Static Symbolic Function Composition

Chris Dipple MSc

inet

Kaggle ARC_AGI Challenge, chris@dipple.org

Abstract

The Kaggle ARC_AGI prize was launched to encourage participation in creating computer systems to solve human Intelligence tests. This challenge is unlikely to be solved by simple memorisation of previous results, or by brute force methods. In the 5 years since the challenge was formulated no solution has approached human level performance. Francois Chollet [1] suggests the solution can most likely be found by functional composition. The proposed solution described here takes that approach, with some additional constraints to limit exponential search space growth. No AI system has so far demonstrated reasoning abilities across a wide range of problems. Without this ability to recognise general features of a problem then experiments to test a hypothesis is not how AI typically works, but it is how Science works. We discuss a mainly top down solution to this problem in this paper. Finding a robust generic solution to this category of reasoning problems would provide scaffolding on which other approaches to AI can relation. At no point are solutions remembered and reused. A search for common patterns is however used and solutions inspired by this. It might be the one approach that binds them all.

Introduction

This paper was written perhaps 20% of the way through a journey to implement a solution to the ARC-AGI challenge. It describes the research on computer search process, to solve the ARC-AGI problem. This paper is therefore a snapshot of what has been learnt so far, and how this informs future directions. The solution currently solves about 12% of the evaluation problems, but disappointingly only 1% of the hidden test set problems. Clearly there is a long way to go.

However, the speed of solving more problems as time goes by and richer representations of the problem space with reusable higher level functional composition is encouraging, and with sufficient effort there is no reason to believe this cannot continue to accelerate and solve most problems. It is estimated that approximately 6 months of effort is required to reach the 85% level. There is another source of problems, the code is written in Rust, not Python and hence some persuasion was needed to simply get it to run in Kaggle. There may be issues with this process as a higher percentage on the secret test run would be expected given the 12% evaluation score. Alternatively the functional breakdown and recomposition of solutions may be insufficiently generic. This was not the case between the test and evaluation sets where solving problems for either case often uplifts the scores with other similar tasks, in fact they are similar for both and as similar categories of problems in one data set were solved this was reflected with the other data set, as would be expecting with generalisation.

Future effort will be spent on solving more problems, in a data driven and generic manner. A number of helpful data structures have been developed which aid in the solving of problems as does a lengthening and improving set of composable functions based on them. Composable functions can often be improved too. For example recursive shape finding and the positioning of those shapes can be ambiguous from a computer point of view but presents few problems to the human visual system. In most cases there are remedies and creating then using them selectively can be further automated. This will be discussed in greater detail later.

Unresolved Issues

Not all categories of Shapes found within Grid have been identified yet. As have not all composable functions or alternative functions with a particular signature. Much more work is required to fill out these missing details.

Overview of future work

Address the unresolved issues. Ensure solutions are as generic as possible and share signatures. Any iterations applying a function to multiple Shapes or other objects should be wrapped in higher order functions, which in turn should be composable.

Methods

Background

The puzzles in ARC-AGI differ in a number of ways. Dimensionality, of shapes, the number of shapes within a grid, the sub-shapes (if any) within shapes and how those shapes differ and map onto answers, the mix of colours used, how shapes extend on another and many others features. These differences can be used as constraints to inform future effort and select from the set of composable functions available. These constraints describe a directed graph that in many instances is a simple tree. This reduces computation by approximately an order of magnitude. A top down approach has been pursued. The lower level functions are developed as needed to encapsulate specific actions, as data driven and parametrised entities. Assumptions about size, shape, colour, number and orientation etc. are not hardcoded but are discovered from context given the examples given (the experiments) and their answers.

The Scientific Method

The ARC-AGI challenge is presented as a scientific endeavour to be solved in a scientific manner. It is also based on human intelligence tests. So solving it using a scientific approach, deterministically, with some introspection on what humans might do when faced with the problems supplied was the guiding principle used to solve this challenge. This is unsurprisingly a huge enterprise, however approaching it as a stochastic problem to be solved with brute force did not seem right. That is not what humans do and we are attempting to emulate human thinking here. Feature design, constraint recognition and then function composition (or rather decomposition, then reassembly) would appear to be a better approach. There may be a place for simple neural new recognition of low level features, however, given the fact we are operating in a simple grid environment would suggest that is unnecessary here for now. For scenes in 'real' life it will almost certainly be required.

Science is first and foremost about Abductive reasoning. A hypothesis is framed, then tested and if it passes all tests it is considered a theory that can be used to answer similar questions. This is exactly the approach taken here.

Importantly, all reasoning should be sound and explainable. This is not the case with Neural nets. They have a place but that place in not in this context, at the reasoning level. The seductive qualities of a methodology to generate a black box universal function approximator is also extremely data hungry and few shot learning is mandated here.

Task Definition

This challenge is modelled on human intelligence tests, they are relatively easy to complete for most humans. They are visual reasoning tests using a 2-D grid with coloured pixels. These pixels for often form 'shapes' in the grid that are then transformed in some way to obtain an answer. No two tests are exactly there same, however, there are a number of common classes of transformations that can be observed in the data. These common transformations are common abstractions that can then be parametrised and solutions found. The common abstractions and the transformations that underlie them can only be applied when certain constraints are satisfied.

