Advanced Corvus User Manual

Corvus is a Raven's-like test generator written in JavaScript. Further details on it, including an explanation of some of the terms used here, can be found in the author's DPhil "Corvus: an Automatic Raven's-like Test Generator", specifically chapters 1 and 4.

Corvus demonstrates the development of an approach to performing cognitive experiments at scale.

As indicated by the version number, Corvus is in ongoing development. There are multiple ways in which Corvus could be improved or altered to fit specific tasks. Corvus is an experimental system designed to enable the studies mentioned in the thesis, and for many other purposes.

It is assumed that investigators will wish to tailor Corvus to their own purpose and need. This manual is written as a brief introduction to help assist users in that endeavour, however this manual does assume that such investigators are comfortable reading and editing JavaScript as a minimum requirement.

If you are not comfortable reading and editing JavaScript, you may find Corax to be of assistance. Corax does not provide the full range of options that editing the script would enable, but it does allow users to tailor and generate their own test items and tests.

At time of writing this user manual (2019), the version of Corvus currently provided on GitHub (v0.8.8) will work on local machines and will download the results once the test is completed to the user's device. This download facility requires a modern browser. However it is presumed that a user will want to alter Corvus to record participant data to their own secure database, and this is what has been done for the author's studies.

The code that triggers downloading the test results can be found in '…/JavaScript/08-main.js' by searching for '// ### DOWNLOAD FUNCTION ###'. This line calls code that can be found in "…/Other files/download.js" — once you have implemented your own backend solution, "download.js" can be removed.

For further guidance on specifically tailoring Corvus to a user's own needs, user's may please feel free to contact the author by opening an issue on the GitHub repository for Corvus.

General Settings
Some general settings can be altered in "JavaScript/01-properties.js". Specifically, the variable 'currentSet' (Search for '// ### Mouse-over ###'), defines how the options interact with mouseovers. See the comments on preceding lines, or chapter 7 of the author's thesis for more details. This file also defines general properties such as the size of icons and the thickness of lines.

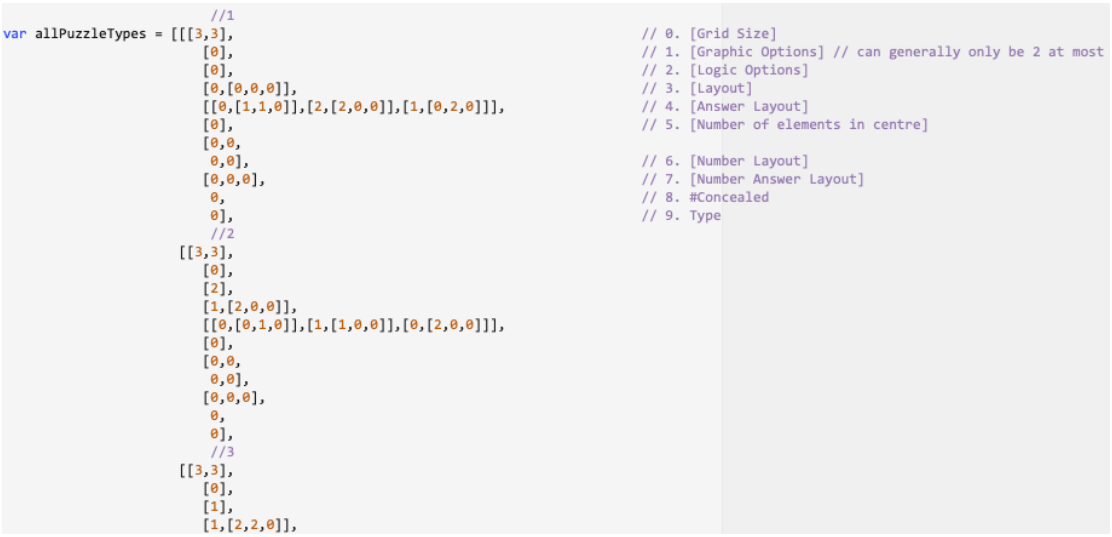
Tailoring Test Items
The key files to be edited start with the prefix '02_', and possibly parts of the file '04-pattern.js'. Editing the graphical appearance of items can be done via files with the prefix '06_'.

