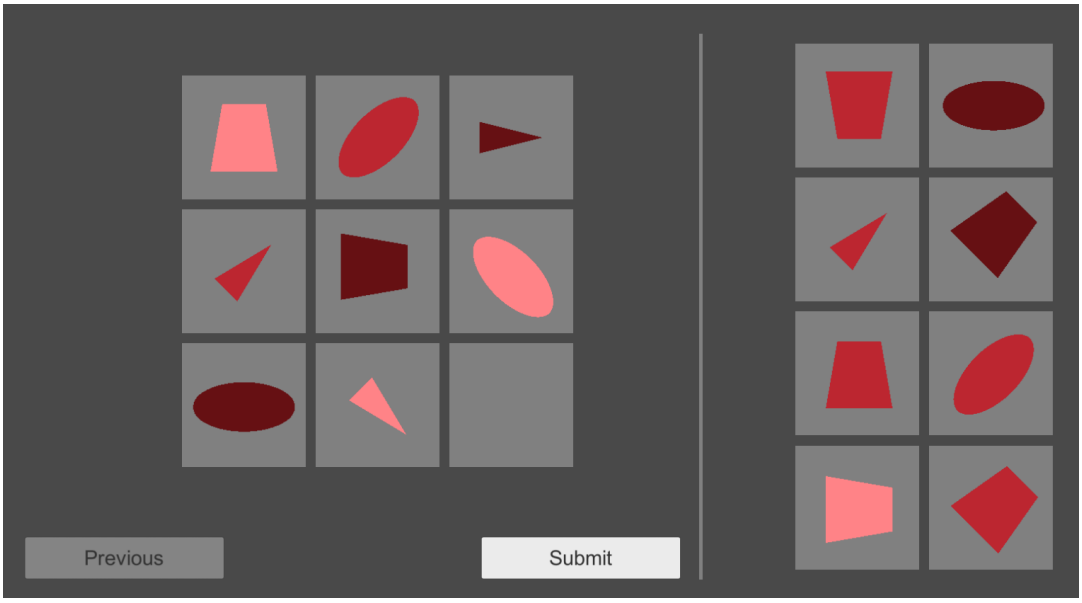

Figure 2.1: A three-relation UCMRT problem.

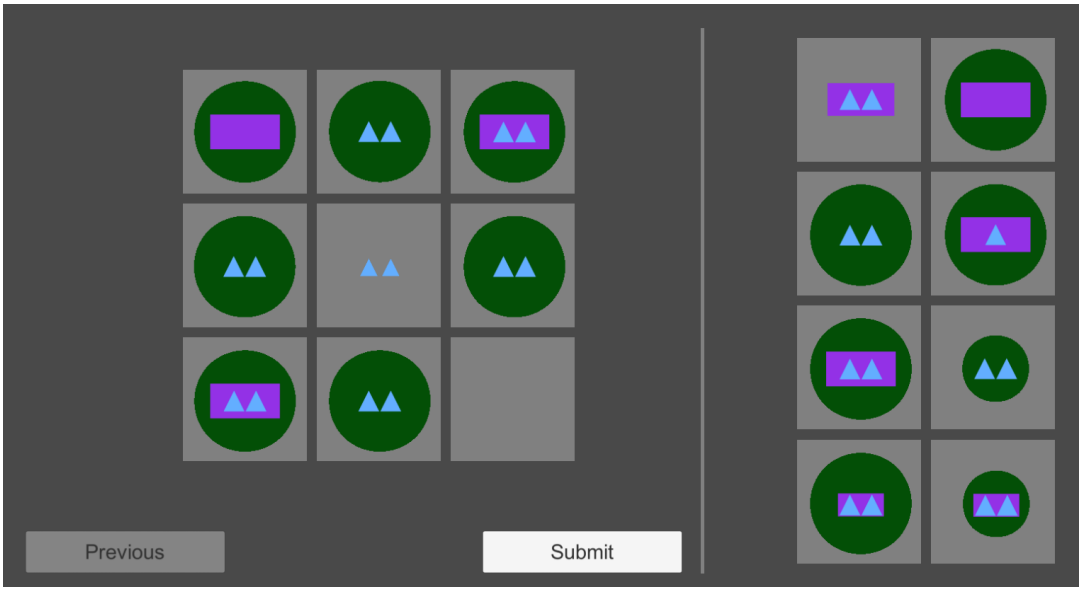

Figure 2.2: An “OR” UCMRT logic problem.