If it is possible to identify the abstraction classes, the special conditions under which they apply (which imply constraints) and the actual parametrised transformation required. Then it is possible to constrain combinatorial explosion and often solve similar problems too with parameterisation.

This process of identifying abstractions, transformations and constraints requires examination of different puzzles, creating solutions when new but similar puzzles are encountered with sufficient similarity to these already solved further parametrising the previous solutions to accommodate the new examples.

A common abstraction is to find shapes in the grid and in turn characterise those shapes This is not a simple process, some shapes touch other shapes, making them harder to identify. There may be a background supplied which can be used to separate these shapes. When there is not post processing may be required to identify touching shapes and separate them out. Shapes have attributes such as size of X and Y dimensions, colours (in fact shapes can be considered as either single coloured shapes of multi-coloured shapes). Shapes may in turn have additional internal structure, essentially shapes within shapes.

Shapes may be laid out in the enclosing grid relative to one another. There is also some ambiguity in this. For example some abstractions that require relatively placing of shapes to operate. Concepts of top right, top left, bottom right, bottom left might be required. These positions may not be simply discoverable from the X and Y coordinates of the shapes and require either a partitioning of the shapes relative to the position in the containing grid, and/or looser definition of position relative to one another (that can be used to sort them) achieved by a looser definition of the Y coordinate (assuming sorting is by X then Y coordinates).

So, the basic strategy has been to identify grid features and common constraints of common features. Then refine those features when necessary (touching shapes, sub-shapes, rel-

ative position etc). Then compose functions that satisfy these examples (the experimental cases). If and only if this works then an answer has probably be identified.

There is also a useful concept of closeness to a solutions. If some experimental examples are satisfied but not all then further refinement will often accommodate the failing examples. Also, reducing pixel difference count between examples and solutions can often indicate partial solutions too, and again further refinement may improve existing solutions to accommodate new cases.

A useful metric of generalisation is available. The ratio of fixed function transformations and problems solved. At the moment this stands at about 1.4, but the greater this number the more generalisation. Time will tell.

Datasets

There are 3 datasets available, training, evaluation and test. They all have a similar structure with the exception of the test set not having answers to the final test. The data describes a 2-D grid of varying sizes with each cell in the grid able to take on 9 different colours. An answer is some some other configuration of the grid derived from the initial grid. There are at least two experimental examples, with answers, given that follow a general principle. These examples can be thought of as experiments that are deterministic answers to a hypothesis. The challenge is to identify the hypothesis so it can be applied to the final test example. The final test in the test set does not have an answer, so the challenge is to derive this.

Motivation

The major motivate in addressing ARC-AGI was frustration after spending a year writing wrappers for LLMs, namely frustration with the quality and unpredictability of the results. It was hoped that it might be possible to use LLMs to generate code when given a code framework and a definition of individual puzzles. This helped a little in some cases but on the whole was slower than hand coding functions and function compositions. This may be an issue with LLM training being less with Rust than other languages like Python. However much richer data-structures and 2 orders of magnitude execution speed increase allow for much more productivity in Rust than Python. Not to mention fewer bugs and more elegant and satisfying code.

Data Structures

The representation of the data for the challenge is important. At the top level is a Grid, composed of Cells which each have a position and colour within the Grid. A Grid can be decomposed into a sequence of Shapes that are again composed of a subset of the Cells within the Grid (with position relative to their position in the Grid).

At each level the object in question is categorised. Grids are categorised by global shape (size, square, rectangle, etc), where this applies across all examples. Similarly for shapes with some additional relational categorisation. Cells are categorised by colour and their position within a shape. This global categorisation is meant to be fast and is used to categorise Objects in a directed graph that can be quickly navigated so various transformations are only attempted when appropriate, reducing search space. This results so far in an approximate 5 fold improvement in execution speed. This is expected to improve as more cases are solved.

The categorisation can optionally be improved with additional predicates that help improve the differentiation between different cases. For example the number of Shapes within a Grid is quick to calculate and so is always done. However, the predicate to identify whether all shapes have the same size, or their relative orientation (which can be fuzzy), is only calculated on demand. Shape identification is not perfect on the initial pass, for example touching shapes are initially identified as single shapes and again some additional processing is done on demand.

Categorisation rarely results in a single possible transformation. In some cases, for example when shapes are rotated or mirrored so a sequence of transformations can be tried and the one that works for all examples then applied to create the final answer. Sometimes the appropriate transformation may be identifiable by categorising the Example to Solution mapping.

The above is used to create a pipeline of categorise, filter, further process and/or further categorise, filter, try variants. Primitive transformations can also be twinned with other transformations to build higher level transformations. This is a process of discovery, at the moment largely static and without long range backtracking. This will likely change with next years iteration of the ARC-AGI competition.

Use Case Examples

Here we discuss some example for particulate cases and algorithms use to solve them.

<u>Black patches on an otherwise fully populated Grid</u>

Preconditions: There are no identifiable shapes smaller than grid size and hence overlapping. There are rectangular areas that are black.

Identify: For Black rectangular areas. Find the pixels surrounding each rectangular area. If it touches an edge of a grid then any colour is permissible along that edge.

Find: other places in the triangular grid that have the same surrounding pixel colours. If more than one, then the internal colour must be in the same order and position.