The file '02_04-testItems.js' handles definitions for fixed items. Elements of these fixed items can then overwritten by the functions in files '02_02-RNGAnulus.js', '02_03-RNGAdd.js', '02_04-RNGLG.js'. The file '02_01-orders.js' is primarily a commented-out function that can be used to calculate an array that generally does not change test-to-test (the results of which have been hard-coded at the end of this file as a result), as well as a function for shuffling arrays. The rest of this section of the manual describes '02_05-testItems.js'.

The file '02_05-testItems.js' begins with a large comment detailing some of the options available for each test item. Some of the options listed were included as future proofing for potential ways to expand on Corvus. Specifically, sizes of Grid other than [3,3], options 8-22 under Logic Options, and option 1 under type are not implemented as of Corvus version 0.8.8.

Note that in the code, 112 is shorthand for distribution of 2, and 123 is shorthand for distribution of 3. Similarly, 'Add' is sometimes used as shorthand for Addition.

This is followed by the array AllPuzzleTypes. This array is where each test item is specified. Further details on its structure, beyond those presented here, can also be found in section 4.2 of "Corvus: An Automatic Raven's-like Test Generator".



```
                        //1
var allPuzzleTypes = [[[3,3],                                   // 0. [Grid Size]
                       [0],                                     // 1. [Graphic Options] // can generally only be 2 at most
                       [0],                                     // 2. [Logic Options]
                       [0,[0,0,0]],                             // 3. [Layout]
                       [[0,[1,1,0]],[2,[2,0,0]],[1,[0,2,0]]],   // 4. [Answer Layout]
                       [0],                                     // 5. [Number of elements in centre]
                       [0,0,
                        0,0],                                   // 6. [Number Layout]
                       [0,0,0],                                 // 7. [Number Answer Layout]
                        0,                                      // 8. #Concealed
                        0],                                     // 9. Type
                        //2
                      [[3,3],
                       [0],
                       [2],
                       [1,[2,0,0]],
                       [[0,[0,1,0]],[1,[1,0,0]],[0,[2,0,0]]],
                       [0],
                       [0,0,
                        0,0],
                       [0,0,0],
                        0,
                        0],
                        //3
                      [[3,3],
                       [0],
                       [1],
                       [1,[2,2,0]],
```

Figure 1: A screenshot of the first few elements of allPuzzleTypes.

This array is followed by a for loop, which iterates through each test item, and replaces parts of each test item with output from the functions in the other JavaScript files starting with "02".

If each participant is to take the exact same test, the random number generator seed can be fixed in "…/random.html" (Search for '// ### Seed ###'), to Math.seedrandom("X") where "X" is any fixed string. If participants are to take very similar tests this loop can instead be disabled (Search for '// ### Override allPuzzleTypes ###'), and the test items to be taken are specified in AllPuzzleTypes. By combining both of these modifications it is ensured that all participants take the same test items as specified by AllPuzzleTypes.

The reason both modifications are necessary is that even without that for loop AllPuzzleTypes does not fully specify all items. Some parts of each test item are left to random number generators, such as if a test item uses squares, triangles or circles. Similarly, the order options are presented in is randomised.

By default Layout, Answer Layout, Number of elements in Centre, Number Layout, and Number Answer Layout are all overwritten by functions after allPuzzleTypes has been initially defined. If there is a need to specify the exact test participants are asked manually, commenting out much of the section immediately following allPuzzleTypes should work.

Graphic Options
The length of this array defines the number of patterns types used. Corvus 0.8.8 is currently only intended to handle one or two pattern types, and if two are needed, one of them – and only one – must be Annulus (i.e. 0).

The *graphic option* chosen must be able to support the *logic option* chosen next. The following table indicates compatibility between *graphic options* and *logic options*. A question mark indicates that that combination of parameters has not been tested with the current version of Corvus, but that it should work in principle or with a small number of alterations or updates to the code.

	Identity	Distributions	Addition	Logic Gates
Annulus	✓	✓		
Dice	?	?	✓	
Petals	?	?	✓	
Spike Rings	?	?	?	
Tessellating Squares	?	?	?	
Tessellating Triangles	?	?	?	
BoxLines	?	?	?	✓

As may be clear from the number of question marks in the table above, this was not considered a priority for the current version, as a minimum of only one *graphic option* is needed for each *logic option*.