2.2 Using Unity to draw shapes

Given that the framework of the problem generation tool had already been created, the first task was to write code that could draw each of the shapes to be used in the matrix problems. The shapes included circles, squares, triangles and trapezoids. To do this, we used a native Unity class called “Texture2D” which enables us to specify the color of each pixel in a square. The function within Texture2D that allows us to define the color of each pixel is called SetPixel. The SetPixel function takes three parameters. The first two parameters determine which pixel we are setting the color of using the standard Cartesian coordinate system. The third parameter specifies the color that the pixel will be set to. For example, if I wanted to make a texture with a single white dot in the lower left corner, I would call SetPixel(0, 0, Color.white) because the bottom left corner of the square we are drawing in is the point (0, 0). To draw an entire shape, we call the SetPixel function on every point in the square, assigning its’ color to clear if it is outside the shape we’re trying to draw and assigning it to the shape’s specified color otherwise. A unique function was required for each shape. As an example, consider how to draw a triangle in code. First, we iterate over every pixel using nested for loops. Inside these for loops, we check whether

our x value is less than or equal to the size of our square divided by two. We make this check because the left and right sides of a triangle have different slopes, so we need to write different code to draw each. If our x value is less than size / 2, we then check whether our current y value is less than or equal to 2 multiplied by our x value (y <= 2 * x). We can arrive at this formula by starting with the common equation for a line, y = mx + b. Since we're trying to draw a triangle with a height of "size" and a width of "size," we can determine that the slope of the left line making up our triangle is 2 and we can plug this value in for 'm' in the equation (y = 2x + b). Additionally, since the left line of the triangle starts at the point (0, 0), we can eliminate the b value in our equation (y = 2x). Next, we consider the fact that the bottom of our triangle is at the point y = 0, so as long as our (x, y) values are on or below the line we found (y = 2x) they will be part of the triangle. This is the thought process that allows us to arrive at the simple statement y <= 2 * x, which enables us to draw the left-hand side of our triangle. A similar process is used for the right side of the triangle, but it requires a more complicated statement since the intercept value is not 0 as in the case of the left side.

2.3 Drawing stripes over shapes

In order to recreate all of the original UCMRT problems using our program, it was not enough to simply be able to draw the shapes used. We also needed to add a stripe feature that could draw a variety of different stripe patterns on any given shape to replicate problems such as the one shown in figure 2.3. Specifically, the stripes had to be drawn using any color specified by the user, and could be drawable as horizontal, vertical, tilted 45° to the left, or tilted 45° to the right. Additionally, the stripes need to be customizable in their width and distance from each other.