Do: fill in the interior of the Black rectangle with the colours found.

Extensions: Rectangle can be any colour, not just Black. A single area fully populated with non-rectangular colour pixels may not be possible. If so find areas, order by largest and fill in each partial answer.

Identify a shape by some criteria

Preconditions: A Grid may contains shapes (in fact most do). Shapes are contained by a rectangle and identified by finding reachable coloured pixels (either the same colour or any colour other that the background colour [normally Black]). Occasionally rectangles touch and under some circumstances can be further separated (when dimensions are known). Second order Shapes may also sometimes be further embedded in first order shapes. Identify the number of shapes in the output of training samples can aid categorisation and help target solutions.

Identify: Identified shapes can be ordered, classified by size, colour, or shape (arms, hollows etc.), orientation, uniqueness, count and pixel positions.

Find: The differentiating features of the shapes found. Often single pixel ‘shapes’ are noise. Multiple shapes of the same size, or colour, or pixel layout may have some relationship with one another. Finding this set of relationships is a major guide to a solution. Knowing the output shapes and their relationships to the input shapes is very useful.

Do: May operations are possible. Single shape grids may simply be turned back into a grid (eliminating surrounding background pixels). Or there may be some transformation of the shape, for example rotation or mirroring. Shapes (often with non-black separating colours) may be combined in different ways (so there would be one output shape, of the same size as the input shapes) Or/And/Xor or overlay in a particular order (a permutation of the number of input shapes) etc. Shapes may be given surrounds, which are often other shapes. There are many possibilities. Again, find these possibilities is crucial to both containing the explosion of seach space and creating generic composable functions.

Extensions: Simply continue to grow the functional ways in which shapes can be combined or otherwise enhanced. Categorisation can help a great deal here in fighting unchecked search space growth. This is a very generic category and can be further sub-categorised in a large number of ways.

Blank Grids

Preconditions: Grids composed with a single uniform colour in every position, normally Black. Under these circumstances it is essential that the target pattern in the output Grid is identified. This might be borders, spirals and possibly other patterns.

Identify: The output pattern. As this set is large then only those seen already can be anticipated, with some simple variants such as direction of the spiral or colour.

Find: The actual output pattern, then scale it for the Grid size (the same as the input size).

Do: If a shape was identified then draw it to scale and with the correct colour (the same as the training examples).

Extensions: There are not many of these, but variants of the shapes seen are probably most likely to be found in the private evaluation set. If they are not, then this is not a very good test!

Indexed Shapes

Preconditions: There are a number of Shapes that approximately in a regular grid pattern within the enclosing Grid.

Identify: Parse the Grid to find Shapes. The size of the Shapes will normally be the same. The origin of each shape will have at least one other Shape with a similar X position and at least one other Shape with a similar Y position.

Find: An ordering for the Shapes in the Grid that minimises the distance from the origin of the grid.

Do: Create an output Grid, that might have a single colour pixel with the same number pixels in the output Grid as the number of Shapes in the input Grid. The output pixel being coloured the same as the input shape.

Extensions: The shapes may vary in size, providing the relative positioning of the Shapes is the same. The output grid may be a single line in the X or Y axis. If the Shapes are not a single colour then this arrangement might be dependant on the majority (or even minority) colour in the Shapes (in which case all Shapes would be mixed colours). Many arrangements are possible, but in all cases constrained by categorical features of the Shapes and the relationship between the input and output Shapes. Of course the assumption is a relationship found in the experimental examples also follows for the final solution.

Joined Shapes

Preconditions: Shapes can be found currently by finding a non-background cell then finding all the cells, of either the same colour or cells that are simply non-background that are reachable (normally diagonally too). The rectangular extent of the cells found is considered a Shape (single coloured or multi-coloured). This works reasonably well, but there are problems, such as when the background colour is not Black. So a background colour discovery process is needed too. Shapes may also touch one another. This can be detected if the majority of the shapes found for every experimental example for a task is a rectangle with one of the dimensions the same as twice the same dimension of the majority. Additional parsing can separate the shapes.

Identify: There are Shapes in the Grid. For each shape if it is anomalous then apply further processing.

Find: Now we have Shapes of uniform dimensionality and colouration, feed these Shapes into the normal Shapes processing pipeline. Further categorisation of said Shapes will guide this process as normal.

Do: Further process Shapes according to variants know and in turn passed this data one to compose functions appropriately.

Extensions: Many are possible. This is largely a process of slow discovery of new arrangements and the corresponding functions required to produce the solutions. Two things are happening here, higher order functions for common groups of transformations are discovered and then used in other similar situations. Also additional lower level functions are discovered which previously were not known. Most functions have the same signatures so composition is easy. Iteration over multiple functions at a level of abstraction is carried out with higher order functions that take a list of more basic functions.

Enhanced Shapes

Preconditions: Shapes can be identified that have some common attribute such as colour and/or size. There is another shape that can visibly surround the shape, or alternatively additional pixels on the example solutions that surround the shape in some consistent way.

Identify: Identify the Shapes and the pattern that surrounds the shapes.

Find: Replicate the pattern found around each shape.

Do: Create a solution for the test with Shapes surrounded appropriately.