These *graphic options* are defined in the JavaScript files starting with the prefix “06”.

Logic Options
This array should be the same length as the *Graphical Options* array, with each element of each array corresponding sequentially to each other.

There are seven logic options currently available. The *logic option* chosen here defines if and how the following six options are used, as per the table below.

	Identity	112	123	Addition	AND	OR	XOR	XNOR
Layout		✓	✓					
Answer Layout		✓	✓					
Number of elements in centre				~	✓	✓	✓	✓
Number layout				✓	✓	✓	✓	✓
Number answer layout				✓	✓	✓	✓	✓
#Concealed		✓	✓					

At present, parameters with ticks in the same row are mutually exclusive. This table — and how it combines with the previous table — is why *Graphic Options* with two pattern types must incorporate one using Annuli, and the other not.

An example combining 123 and XOR can be found towards the end of this manual.

Layout
Layout is an array in the form [magnitude, [colour, shape, rotation]] or [magnitude, [colour, shape, rotation], [colour, shape, rotation]], depending on the number of annuli wanted.

Note that Corvus does not check coherence. If the same layout for two different rules is used, it will give the test item solving redundancy; i.e. multiple independent ways of solving the same test item.

Each entry in the array defines the form the rule defined by *Logic Options*. A zero entry defines an Identity – or arguably, a lack of pattern. An entry of 1 or 2 randomly defines an orthogonal (horizontal or vertical) form; if both 1 and 2 then Corvus ensures that they are different to each other, but still randomly assigned to vertical or horizontal. Similarly, if two entries are the same, Corvus ensures that they are assigned to the same resulting form. Entries of 3 or 4 work similarly, but for increasing and decreasing diagonal forms.

If two pattern types are to be combined, it is advised that two annuli are not to be used (i.e. define Layout to be in the form [magnitude, [colour, shape, rotation], [colour, shape, rotation]]). While this will work, it will result in the inner most shapes being very small, which could present issues.

N.B. Layout is the word I initially used to refer to Form. Form is the test item property defined by the variable named layout.

Answer Layout
Answer Layout is an array of elements, each individually taking the same form as the Layout. The length of the array determines the number of incorrect options presented.

If Layout uses two annuli, then all elements of Answer Layout should also do the same.

However, here each non-zero number represents a delta from the correct answer. In other words, [0, [0,0,0]] would be the correct answer (for a test item using one annuli, and no other pattern type) – however the correct answers should not be included here; they will be added automatically at a later date, then the order of the answers will be shuffled.

An answer layout of [[0, [1,0,0]], [0, [0,1,0]], [0, [0,1,1]]] would generate a test item with four options in a random order. One of which would be the correct answer. One of the alternative options would differ in a randomly determined way in regard to colour, another would differ in regard to shape, and the last would differ in the same way as the second in regard to shape, but also in rotation.

Similarly to the way entries worked in the previous parameter, non-zero digits are randomly assigned to a particular attribute value. The digit 1 assigned to colour in one alternative option, will be assigned to the same attribute value as a 1 assigned to colour in another alternative option.

NOTE: If any matrix element or option has a rotation relative to the others, for any reason, then all matrix elements and options will have a ‘rotation tab’ which removes rotational symmetry.

Number of Elements in Centre

This variable is used by Corvus to interpret the following two variables. It defines the number of sub-elements used in each element.

Number of elements in centre is always 1 for Addition.

It may be the case that it is possible to remove this from the code, as in, in theory, it could be made redundant. However doing so has not been a priority.