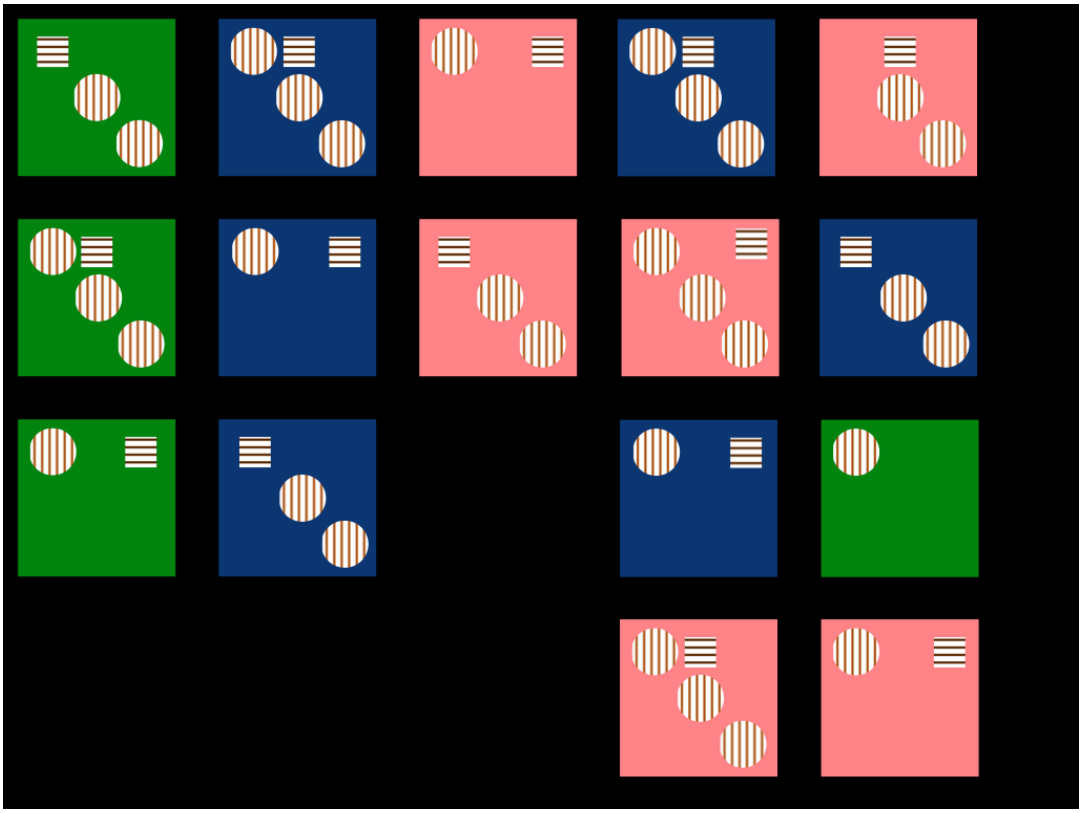

Figure 2.3: A UCMRT problem including stripes.

There are three parts to our stripe implementation. First, we draw the specified shape as discussed in the previous section. We do this first because the stripes need to be overlaid on top of the shape so that they are visible. Next, we determine which direction the stripes are supposed to be oriented in and call the function DrawStripe(), passing in an integer parameter that represents the midpoint of a particular stripe. Finally, the DrawStripe function iterates through the entire length of a stripe centered at the midpoint parameter and changes the color of the pixels along it.

Stripe midpoints are calculated based on the field “stripeDistance.” This field specifies how many pixels separate two stripes that are next to each other. We first set the midpoint of the center stripe. For horizontal and vertical stripes, the center stripe will be centered on the point “size / 2” because we want it to pass through the center of the shape. For stripes angled 45° to the right the center stripe will start at the point ‘0’ because we want it to start at the bottom left of the

shape. For stripes angled 45° to the left, the center stripe will start at the point “size” so that it will start at the top left of the shape. Once we’ve identified our first stripe midpoint, we call DrawStripe() to create the first stripe. From there we create two variables “startingPointRight” and “startingPointLeft” and assign them with the sum of our midpoint plus stripeDistance and the midpoint minus stripeDistance, respectively. These two variables store the midpoints of the next two stripes we will draw. Next we call DrawStripe() on the newly created variables and then increment and decrement our startingPoint variables again to get the midpoints of the next two stripes. This continues until the entire field has been filled with stripes.

The code described above does not account for the fact that not every shape fills the “size” by “size” square that the shapes are drawn in. If implemented exactly as described, stripes would span the entire length of the square, beyond the boundaries of each shape. To account for this we first draw the shape and then within DrawStripe() we call another function, DrawPixel(), to change the color of a particular pixel to the value of our variable stripeColor. In DrawPixel(), we check whether the color of the pixel at Cartesian coordinates (x, y) is clear. If the color of the pixel is clear, rather than shapeColor, this means that the pixel is not part of the shape and we should therefore not change the color of that pixel. If the pixel is not clear, however, that means it is a valid stripe location and we update the pixel’s color to stripeColor.

