First Steps of an Approach to the ARC Challenge based on Descriptive Grid Models and the Minimum Description Length Principle

Sébastien Ferré

Univ Rennes, CNRS, IRISA
Campus de Beaulieu, 35042 Rennes, France
Email: ferre@irisa.fr

Abstract. The Abstraction and Reasoning Corpus (ARC) was recently introduced by François Chollet as a tool to measure broad intelligence in both humans and machines. It is very challenging, and the best approach in a Kaggle competition could only solve 20% of the tasks, relying on brute-force search for chains of hand-crafted transformations. In this paper, we present the first steps exploring an approach based on descriptive grid models and the Minimum Description Length (MDL) principle. The grid models describe the contents of a grid, and support both parsing grids and generating grids. The MDL principle is used to guide the search for good models, i.e. models that compress the grids the most. We report on our progress, improving on the general approach and the models. Out of the 400 training tasks, our performance increased from 5 to 94 solved tasks, only using 30s computation time per task. Our approach not only predicts the output grids, but also outputs an intelligible model and explanations for how the model was incrementally built.

1 Introduction

This document describes our approach to the ARC (Abstraction and Reasoning Corpus) challenge introduced by François Chollet as a tool to measure intelligence, both human and artificial [2]. ARC was introduced in order to foster research on Artificial General Intelligence (AGI) [5], and to move from system-centric generalization to developer-aware generalization. The latter enables a system to "handle situations that neither the system nor the developer of the system have encountered before". For instance, a system trained to recognize cats features system-centric generalization because it can recognize cats that it has never seen, but in general, it does not feature developer-aware generalization because it can not learn to recognize horses with only a few examples (unlike a young child would).

ARC is made of 1000 learning tasks, 400 for training, 400 for evaluation, and 200 kept secret by its author in order to ensure bias-free evaluation. Each task consists in learning how to generate an output colored grid from an input colored grid (see Figure 1 for an example). ARC is a challenging target for AI.

As F. Chollet writes: “to the best of our knowledge, ARC does not appear to be approachable by any existing machine learning technique (including Deep Learning)” [2]. The main difficulties are the following:

– The expected output is not a label, or even a set of labels, but a colored grid with size up to 30x30, and with up to 10 different colors. It therefore falls in the domain of *structured prediction* [3].
– The predicted output has to match exactly the expected output. If a single cell is wrong, the task is considered as failed. To compensate for that, three attempts are allowed for each input grid.
– In each task, there are generally between 2 and 4 training instances (input grid + output grid), and 1 or 2 test instances for which a prediction has to be made.
– Each task relies on a distinct transformation from the input grid to the output grid. In particular, no evaluation task can be solved by reusing a transformation learned on the training tasks. Actually, each task is a distinct learning problem, and what ARC evaluates is broad generalization and few-shot learning.

A compensation for those difficulties is that the the input and output data are much simpler than real-life data like photos or texts. We think that this helps to focus on the core features of intelligence rather than on scalability issues.

A kaggle competition[1] was organized in Spring 2020 to challenge the robustness of ARC as a measure of intelligence, and to see what could be achieved with state-of-the-art techniques. The winner’s solution manages to solve 20% over 100 (secret) evaluation tasks, which is already a remarkable performance. However, it is based on the brute-force search for chains of grid transformations, chosen from a hand-crafted set of 142 elementary transformations. This can hardly scale to larger sets of tasks, and as the author admits: “Unfortunately, I don’t feel like my solution itself brings us closer to AGI.” Another competitor used a similar domain-specific language based on transformations, and grammatical evolution to search the program space [4]. Their approach solves 3% of the Kaggle tasks, and 7.68% over the 400 training tasks.