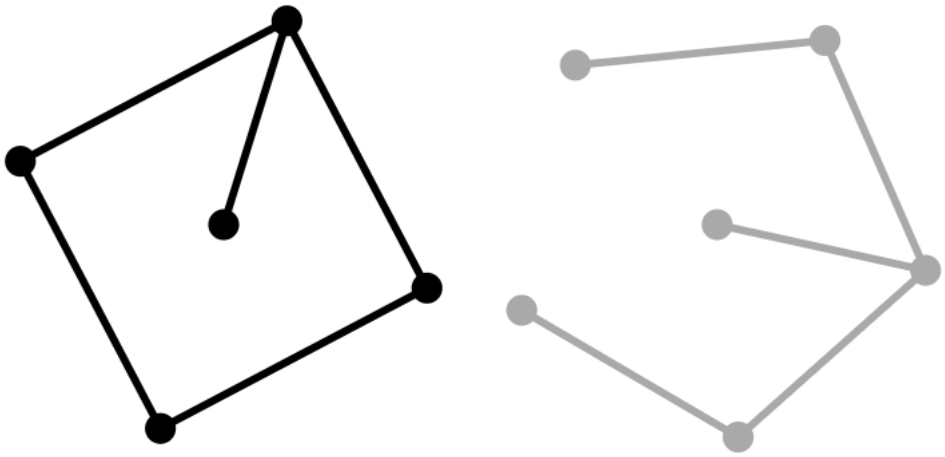


Figure 2: To generate elements like the example on the left the 'Number of Elements in Centre' should be set 5, while for generating elements like the one on the right, the same parameter should be set to 6.

Number Layout
Number Layout works slightly differently for Addition and Logic Gates. However in both cases Number Layout has 4 parts – each representing the four elements in the grid that are not in the same row or column as the missing element. If the missing element is in the bottom right-hand corner, then these arrays map onto the four elements in the top-left of the grid or matrix as follows; [top-left, top-centre, centre-left, centre].

These parts are arrays with length equal to *Number of Elements in Centre*, or integers when *Number of Elements in Centre* is one (i.e. for Addition).

With Addition, each integer is graphically encoded directly using the *Graphic Option* selected above. It is important to ensure that each of the integers, the sum of each row and column and the sum of all four integers all lie within the range that the *Graphic Option* can handle. For the Dice and Petal encodings, this range is -9 to +9. Though the Petal encoding does not have as hard a limit as the Dice encoding, that is, it could technically manage a larger range, this would not be ideal.

With Logic Gates each sub-element links to the presence or absence of a part of the graphical encoding. Zero indicates no difference between this part of the grid, and the correct answer or missing element. It is important to ensure that every permutation of Logic Gate inputs is

included in the matrix, so that participants can identify, with surety, which Logic Gate is in use. This means when considering each row and column of *Number Layout*, that all four binary permutations (11, 10, 01, and 00) should all be included. Ensuring that they appear in the *Number Layout*, ensures that their output is also included; a binary permutation appearing in the Matrix or grid, but not the Number Layout is a potential problem.

While it is possible to construct test items where this is not essential, doing so requires extra work; i.e. ensuring that every binary permutation that does not appear in *Number Layout*, also does not appear anywhere in the Matrix.

Number Answer Layout

Each element of this parameter is a distinct alternative option, and much like *Answer Layout*, the correct answer would be indicated by zeros, but is not included in this parameter as it is shuffled in at a later point.

As with *Number Layout*, this alters slightly depending on whether Addition or Logic Gates are being used. With Addition, this will be an array of integers, while for Logic Gates they will be arrays of size equal to *Number of Elements in Centre*.

For addition, each number is [double check if includes correct answer – it would be unusual for it to do so, but also I am not sure how setting 0 as the correct answer would work for addition] that element's delta from the correct answer modulated so that the end result lies in the range -9 to +9.

For Logic Gates, each 1 indicates that the sub-element indicated by its array index has the opposite presence or absence to the correct answer.

#Concealed
The concealed parameter functions with the *logic option*s for distribution ("112" and "123").

Concealed values grater than zero cause the function requiredVisSet(), in "…/JavaScript/04-pattern.js" (Search for '// ### Calculate Minimum Unconcealed ###') to run. This function attempts to conceal elements within the matrix until it either cannot conceal additional elements without compromising the test item, or until it has reached the number of elements concealed as set in this parameter.

Functions
In the file ’02_05-testItem.js’, after the array allPuzzleTypes is defined, a series of functions can be applied to the array — and are switched on by default.

These functions are defined in the files ‘02_02-RNGAnulus.js’, ‘02_03-RNGAdd.js’, and ‘02_04-RNGLG.js’. The files are analogous to each other in purpose, respectively for Distributions, Addition, and Logic Gates.

Each file primarily consists of two functions. The first (with the suffix ‘Qu’) redefines the Rule’s layout.