2.4 Creating matrix problems

There are three main steps to using the UCMRT logic problem creation tool. The first step simply requires a user to enter a filename, target index, and type of logic problem. The target index indicates which of the eight answer choices will be the correct one, and the type of problem can be “AND,” “OR,” or “XOR.” The next step in the creation of a logic problem is defining all of the shapes that the problem will use. The “Add/Edit Shape” menu provides a great

deal of customization, allowing users to specify a shape's color, x and y ratio, location within a problem square, rotation, stripe width and stripe distance. Once these fields are filled in, the shape will be added to the "Add or edit shapes" menu where it can be edited or deleted. If a necessary field is left empty, a message will be displayed prompting the user to return to the edit menu and fill in the field. If a non-essential field is left empty, the field will be auto filled, and the user will not be prompted with an error message. For example, if the user types "False" into the "Stripes" field, indicating that stripes will not be included on the shape, then the user is not required to fill in the stripe color, width, direction or distance fields.

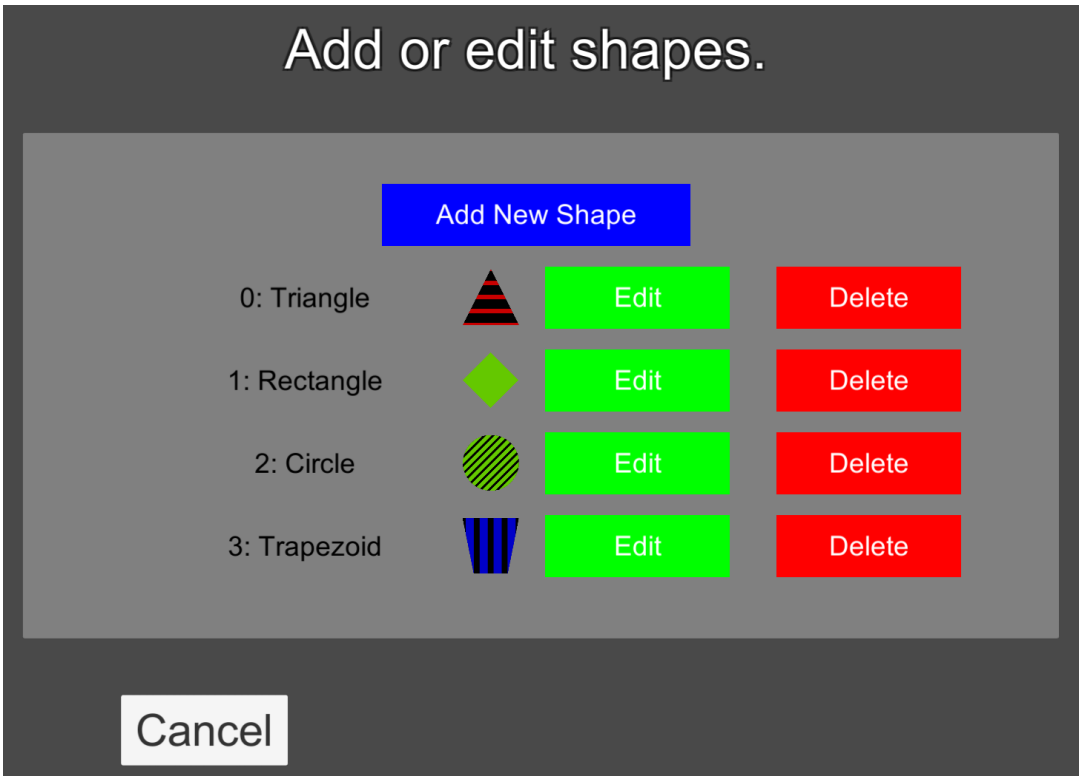

Figure 2.4: The "Add or edit shapes" menu where a user can view the shapes they have defined.