Extensions; The surrounding patterns may be lines going horizontally, vertically of diagonally to other Shapes or to the edges of the enclosing Grid. Lines that cross may change the colour of the crossing point, or overlay the previous colour.

Other Shape based Transformations

Preconditions: Shapes exist, that are not covered by the previous categorisation of shapes. This is an evolving task and is in fact the primary missing piece from the existing submission.

Identify: What makes the grid and it’s contained shapes different?

Find: Parametrise instances of this category and enumerate the possibly generic transformations that might transform it into the desired answer.

Do: Carry out the discovered transformation on all task examples. If they agree with the experimental answer, then generate a candidate solution.

Extensions: This the biggest category so far. Finding Shapes is a prerequisite. The relative characteristics of a Grid/Set of Shapes and the relationship between the shapes (and the output shapes in the solved examples) will occupy the majority of the effort between now and the next iteration of this prize. There are subcategories of solutions too. A solution may require a unique sequence of functional compositions, or it may be general enough to be found with some search over a set of transformations at some level (seen before), or it may be capable of transforming more than one task into a solution because the intermediate transformations are sufficiently generic. Most effort will be spent on the second and third cases. There are already cases where unique solutions have been subsumed by more generic code.

There are many other generic patterns.

Assumptions

There are some fundamental assumptions in this work that bare mentioning:

- The search space within a Grid is not random. There a patterns that a human brain can identify and extrapolate from. This constrains the search space.
- In a two dimensional space, there are only so many ways in which it can be partitioned and be 'meaningful' to a human.
- There are only so many patterns that make sense to humans.
- Patterns can be distinguished in only so many ways. Sub-areas of the Grid form what is called here Shapes. These can be negative Shapes using what might be a background colour superimposed on a more noisy background. Patterns repeating colours in lines going up/does, left/right or along diagonals my occur too. Identifying these patterns and testing for some permutation of them in the example solutions is a good guide to a solution.
- Using features of the space the partitioning can be a strong guide to how the Grid and the objects identified in the Grid relate to one another. Again constraining search space.
- Find the 'right' partitioning and the 'right; functions to compose (and the composition ordering) is the main task required to solve this challenge.

Results

Metrics

With ARC-AGI the only real metric is how many tasks in the secret hold out set are passed. This might be considered both a failing and a strength. So there is only an accuracy score, there is no concept of Precision/Recall or F1 score. It is hence difficult to determine how 'close' a score it to the correct solution, or even how many examples/experiments with solutions worked with a particular task.

For this reason new metrics were invented.

A primary metric of success is will a particular function closure solve more than one problem. So what is the ratio of solutions to problems answered. This is currently about 1.4, so for each closer approximately 1.4 problems are solved. This is likely to improve greatly with more effort and time spent refining the constraints, lower level functions, parametrisation from training data and other factors. As with many problems there are only so many ways things are actually done, entropy does not tend to increase when things are transformed by intelligent systems, even if the possible space is much larger. This can be used as an additional constraint on likely outcomes as can decreasing pixel count difference given experimental examples.

The training data set tends be simpler, there are generally only one or two features that need to be modelled. The evaluation set has more possibilities. So a second metric is how many training solutions result in how many evaluation set solutions. This was disappointing when first tried, very few being the answer, but developing evaluation set answers then reapplying them to the training data tended to yield better conversion rates. This is the reverse of what the competiton authors expected or wanted, but works well for the approach taken here given the public data.

Another metric is pixel cover. For a candidate solution if a series of transformations resulting a in monotonically decreasing number of pixel differences between the given solution and the progress so far. This again can be used to determine the efficacy of a transformation, if the pixel different count increases then the transformation is probably not useful and another transformation can be applied. This is not a perfect guide but nevertheless useful.

What is a little confusing is this does not work as well as expected with the secret holdout test set. There are two plausible explanations for this, firstly that the approach is simply flawed. Alternatively that the JSON generated for the solutions file is flawed in some way or there is some other issue that is not flagged as an error but nevertheless is poisoning the solutions generated somehow. Some time was spent trying to find the closure that actually resulting in a score. Given the code is deterministic a binary chop of the results was undertaken. This did not find the 'good' closure, even worse if half the file is forced to return a dummy and hence wrong result and this test is run twice on distinct and separate halves of the closures, then it will produce a 1.00 on both halves. This is nonsense. A possible solution is to do the I/O in python and pass solutions back from Rust to python. That is a lot of unanticipated and what should be unnecessary work with no guarantee of success. The Rust file, and this has been checked multiple times, does not generate scoring errors and is formatted correctly. However, it does not contain white space after commas for example, which is a difference but is still valid JSON.

Applicability of ML Theory

This solution to the ARC-AGI prize is essentially Symbolic AI with constraints to limit search space explosive growth. Constraints are applied in order of 'cost' simple constraints roughly partition the search space. Within each subspace further constraints can be applied (for example we have found N shapes of equal size). This is recursive (there may be an even number of shapes, with a particular orientation to one another and a similar number of shapes [or trivial shapes, I.e pixels] in the output example solutions).

The constraints ideally would form a tree structure. At the moment there appears to be at best a directed graph. So categorisation may need to shape terminal nodes (which are function closures containing a sequence of parametrised and generic transformations). This is not necessarily a bad thing as it suggests there is either a better categorisation possible and/or the function closure is generic enough to span categorisations.