For distributions this is very simply done using taking the definition template at position Math.floor(dif/maxDif^2) of the list of possible definitions in the hard-coded array annulusRuleArray, which is defined in ’02_01-orders.js’ (a lot of the commented-out code in that file can be used to calculate annulusRuleArray). Where dif/maxDif^2 is the test item number divided by the square of the total number of test items in allPuzzleTypes. If upgrading Corvus to work with Adaptive Testing, then variables found in the code such as dif and difficulty would make a good starting place for theta. The array annulusRuleArray only has values 1 for orthogonal, and 2 for diagonal. The function returns values that alternate between the two options within each category of form as it reads the generated Form from left to right, from a random starting Form subcategory.

For Addition and Logic gates, these functions generate a 2x2 array, while ensuring that every element in the full 3x3 matrix can be processed by the graphical encoding chosen. For Logic Gates, the function ensures that the full set of Boolean combinations and their results are present in the matrix, as these are necessary to fully define each Logic Gate distinctly from all other Logic Gates.

This template then replaces the original value, and similar process to that discussed under *Layout* and *Number Layout* in the previous section is applied.

The second function (with the suffix Ans) redefines a list of alternative options, i.e. *Answer Layout* and *Number Answer Layout*.

This function works by taking the number of answers determined in allPuzzleTypes, and the number of rules used.

For distributions, the function uses the number of rules to define minimum number of options that do not relate to the rules (anomalies), generating an even spread of alternative answers, generating the

anomalies and then inserting them in to the array. This is necessary as with fewer rules a larger number of anomalies becomes essential. It also uses anomalies to ensure that an even distribution of values is used, in order to avoid clues and anti-clues.

For addition and logic gates the functions calculate a set distribution of answers around the correct answer, with some systematic randomisation. For Logic Gates the function also factors in test item difficulty.

Examples

Both of the test items in Figure 3, and Figure 4 were generated with the same code, which is shown in Figure 5.

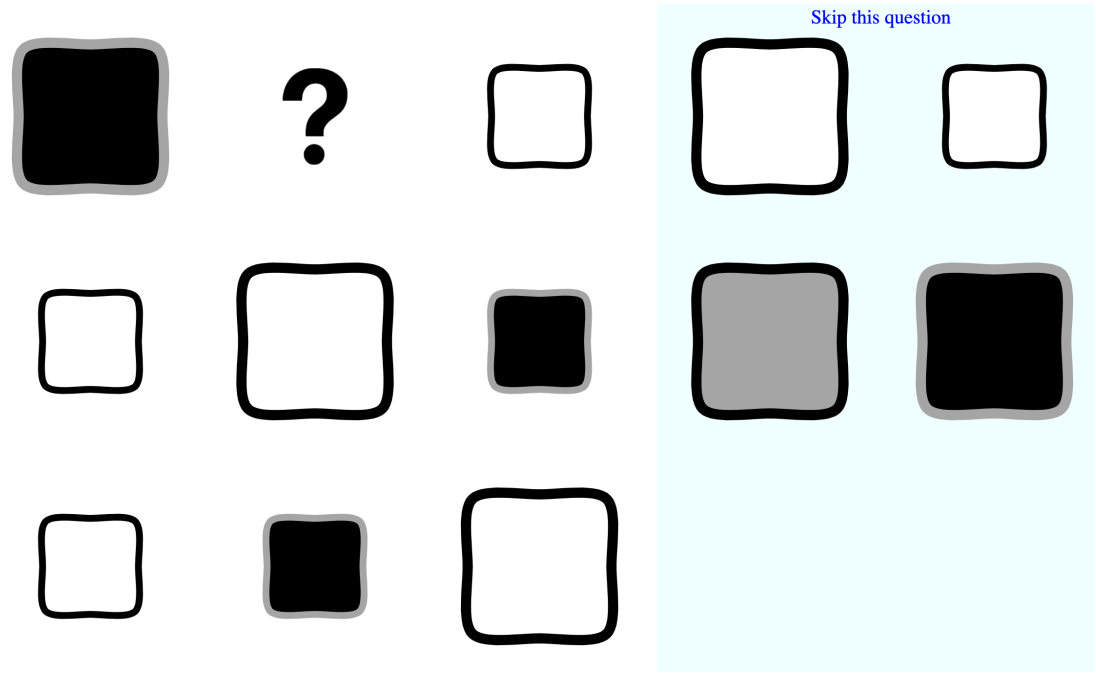


Figure 3: A screen shot of a test item with two distributions of 2.