The final step of creating a logic problem is specifying which shapes will go in each square of the matrix. Shape positions can be specified using the "R#C#" buttons on the main page of the logic problem builder. The "R" and "C" numbers correspond to a row and column of the matrix. When a user clicks on R1C1, the shapes that they choose to add will be placed in the

relevant locations of row and column 1 depending on the type of logic problem that was specified in the first step. Once the logic problem is ready to export, the “Create JSON” button is clicked which exports the problem in a JSON format that can be interpreted by the program later to display for a research participant.

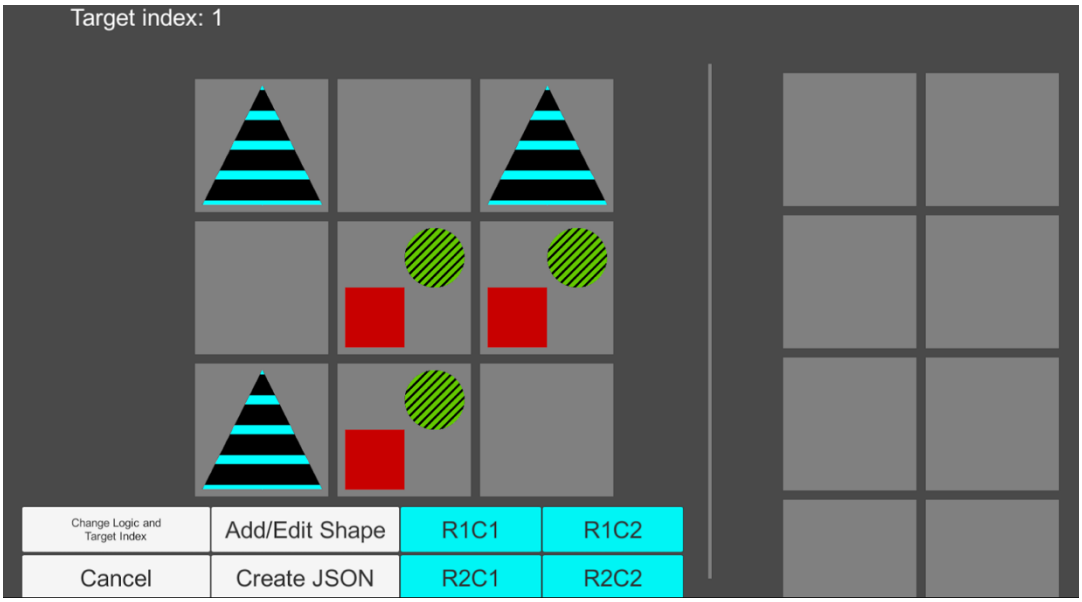

Figure 2.5: The main create logic problem page, with a sample “work in progress” logic problem.

2.5 Edge cases

While generating variants of the original UCMRT problems, we occasionally encountered edge cases that did not translate well to the new tool. Edge cases are situations that would not typically arise through normal use of a program, but that can be produced through a particular input or series of inputs. The most common issues were caused by stacked shapes that didn’t fit properly within each other. For example, figure 2.6 shows an original UCMRT problem, while figure 2.7 shows the problem variant generated using the new program. In the newly generated variant, the shapes overlap in such a way that decreases visibility. We encountered similar issues in three problems that were translated from the original set. Sandia’s problems occasionally sacrificed ease of understanding for the sake of producing a great number

of problem variations. Since UCMRT aims to improve on the tests produced by Sandia, the three problematic variants were excluded from the final set of problems used in our early pilot testing. Over time, as we gain a better sense of what inputs can lead to these edge cases, we will be able to refine the code in such a way that avoids them or disallows them altogether. Consideration of edge cases also presents a unique challenge in the process of adding random problem generation. It will be crucial to account for and avoid problematic edge cases. As discussed in section 1.3, special care must be taken to maintain the same level of accessibility that the original problem sets demonstrated. If this is not handled properly, testing may reveal that the randomly generated problems exhibit reduced accuracy when compared to Raven's APM and the original UCMRT problem set.