Our approach is based on the MDL (Minimum Description Length) principle [9,6] according to which *“the best model for some data is the one that compresses the most the data”*. The MDL principle has already been applied successfully to pattern mining of various data structures (transactions [11], sequences [10], relational databases [7], or graphs [1]), as well as classification [8]. A major difference with those work is that the model to be learned needs not only explain the data but also be able to perform a structured prediction, here the output grid from the input grid. At this stage (Version 2.6), we consider rather simple models that are far from covering a large proportion of ARC tasks: single-color points and shapes, and basic arithmetics and geometry. Our prime objective is to demonstrate the effectiveness of an MDL-based approach to the ARC challenge, and hopefully to AGI. Our approach succeeds on 94/400 (23.5%)

[1] `https://www.kaggle.com/c/abstraction-and-reasoning-challenge`

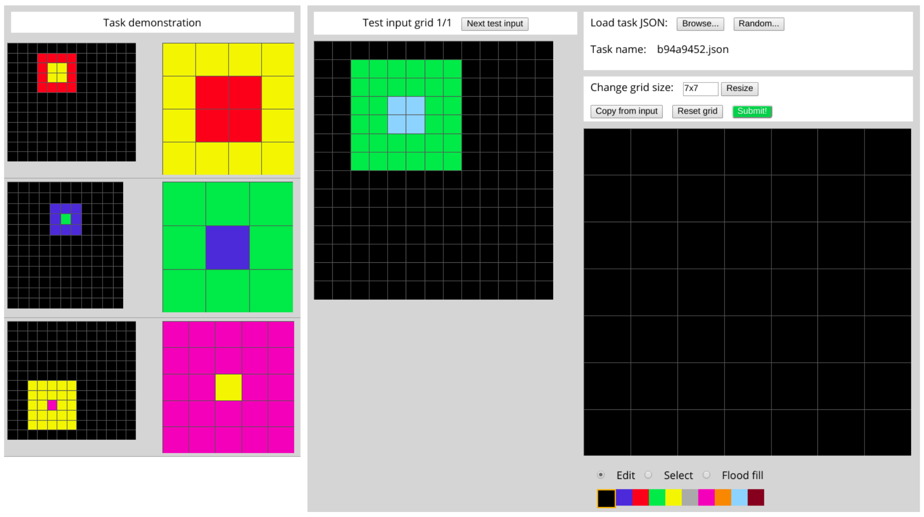

Fig. 1. One of the training tasks (`b94a9452`), with three examples on the left, a test input grid in the middle, and the widget on the right that can be used to draw the missing output grid.

training tasks and 23/400 (5.75%) evaluation tasks, with a learning timeout of 30s only per task. We find such a result quite encouraging given the difficulty of the problem. Moreover, the solved tasks are quite diverse, and the learned models are often close to the simplest models for the task (in the chosen class of models).

This report is organized as follows. Section 2 gives a formal definition of ARC grids and tasks. Section 3 explains our modelling of the ARC tasks in the framework of two-parts MDL, in particular our class of models. Section 4 describes the MDL-based learning process, which boils down to defining description lengths and model refinements. Section 5 reports on the evaluation of our approch on training and evaluation ARC tasks, as well as on the learned models for a few tasks.

2 Defining ARC Tasks

Definition 1 (colors). *We assume a finite set C of distinct symbols, which we call* colors.

In ARC, there are 10 colors coded by digits (0..9) in JSON files, and by colors (e.g., black, purple, red) in the web interface (see Figure 1).

Definition 2 (grid). *A* grid *$g \in C^{h \times w}$ is a matrix of colors with $h > 0$ rows, and $w > 0$ columns. A grid is often displayed as a colored grid. The number of rows $h = height(g)$ is called the* height *of the grid, and the number of columns $w = width(g)$ is called the* width *of the grid.*

A grid cell *is caracterized by* coordinates (i, j)*, where i selects a row, and j selects a column. The color at coordinates (i, j) is denoted either by g_{ij} or by $g[i, j]$. Coordinates range from* $(0, 0)$ *to* $(h - 1, w - 1)$*.*

In ARC, grids have a size from 1×1 to 30×30.