Within function closures a it is possible to call additional functions (all with the same signature) that take a lower level function and apply possible transformations. This is also a fruitful area and it is hoped this can be expanded beyond the And/Or/Xor, Overlay order, rotations and mirroring currently catered for.

Comparison of Results

To Do:

1. Improve categorisation. Possibly semi-automating it with a search through constraint space at a number of levels and grouping categorisations. This would help generate code statically as a first pass, then possibly be extended later, if necessary, to an interpreted approach operating on a new and unknown set of Shapes. It is currently thought that a static search may be sufficient as the novel permutations of possibilities will diminish as there are only so many ways of doing things. Either way, static categorisation will greatly trim search space with little run time overhead.
2. Improve Shape recognition. Shapes may touch, overlap and contain sub-shapes, improve it's discrimination and discovery capabilities.
3. Improve Shape positioning and ordering recognition. At the moment shapes are only sorted by X or Y axis, this means that if there are one or a small number of positional differences in one or both of these axis, they are not sorted in a useful order. Fuzzy positioning and sorting is required, for example when Grid filling with derived shapes for an example solution.
4. Additional grouping of functions with the same signature that can be applied at some point in the sequence of functional compositions. Possibly Shapes containing other shapes and extensions of shapes to touch or cross other shapes possibly to the Grid borders. Also transposing of shapes to the edges of a Grid or some other constraining Shape.
5. Produce more intermediate results that are a step towards the answer and can in turn be composed.
6. Take examples and generate more 'similar' examples and use a learning algorithm. Thos would be a last resort.
7. Further Identify the degrees of freedom, of objects in the object hierarchy. Colour(s), position (relative to peers and relative to other levels), size, orientation, colour, …).
8. Relative distance between objects. Ratios of all measurements.
9. Possible need for backtracking when potential solutions do not work. Only needed if categorisation does not result in a tree with single and uniuque terminal nodes (but is a directed graph) and trying of multiple solutions has a significant performance overhead.

Discussion

Significance of Results

To date only about 12% of the tasks are solved by this approach. It was slow to start but is now accelerating as better categorisation is applied and improved. Functions are improving and becoming more generic. So solving new cases is often relatively simple, or they further inform existing code to make it more generic. There are still novel problems to solve but with the expectation that the same reasoning applies and these new cases will be solved in an accelerating manner.

How to improve results

Better categorisation and automated categorisation. More iteration over functions with same signature. Better parametrisation of existing function (no numeric constants anywhere). More guidance from example solutions. 'Fuzzy' relative positioning of Shapes. Better shape parsing (touching/overlapping)

Limitations

There is no reason to assume that the methodology outlined in this paper will not be able to solve all of the problems in this competition. However, the natural extension of this is to identify 2 dimensional Grids with very large pixel counts in the real world and then 3 dimensional Objects. Identifying low level Shapes, composing them then identifying higher level Shapes is probably a sensible strategy. However it will be orders of magnitude harder to achieve in the real world so new tools will likely be required. There will also likely be a place for Neural Net based low level object recognition too as part of those tools. The usefulness of LLMs is questionable.

References

Francois Chollet, Mike Knoop, Bryan Landers, Greg Kamradt, Hansueli Jud, Walter Reade, Addison Howard. (2024). ARC Prize 2024. Kaggle. https://kaggle.com/competitions/arc-prize-2024

Code Examples

Mirrored in X and Y

```
let func = |ex: &Example| {
    // Ensure there is one and only one Shape in the Grid
    if ex.input.coloured_shapes.len() != 1 {
        return Grid::trivial();
    }

    // Turn the Grid into a Shapes struct
    let s = ex.input.grid.as_shape();

    let rows = s.cells.rows;
    let cols = s.cells.columns;
    // Double both X and Y dimensions
    let mut shapes = Shapes::new_sized(rows * 2, cols * 2);

    // Make four copies of the original shape with appropriate mirroring
    shapes.shapes.push(s.mirrored_x().mirrored_y());
    shapes.shapes.push(s.mirrored_y().translate_absolute(rows, 0));
    shapes.shapes.push(s.mirrored_x().translate_absolute(0, cols));
    shapes.shapes.push(s.translate_absolute(rows, cols));

    // Turn Shapes back into a Grid for the final answer
    shapes.to_grid()
};

// Run experiment over all example experiments for above closure
if run_experiment(&file, 280, is_test, &example, &targets, &mut done, &func,
&mut output) { continue; };
```

Possible improvements would be to calculate the difference in size between input and output and then create an appropriate number of output subshapes in correct dimensions. Also discover the translations required from the example experiments and apply those, not a fixed set of transformations.