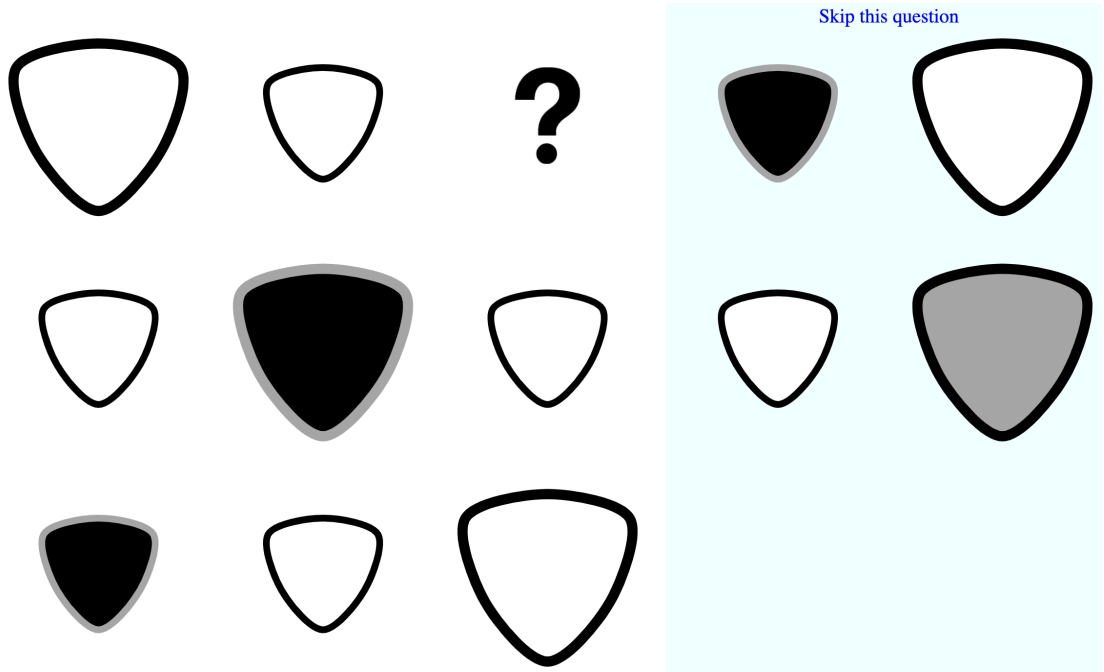


Figure 4: A screen shot with a different version of the same item as seen in Figure 1.

```
[[3,3],
   [0],
   [2],
   [1,[2,0,0]],
   [[0,[0,1,0]],[1,[1,0,0]],[0,[2,0,0]]],
   [0],
   [0,0,
    0,0],
   [0,0,0],
    0,
    0],
```

Figure 5: The AllPuzzleTypes element used to generate Figures 2 and 3.

If the functions had been turned off, this test item would have produced two distributions of 3 one working with size and the other with colour, randomly one of which would be horizontal and the other one vertical.

However the functions were active when generating the test items in Figures 2 and 3, but as the *Logic Option* was set to 2, the functions overwrite the data relevant to a Distribution of Three.

This test item was the second of ten test items in this test, so the layout or Form taken from annulusRuleArray is [2,[2,0,0]], rather than the value in allPuzzleTypes – as it happens, the only changes relate to the Forms. In annulusRuleArray, the value 2 indicates a diagonal Form, and as there are two of them, they are randomly assigned to different diagonal Forms.