Definition 3 (example). *An* example *is a pair of grids $e = (g^i, g^o)$, where g^i is called the* input grid*, and g^o is called the* output grid*.*

As illustrated by Figure 1, the output grid needs not have the same size as the input grid, it can be smaller or bigger.

Definition 4 (task). *A* task *is a pair $T = (E, F)$, where E is the set of* train examples*, and F is the set of* test examples*.*

The learner has only access to the input and ouput grids of the train examples. Its objective is, for each test example, to predict the output grid given the input grid. As illustrated by Figure 1, the different input grids of a task need not have the same size, nor use the same colors. The same applies to test grids.

The ARC is composed of 1000 tasks in total: 400 for training, 400 for evaluation by developers, and 200 for independent evaluation. Figure 1 shows one of the 400 training tasks, with three train examples, and one test input grid. Developers should only look at the training tasks, not at the evaluation tasks, which should only be used to evaluate the broad generalization capability of the system. Each task has generally between 2 and 4 train examples, and 1 or 2 test examples.

3 Modelling ARC Tasks in Two-Parts MDL

The objective is here to design a class of *task models* that can be used to represent a transformation from input grids to output grids. Our key intuition is that a task model can be decomposed into two grid models, one for the input grids, and another for the output grids. The *input grid model* expresses what all train input grids have in common, while the *output grid model* expresses what all train output grids have in common, as a function of the input grid. The actual transformation from input grid to output grid involves the two models as follows. The input grid model is used to *parse* the input grid, which produces some information that characterizes the input grid. Then, that parsing information is fed into the output grid model in order to generate an output grid.

In the following, paragraphs starting with *Version X* are specific to that version, in contrast to other paragraphs that are generic.

3.1 Models

Version 2. The main component of our grid models is *objects*. Indeed, ARC grids can often be read as objects with various shapes over a background. Objects can

$$
\begin{array}{lcl}
\textit{Pair} & ::= & \mathbf{InOut}(\textit{in} : \textit{Grid}, \textit{out} : \textit{Grid}) \\
\textit{Grid} & ::= & \mathbf{Grid}(\textit{size} : \textit{Vector},\ \textit{color} : C,\ \textit{layers} : \textit{list}(\textit{Object})) \\
\textit{Object} & ::= & \mathbf{PosShape}(\textit{pos} : \textit{Vector},\ \textit{shape} : \textit{Shape}) \\
\textit{Shape} & ::= & \mathbf{Point}(\textit{color} : C) \\
 & | & \mathbf{Rectangle}(\textit{size} : \textit{Vector},\ \textit{color} : C,\ \textit{mask} : \textit{Mask}) \\
\textit{Vector} & ::= & \mathbf{Vec}(i : \mathbb{N},\ j : \mathbb{N}) \\
\textit{Mask} & ::= & \mathbf{Bitmap}(\textit{bitmap} : M) \\
 & | & \mathbf{Full} \ |\ \mathbf{Border} \\
 & | & \mathbf{EvenCheckboard} \ |\ \mathbf{OddCheckboard} \\
 & | & \mathbf{PlusCross} \ |\ \mathbf{TimesCross}
\end{array}
$$

Fig. 2. Datatypes used in grid models and grid data

be disconnected, nested or overlap each other. Each object has a position and a shape.

A shape has a color, and either it is a point, or it fits into a rectangular box of some size. In the latter case, the precise shape is specified by a mask that can be a custom bitmap or one of a few pre-defined regular shapes. The position of objects is relative to the top-left cell of the shape. Each object attribute (e.g., position, size, color) may be constant (e.g., the shape is always red) or variable across the examples.

We define our models as templates over data structures representing concrete pairs of grids. Figure 2 defines those data structure with algebraic data types for gird pairs, grids, objects, shapes, masks, and vectors (2D positions and sizes). They use primitive types for natural numbers ($\mathbb{N}$), colors (C), and 2D bitmaps (M). Uppercase names in bold are called *constructors*, and the lowercase names of constructor arguments are called *fields*.