I/O mapping with some attribute removed but using common mapping

```
// Cross example knowledge needed for closure. This takes input Grid examples,
// removes colour data then maps it to a solution Grid. The bleached map is
// indexed as json as that is a simple way to capture the distribution
// of Pixels while bleaching removes colour differences.
// The bleached map can then be used trivially to answer the experiments but
// more importantly the same structure, but not colour, is used by the
// final solution.
// NOTE: We need to be very careful here that the examples are categorised
// correctly or false solutions might result as we would just be replicating what
// exists already and the experiments framework gives us no useful information.
let h = example.bleached_io_map();

let func = |ex: &Example| {
    if let Some(grid) = h.get(&ex.input.grid.bleach().to_json()) {
        grid.clone()
    } else {
        Grid::trivial()
    }
};

if run_experiment(&file, 221, is_test, &example, &targets, &mut done, &func,
&mut output) { continue; };
```

Apply set of functions with same signature to find answer to all experiments

```
// Function that iterates over a set of functions
pub fn gravity_only(grid: &Grid, n: &mut usize) -> Grid {
    let func = [Grid::stretch_down, Grid::gravity_down, Grid::gravity_up];
    if *n == usize::MAX {
        *n = func.len();
    }
    if *n == 0 {
        Grid::trivial()
    } else {
        func[func.len() - *n](grid)
    }
}

if run_experiment_tries(&file, 350, is_test, &example, &targets, &mut done, &|ex,
_, n| gravity_only(ex, n), &mut output) { continue; };
```

Possible improvements are to add additional transformations

Create array of colours from the coloured shapes in input array with size > 1

```
// Find dimensions and colour for outpur, from example output
let x = example.examples[0].output.grid.cells.rows;
let y = example.examples[0].output.grid.cells.columns;

// One dimension should be 1
if x == 1 || y == 1 {
    let colour = example.examples[0].output.grid.colour;

    let func = |ex: &Example| {
        if x == 0 || y == 0 {
            return Grid::trivial();
        }
        let mut i = 0;
        let mut grid = Grid::new(x, y, Colour::Black);

        for s in ex.input.shapes.shapes.iter() {
            if i >= y { // Make sure we don't exceed output size
                return Grid::trivial();
            }
         // If colour and size correct then add to output
         if s.colour == colour && s.size() > 1 {
             if x == 1 { // change depending on dimensionality
                 grid.cells[(0, i)].colour = colour;
             else {
                 grid.cells[(i, 0)].colour = colour;
             }
             i += 1;
         }
        }

        grid
    };

    if run_experiment(&file, 400, is_test, &example, &targets, &mut done, &func,
&mut output) { continue; };
}
```