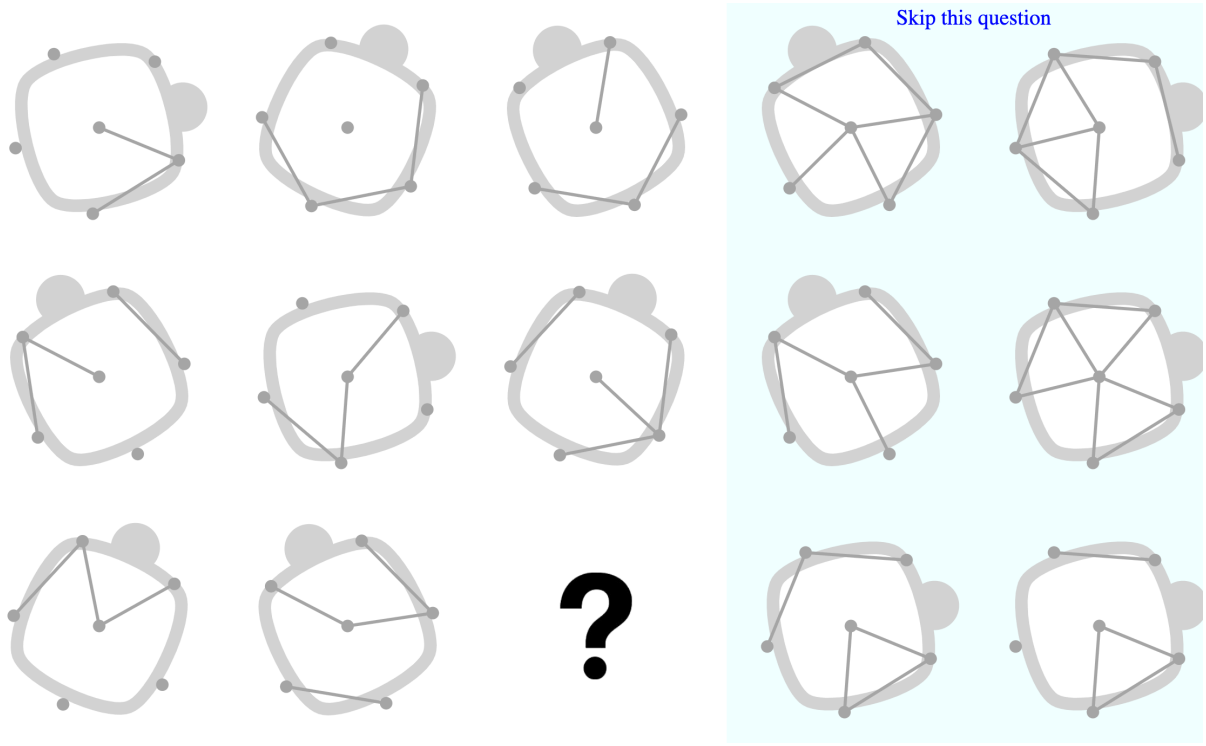


Figure 6: A screen shot of a test item combining the XOR Logic Gate with a Distribution of Three acting on rotation in an increasing diagonal Form. The example shown in Figure 6 was generated using the code shown in Figure 7.

```
[[3,3],
   [6],
   [6],
   [0,[0,0,4]],
   [[0,[0,0,1]],[0,[0,0,1]],[0,[0,0,0]],[0,[0,0,0]],[0,[0,0,0]]],
   [10],
   [[0,1,0,0,1,1,0,1,1,0],
    [0,0,0,1,0,1,0,1,0,0],
    [1,1,0,0,0,1,0,1,0,0],
    [0,1,0,1,1,1,0,0,0,1]],
   [[0,0,0,0,0,1,0,0,0,0],
    [1,0,1,0,0,0,0,0,0,0],
    [0,0,0,1,0,0,0,0,1,1],
    [1,0,1,0,0,0,0,1,0,0],
    [1,0,0,0,0,0,0,0,1,0]],
    0,
    0],
```

Figure 7: The allPuzzleTypes element used to define the test item in Figure 6.

Figure 6 shows one of the most complex test items used in the author's thesis.

Because the L*ogic Option* is set to the XOR Logic Gate, the functions only overwrite the Logic Gate portion of the array (*Number of Elements in Centre, Number Layout, and Number Answer Layout*); the Distribution of Three rules, form, and the relevant attributes of the option set are left as they are defined in Figure 7.

It can be seen from the number of rows in *Number Answer Layout*, that there will be 5 alternative options; together with the correct answer this gives the six options we see in Figure 6. Had the functions been turned off it would have been necessary to check that every Boolean set needed to define a Logic Gate, was incorporated into the visible elements of the matrix, as defined by *Number Layout*. As it is, the functions overwrite this portion of the array, and ensure that this is maintained.

Final Word

This manual is only a very brief introduction to editing a large application. If you would like to do anything this manual does not cover, or have any questions, please feel free to contact the author by opening an issue on the GitHub repository for Corvus https://github.com/Thimbleby/Corvus.