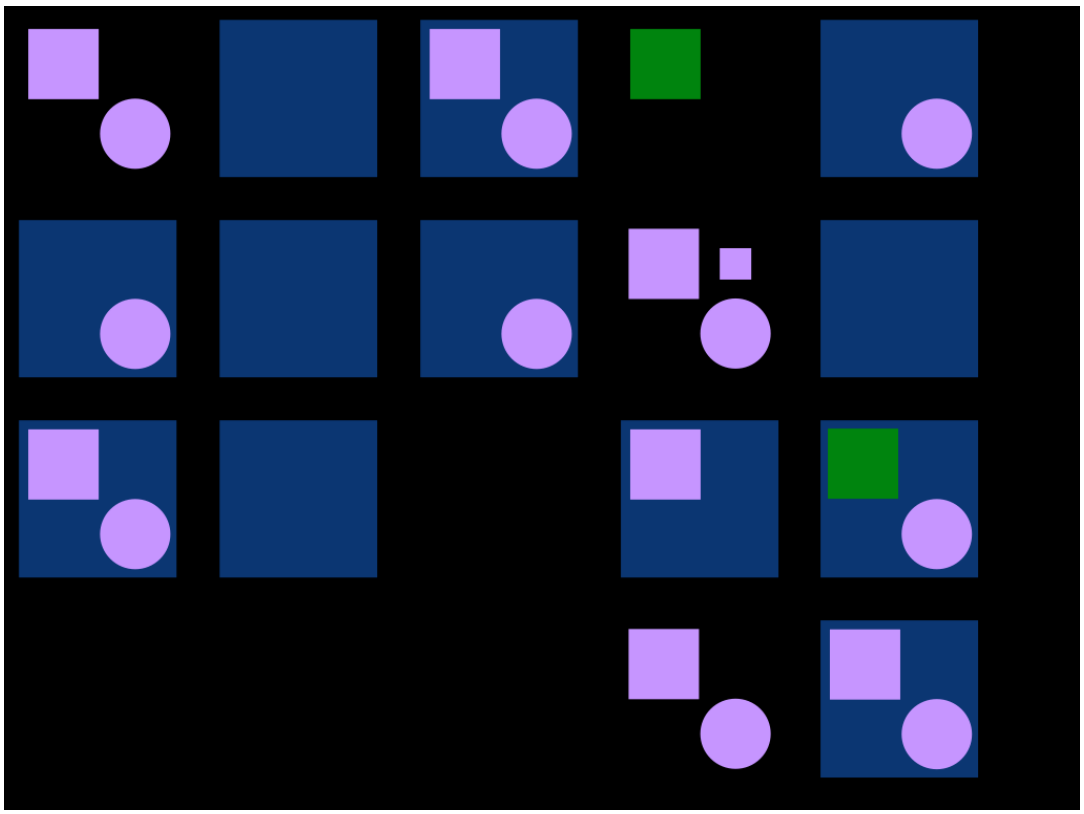

Figure 2.6: A sample problem from the original UCMRT problem set.