A task is made of two grids, one for the input grid, and another for the output grid. Each grid is described as a stack of layers on top of a background, each layer being made of a single object. The background has a size (a 2D vector), and a color. An object is a shape at some position. A shape is either a point, described by its color; or a rectangle, described by its size, color, and mask. That mask allows to account for regular and irregular shapes, and indicates which pixels in the rectangle box actually belong to the shape (other pixels can be seen as transparent). A mask can be a custom bitmap or one of a few common regular shapes: full rectangle, rectangle border, checkboards, and centered crosses. A data structure describing the first train pair in Figure 1 is:

$$
\begin{array}{l}
\mathbf{InOut}(\\
\quad \mathbf{Grid}(\mathbf{Vec}(12, 13), \textit{black}, \\
\qquad [\mathbf{PosShape}(\mathbf{Vec}(2, 4), \mathbf{Rectangle}(\mathbf{Vec}(2, 2), \textit{yellow}, \mathbf{Full})), \\
\qquad\ \mathbf{PosShape}(\mathbf{Vec}(1, 3), \mathbf{Rectangle}(\mathbf{Vec}(4, 4), \textit{red}, \mathbf{Full})])), \\
\quad \mathbf{Grid}(\mathbf{Vec}(4, 4), \textit{yellow}, \\
\qquad [\mathbf{PosShape}(\mathbf{Vec}(1, 1), \mathbf{Rectangle}(\mathbf{Vec}(2, 2), \textit{red}, \mathbf{Full})]))).
\end{array}
$$

A *path* in a data structure is a sequence of fields that goes from the root of the data structure to some sub-structure. In the above example, path $in.size.j$ refers to the input grid width, which is 13; path $out.layers[0].shape.color$ refers to the color of the top shape in the output grid. which is *red*; and path $in.layers[1].pos$ refers to the position of the bottom shape in the input grid, which is $\mathbf{Vec}(1,3)$. Given a data structure d and a path p, the substructure of d that is refered by p is written with the dot-notation $d.p$.

In order to generalize from specific pairs of grids to general task models, we introduce two kinds of elements that can be used in place of any sub-structure: unknowns and expressions. An *unknown*, which we note with a question mark ?, indicates that any data structure of the expected type can fit in. For example, a grid model that matches all input grids in Figure 1 is:

$$\begin{array}{l}\mathbf{Grid}(\mathbf{Vec}(12,?), \mathit{black},\\ \quad [\mathbf{PosShape}(?, \mathbf{Rectangle}(?,?,?)),\\ \quad\ \mathbf{PosShape}(?, \mathbf{Rectangle}(?,?,\mathbf{Full}))])\end{array}$$

It matches grids with 12 rows and a black background, and two stacked rectangles of any position, any size, and any color. The top rectangle may have any mask, while the bottom rectangle should be full. The purpose of parsing is to fill in the holes, replacing unknowns with data structures such that the whole resulting data structure correctly describes a grid or a pair of grids. An *unknown path* is a path in the model that leads to an unknown. The unknown paths in the above grid model are: $size$, $layers[0].pos$, $layers[0].shape.size$, $layers[0].shape.color$, $layers[0].shape.mask$, $layers[1].pos$, $layers[1].shape.size$, $layers[1].shape.color$. The set of unknown paths of a model m is denoted by $U(m)$.

A data structure that may contain unknowns is called a *pattern*. A data structure is said *ground* if it contains no unknown. Given a ground data structure d and a pattern p, the notation $d \sim p$ says that d *agrees with* p or that p *matches* d. It means that the ground data structure d can be obtained by a substitution of the pattern unknowns with ground data structures. For example, we have $\mathbf{Vec}(12,14) \sim \mathbf{Vec}(12,?)$.