Output public test run

cargo run --release test

007bbfb7: {Div9In, IdenticalNoColours, InLessThanOut, InOutSquare, Is3x3In, Is3x3Out, SameColour, SingleColourIn, SingleColourOut}
Success: 00130 / 007bbfb7
00d62c1b: {BGGridInBlack, BGGridOutBlack, EvenRowsIn, EvenRowsOut, InOutSquareSameSize, InOutSquareSameSizeEven, SingleColourIn}
017c7c7b: {EvenRowsIn, IdenticalNoColours, InLessThanOut, InOutSameShapes, InOutSameShapesColoured, InOutShapeCount, Is3x3In, Is3x3Out, SingleColourIn, SingleColourOut, SingleShapeIn, SingleShapeOut}
025d127b: {EvenRowsIn, EvenRowsOut, IdenticalColours, InOutSameShapes, InOutSameShapesColoured, InOutSameSize, InOutShapeCount}
045e512c: {BGGridInBlack, BGGridOutBlack, IdenticalNoColours, InLessCountOut, InOutSquareSameSize, InOutSquareSameSizeOdd, Is3x3In, Is3x3Out}
0520fde7: {Div9Out, In7x3, Is3x3Out, OutLessThanIn, OutSquare, SingleColourOut}
Success: 00320 / 0520fde7
05269061: {FullyPopulatedOut, IdenticalNoColours, InOutSquareSameSize, InOutSquareSameSizeOdd, SingleColouredShapeOut}
05f2a901: {IdenticalColours, InOutSameShapes, InOutSameSize, InOutShapeCount, InSameCountOut, SingleColouredShapeOut}
06df4c85: {BGGridInBlack, BGGridInColoured, BGGridOutBlack, BGGridOutColoured, IdenticalNoColours, InOutSameShapes, InOutSameShapesColoured, InOutShapeCount, InOutSquareSameSize, InSameCountOut}
08ed6ac7: {BGGridInBlack, BGGridOutBlack, IdenticalNoPixels, InOutSameShapes, InOutSameShapesColoured, InOutShapeCount, InOutSquareSameSize, InOutSquareSameSizeOdd, InSameCountOut, Is3x3In, Is3x3Out, SingleColourIn}
09629e4f: {BGGridInBlack, BGGridInColoured, BGGridOutBlack, BGGridOutColoured, InOutSameShapes, InOutSameShapesColoured, InOutShapeCount, InOutSquareSameSize, InOutSquareSameSizeOdd, InSameCountOut}
Success: 00050 / 09629e4f
0962bcdd: {BGGridInBlack, BGGridOutBlack, EvenRowsIn, EvenRowsOut, IdenticalNoColours, InOutSameShapes, InOutSameShapesColoured, InOutShapeCount, InOutSquareSameSize, InOutSquareSameSizeEven, InSameCountOut, Is3x3In, Is3x3Out}
0a938d79: {IdenticalNoColours, InLessCountOut, InOutSameSize}
0b148d64: {OutLessCountInColoured, OutLessThanIn, SingleColourOut}
Success: 00222 / 0b148d64
0ca9ddb6: {BGGridInBlack, BGGridOutBlack, InOutSameShapesColoured, InOutSquareSameSize, InOutSquareSameSizeOdd, InSameCountOutColoured, Is3x3In, Is3x3Out}
0d3d703e: {Div9In, Div9Out, FullyPopulatedIn, FullyPopulatedOut, IdenticalNoColours, InOutSameShapes, InOutSameShapesColoured, InOutShapeCount, InOutSquareSameSize, InOutSquareSameSizeOdd, InSameCountOut, Is3x3In, Is3x3Out, MirrorXIn, MirrorXOut, SingleColouredShapeIn, SingleColouredShapeOut}
Success: 00251 / 0d3d703e
0dfd9992: {BlackPatches, FullyPopulatedOut, IdenticalNoColours, InOutSameShapesColoured, InOutSquareSameSize, InOutSquareSameSizeOdd, InSameCountOutColoured, Is3x3In, Is3x3Out, SingleColouredShapeIn, SingleColouredShapeOut}
0e206a2e: {EvenRowsIn, EvenRowsOut, IdenticalNoColours, InOutSameSize, OutLessCountInColoured}
10fcaaa3: {Double, EvenRowsOut, InLessCountOut, InLessThanOut, SingleColourIn}
Success: 00190 / 10fcaaa3
11852cab: {BGGridInBlack, BGGridOutBlack, EvenRowsIn, EvenRowsOut, IdenticalNoColours, InOutSquareSameSize, InOutSquareSameSizeEven, SingleColouredShapeOut}
1190e5a7: {BGGridInBlack, BGGridInColoured, FullyPopulatedIn, FullyPopulatedOut, InSquare, MirrorXOut, MirrorYOut, OutLessCountInColoured, OutLessThanIn, SingleColourOut, SingleShapeOut, SymmetricOut}
137eaa0f: {BGGridInBlack, Div9Out, IdenticalNoColours, InOutSquare, Is3x3Out, OutLessCountInColoured, OutLessThanIn, SingleColouredShapeOut}
150deff5: {IdenticalNoPixels, InOutSameShapesColoured, InOutSameSize, SingleColourIn}
178fcbfb: {IdenticalNoColours, InLessCountOut, InOutSameSize, SingleColouredShapeOut}
1a07d186: {InOutSameSize, OutLessCountInColoured}
1b2d62fb: {OutLessThanIn, SingleColourOut}
1b60fb0c: {BGGridInBlack, BGGridOutBlack, EvenRowsIn, EvenRowsOut, InLessCountOut, InOutSameShapesColoured, InOutSquareSameSize, InOutSquareSameSizeEven, SingleColouredShapeOut, SingleColourIn, SingleShapeIn}
1bfc4729: {BGGridInBlack, EvenRowsIn, EvenRowsOut, IdenticalNoColours, InOutSameShapes, InOutShapeCount, InOutSquareSameSize, InOutSquareSameSizeEven, InSameCountOut, MirrorYOut, SingleColouredShapeOut}
1c786137: {OutLessThanIn}
1caeab9d: {IdenticalColours, InOutSameShapes, InOutSameSize, InOutShapeCount, InSameCountOut}
1cf80156: {IdenticalColours, InOutSameShapes, InOutSameShapesColoured, InOutShapeCount, OutLessThanIn, SameColour, SingleColourIn, SingleColourOut, SingleShapeIn, SingleShapeOut}
Success: 00200 / 1cf80156
1e0a9b12: {GravityDown, IdenticalColours, InOutSquareSameSize, OutLessCountInColoured}
Success: 00350 / 1e0a9b12
39a8645d: {BGGridInBlack, Div9Out, EvenRowsIn, InOutSquare, Is3x3Out, MirrorXOut, OutLessCountInColoured, OutLessThanIn, SingleColourOut, SingleShapeOut}