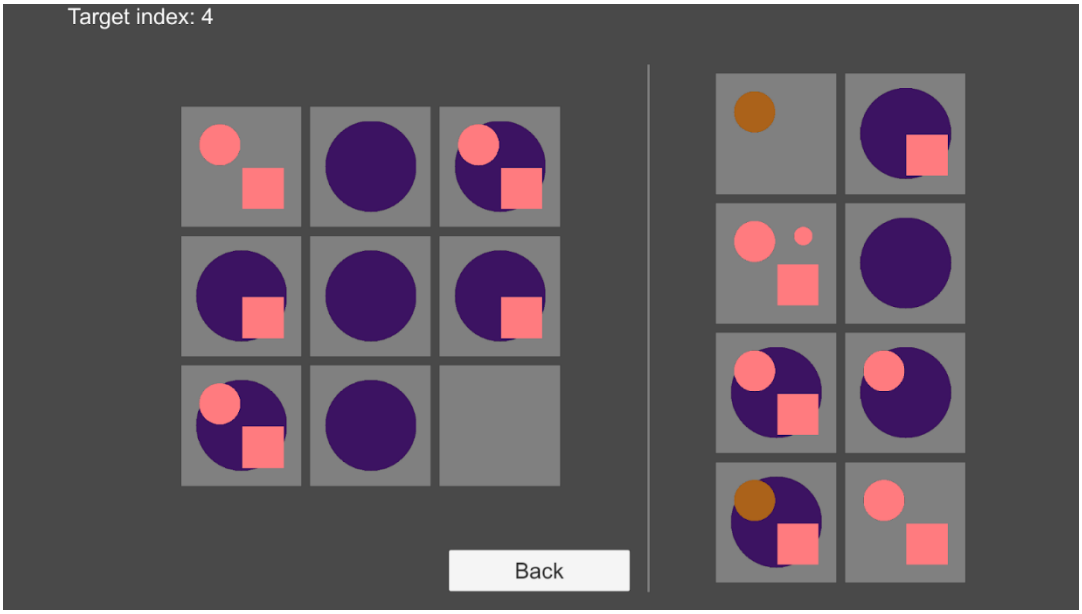

Figure 2.7: A problematic edge case generated using the new UCMRT program.

2.6 Current pilot study

At present, we have completed the program that allows a user to easily generate logic problems based on a set of manually defined inputs. We have used the program to generate three problem sets based on the original three UCMRT problem sets. Although our newly generated problem sets are very similar to the original ones, it is still important to validate that the results of the tests have not been altered significantly by the slight changes in appearance. Toward this end, we are currently running a small pilot program involving RA's at the University of California, Irvine before moving forward with adding the random generation feature.

2.7 Post-pilot tasks

After the accuracy of our new problem variants is verified by the pilot study, we will be able to move forward with adding random generation to the program. The random generation code will be based on the guiding principles discussed for each problem type covered in section 2.1. Essentially, each variation of matrix problems is defined by a certain number of rules. If we are trying to generate a two-relation problem, our random generation code will select two rules at

random apply a series of shape, color, or location transformations depending on which rules were selected. The difficulty in creating the random generation feature will be in ensuring that valid, user friendly problems are created. There are many niche cases that must be taken into consideration and many scenarios that need to be prohibited by the code. For instance, the code cannot allow a problem to be created where a rotation rule is applied to a circle. However, we still want to maximize the variety of problems that can be generated. So, if the shape chosen is a circle with an x or y ratio change applied, making it an oval, the rotation rule can be applied. This brings us back to the conversation of edge cases, which must be thoroughly explored in the process of making the randomization feature.

3. Challenges

3.1 Learning C# and Unity

Before contributing to the UCMRT codebase, I needed to learn two new tools, the programming language C# and the game development engine Unity. To accomplish this, I first completed a short, guided tutorial in which I created a simple tablet game to familiarize myself with the syntax of the C# language. Next, I made an addition to the project independent of the tutorial to demonstrate my ability to use C# and Unity in an unguided setting. Familiarizing myself with these new tools was difficult because they differ substantially from coding languages and tools that I've used in the past for coursework and side projects. After completing the Brain Game Center onboarding process, however, I was able to learn more about the tools by using them while working on the project itself.

3.2 Interfacing during a pandemic

Further challenges were introduced by the ongoing global pandemic that began around the same time that I began working on the UCMRT project. These challenges were further

exacerbated by the fact that two teams at separate universities, the University of California, Riverside as well as the University of California, Irvine, were working together on the project. The UCR team was responsible for developing the coding tools, whereas the UCI team handled planning and creation of the actual UCMRT problems using the application developed by the UCR team. With the exception of meetings held every other week, all communication between the teams was done through email or Slack messaging. This added a level of difficulty because any complicated questions or clarifications needed to be communicated in back-and-forth email chains. The challenge of working with the teams digitally persisted throughout the course of the project, but it was mitigated by frequent communication and thorough updates in the bi-monthly meetings.

3.3 Contributing to an evolving codebase

Another challenge encountered while working on the project was the result of multiple team members contributing to the same codebase. At the time that I began working on the project, the outline of the codebase for the application had already been established. This meant that before contributing any code, I needed to familiarize myself with the structure of the existing code, and after doing this, I needed to determine how to work within the framework that had been set up before I could add any features. In addition to this, the codebase was frequently being updated, occasionally in ways that substantially affected the way the program functioned. As an example, while I was working on implementing the stripe feature, the code was changed in such a way that previously drawn shapes were stored in a dictionary so that they could be quickly used again without redrawing them. This change improved the performance of the program, but it also resulted in errors when I attempted to add the stripe code that I had already written. To fix this, I needed to restructure my code so that it would function as expected. I was

able to overcome the challenge of working in an evolving codebase by checking for updates more frequently so that I could make any necessary changes to the features I was adding before any compatibility issues could propagate into larger problems.

4. Future Developments

4.1 Adapting UCMRT for different populations

Currently, the problems generated for UCMRT have been an appropriate level of difficulty for healthy young adults (Pahor et al., 2018). In a future study, the UCMRT application and the improvements made through this project could be utilized to develop matrix test problems for groups outside the healthy young adult population. For instance, new problems of a different difficulty level could be developed that would be more appropriate for testing elderly populations. This would be beneficial because it would improve the accuracy of fluid intelligence evaluation among this population. Accordingly, creating a test that is accurate for this population could aid in developing brain training programs directed at the elderly. In addition to adapting UCMRT for particular age groups, the random generation feature could be altered and utilized to generate tests for more specific groups that fall outside of the normal range of cognition. Suppose that a researcher wants to use a matrix test in their study, but they only want to use logic type problems. If the researcher were to use Raven's APM or the original UCMRT problem set, they may only be able to generate a very limited test. Having a reduced number of problems can reduce the test's reliability (Matzen et al., 2010). The researcher could, of course, produce a logic problem test manually but it would require a great deal of time and money to manually create the problems and validate them. Alternatively, the UCMRT random problem generation program could produce a nearly unlimited number of valid logic problems, enabling repeated testing of the same individuals over time, and saving the researcher a great

deal of time and funding. Although a great deal of validation would be required before the program gets to this point since randomly generated problems must match or come close to the efficacy of manually generated problems, the potential for UCMRT to benefit countless researchers is eminently clear.

5. Conclusion

5.1 Summary

The original UCMRT application made substantial improvements over Raven's APM, but it still faced the challenge of a somewhat limited collection of problem sets. Sandia's tool attempted to alleviate this issue, but it sometimes produces variants that are not user friendly. At present, we have created a tool that lays the groundwork for random generation of novel problem sets, a feature which could help many researchers within the realm of brain training and beyond.

References

Ackerman, P. L., & Kanfer, R. (2009). Test length and cognitive fatigue: An empirical examination of effects on performance and test-taker reactions. *Journal of Experimental Psychology: Applied, 15*(2), 163-181. https://doi.org/10.1037/a0015719

Arthur, W., & Day, D. V. (1994). Development of a short form for the raven advanced progressive matrices test. *Educational and Psychological Measurement, 54*(2), 394-403. https://doi.org/10.1177/0013164494054002013

Cattell, R. B. (1963). Theory of fluid and crystallized intelligence: A critical experiment. *Journal of Educational Psychology, 54*(1), 1-22. https://doi.org/10.1037/h0046743

Matzen, L. E., Benz, Z. O., Dixon, K. R., Posey, J., Kroger, J. K., & Speed, A. E. (2010). Recreating Raven's: Software for systematically generating large numbers of Raven-like matrix problems with normed properties. *Behavior Research Methods, 42*(2), 525-541. https://doi.org/10.3758/BRM.42.2.525

Pahor, A., Stavropoulos, T., Jaeggi, S. M., & Seitz, A. R. (2018). Validation of a matrix reasoning task for mobile devices. *Behavior Research Methods, 51*(5), 2256-2267. https://doi.org/10.3758/s13428-018-1152-2