39a8645d 20 : 1 worked out of 3
Success: 00030 / 39a8645d
39e1d7f9: {BGGridInBlack, BGGridInColoured, BGGridOutBlack, BGGridOutColoured, IdenticalNoColours, InOutSameShapes, InOutSameShapesColoured, InOutShapeCount, InOutSquareSameSize, InSameCountOut, Is3x3In, Is3x3Out}
3aa6fb7a: {BGGridInBlack, BGGridOutBlack, InOutSameShapesColoured, InOutSquareSameSize, InOutSquareSameSizeOdd, InSameCountOutColoured, SingleColourIn}
3ac3eb23: {EvenRowsIn, EvenRowsOut, IdenticalNoColours, InOutSameShapes, InOutSameShapesColoured, InOutSameSize, InOutShapeCount}
3af2c5a8: {Double, EvenRowsOut, IdenticalNoColours, InLessThanOut, InOutSameShapes, InOutSameShapesColoured, InOutShapeCount, MirrorXOut, MirrorYOut, SameColour, SingleColourIn, SingleColourOut, SingleShapeIn, SingleShapeOut, SymmetricOut}
3bd67248: {InLessCountOut, InOutSquareSameSize, InOutSquareSameSizeOdd, MirrorXIn, SingleColouredShapeOut, SingleColourIn}
3bdb4ada: {EvenRowsIn, EvenRowsOut, IdenticalNoColours, InOutSameShapes, InOutSameShapesColoured, InOutSameSize, InOutShapeCount, InSameCountOut}
3befdf3e: {BGGridInBlack, BGGridOutBlack, EvenRowsIn, EvenRowsOut, IdenticalNoColours, InOutSameShapesColoured, InOutSquareSameSize, InOutSquareSameSizeEven, InSameCountOutColoured, SingleColouredShapeIn, SingleColouredShapeOut}
Success: 00050 / 3befdf3e
3c9b0459: {Div9In, Div9Out, FullyPopulatedIn, FullyPopulatedOut, IdenticalColours, InOutSameShapes, InOutSameShapesColoured, InOutShapeCount, InOutSquareSameSize, InOutSquareSameSizeOdd, InSameCountOut, Is3x3In, Is3x3Out, Rot180, SingleColouredShapeIn, SingleColouredShapeOut}
Success: 00000 / 3c9b0459
3de23699: {OutLessCountInColoured, OutLessThanIn, SingleColourOut}
3e980e27: {BGGridInBlack, BGGridOutBlack, IdenticalNoColours, InOutSameShapesColoured, InOutSquareSameSize, InOutSquareSameSizeOdd, InSameCountOutColoured}
3eda0437: {InLessCountOut, InOutSameSize}
3f7978a0: {IdenticalNoColours, OutLessCountInColoured, OutLessThanIn}
40853293: {EvenRowsIn, EvenRowsOut, IdenticalNoColours, InOutSameSize, OutLessCountInColoured}
Success: 00051 / 40853293
4093f84a: {EvenRowsIn, EvenRowsOut, IdenticalNoPixels, InOutSquareSameSize, InOutSquareSameSizeEven, OutLessCountInColoured, SingleColourOut, SingleShapeOut}
41e4d17e: {BGGridInBlack, BGGridInColoured, FullyPopulatedIn, FullyPopulatedOut, IdenticalNoPixels, InLessCountOut, InOutSquareSameSize, InOutSquareSameSizeOdd, Is3x3In, Is3x3Out, SingleColouredShapeOut}
4258a5f9: {BGGridInBlack, BGGridOutBlack, InOutSquareSameSize, InOutSquareSameSizeOdd, Is3x3In, Is3x3Out, SingleColourIn}
4290ef0e: {IdenticalNoColours, MirrorXOut, MirrorYOut, OutLessThanIn, SingleColouredShapeOut, SymmetricOut}
42a50994: {IdenticalNoColours, InOutSameSize, OutLessCountInColoured, SameColour, SingleColourIn, SingleColourOut}
4347f46a: {IdenticalNoColours, InOutSameShapes, InOutSameShapesColoured, InOutSameSize, InOutShapeCount}
444801d8: {BGGridInBlack, BGGridOutBlack, EvenRowsIn, EvenRowsOut, IdenticalNoColours, InOutSameShapes, InOutShapeCount, InOutSquareSameSize, InOutSquareSameSizeEven, InSameCountOut}
445eab21: {BGGridInBlack, EvenRowsIn, EvenRowsOut, FullyPopulatedOut, InOutSquare, OutLessThanIn, SingleColourOut, SingleShapeOut}
Totals: {BGGridInBlack: 13, BGGridOutBlack: 29, BlackPatches: 4, BlankIn: 1, Div9In: 5, Div9Out: 3, Double: 2, FullyPopulatedOut: 13, GravityDown: 2, IdenticalNoPixels: 8, InOutSameShapes: 29, InOutSameSize: 20, InOutSquare: 3, InOutSquareSameSize: 50, Is3x3In: 18, MirrorXOut: 6, OutLessThanIn: 36, SingleColouredShapeOut: 22, SingleColourIn: 15, SingleColourOut: 10, SingleShapeOut: 16}
Complete: {BGGridInBlack: 2, BGGridOutBlack: 2, BlackPatches: 1, BlankIn: 0, Div9In: 1, Div9Out: 1, Double: 1, FullyPopulatedOut: 0, GravityDown: 2, IdenticalNoPixels: 0, InOutSameShapes: 0, InOutSameSize: 1, InOutSquare: 1, InOutSquareSameSize: 3, Is3x3In: 0, MirrorXOut: 0, OutLessThanIn: 6, SingleColouredShapeOut: 1, SingleColourIn: 0, SingleColourOut: 0, SingleShapeOut: 3}
cnt = 75, tries = 305, done = 25

{"007bbfb7", "0520fde7", "09629e4f", "0b148d64", "0d3d703e", "10fcaaa3",
"1cf80156", "1e0a9b12", "1f85a75f", "1fad071e", "2013d3e2", "22eb0ac0",
"239be575", "25ff71a9", "27a28665", "2dc579da", "31aa019c", "3428a4f5",
"3618c87e", "363442ee", "3906de3d", "39a8645d", "3befdf3e", "3c9b0459",
"40853293"}

So, there where 305 experiments carried out checking the input and output of examples and then ifand only they are correct generating a sumbission file. Each task lists the higher level catagories found for the Grid. Of which 25 succeeded and 75 failed. This took 144 milliseconds on a single core of an i9.

Github Source Code

Cratesio: https://crates.io/search?q=arc-agi

Github: https://github.com/intelligentnet/arcagi