An *expression* defines part of the data structure as the result of a computation over some input data structure, which we call the *environment*. Expressions are made of primitive values, constructors, operators/functions, and variables. Variables are references to parts of the environment. A convenient way to refer to such parts is to use *paths* in the environment data structure. Expressions that would only be made of primitive values and constructors would actually be data structures, and are therefore not considered as expressions. Figures 3, 4, and 5 define the supported expressions in Version 2.6. Simple arithmetics on integers and vectors, geometric transformations, and Boolean logics on masks account for the majority of operators/functions. There are also constant functions for common values to favor them over arbitrary integers (e.g., $(0,0)$ is a common and special value for a position). The other functions are defined as follows:

- $\mathit{Corner}((i_1,j_1),(i_2,j_2)) = (i_1,j_2)$;
- $\mathit{Span}((i_1,j_1),(i_2,j_2)) = (|i_2-i_1|+1, |j_2-j_1|+1)$;

$$
\begin{array}{lcl}
F_{\mathbb{N}} & ::= & P_{\mathbb{N}} \\
 & | & \mathit{Area}(P_{\mathit{Shape}}) \\
 & | & \mathit{ColorCount}(P_{\mathit{Grid}}) \\
 & | & \mathit{Right/Center/Bottom/Middle}(P_{\mathit{Object}}) \\
 & | & \mathit{Span/Average/Min/Max}(P_{\mathbb{N}}, P_{\mathbb{N}}) \\
F_{\mathit{Vector}} & ::= & P_{\mathit{Vector}} \\
 & | & \mathit{Size}(P_{\mathit{Grid}}) \\
 & | & \mathit{TranslationOnto}(P_{\mathit{Object}}, P_{\mathit{Object}}) \\
 & | & \mathit{TranslationSym}(P_{\mathit{Object}}, P_{\mathit{Object/Grid}}) \\
 & | & \mathit{ProjI/ProjJ}(P_{\mathit{Vector}}) \\
 & | & \mathit{Corner}(P_{\mathit{Vector}}, P_{\mathit{Vector}}) \\
 & | & \mathit{Span/Average/Min/Max}(P_{\mathit{Vector}}, P_{\mathit{Vector}}) \\
F_{\mathit{Color}} & ::= & P_{\mathit{Color}} \\
 & | & \mathit{MajorityColor}(P_{\mathit{Grid}}) \\
F_{\mathit{Mask}} & ::= & P_{\mathit{Mask}} \\
F_{\mathit{Shape}} & ::= & P_{\mathit{Shape}} \\
F_{\mathit{Object}} & ::= & P_{\mathit{Object}} \\
F_{\mathit{Grid}} & ::= & P_{\mathit{Grid}} \\
 & | & \mathit{Strip/PeriodicFactor}(P_{\mathit{Grid}}) \\
 & | & \mathit{Crop}(P_{\mathit{Grid}}, P_{\mathit{Object}})
\end{array}
$$

Fig. 3. Factor-level expressions F_τ for each datatype τ, defined in terms of path variables P.

$$
\begin{array}{lcl}
T_{\mathbb{N}} & ::= & 0..3 \;|\; F_{\mathbb{N}} \\
 & | & \mathit{Incr/Decr}(F_{\mathbb{N}}, 1..3) \\
 & | & \mathit{ScaleUp/ScaleDown}(F_{\mathbb{N}}, F_{\mathbb{N}}) \\
 & | & F_{\mathbb{N}} + F_{\mathbb{N}} \;|\; F_{\mathbb{N}} - F_{\mathbb{N}} \\
T_{\mathit{Vector}} & ::= & (0..3, 0..3) \;|\; F_{\mathit{Vector}} \\
 & | & \mathit{Incr/Decr}(F_{\mathit{Vector}}, 1..3) \\
 & | & \mathit{ScaleUp/ScaleDown}(F_{\mathit{Vector}}, F_{\mathbb{N}}) \\
 & | & F_{\mathit{Vector}} + F_{\mathit{Vector}} \;|\; F_{\mathit{Vector}} - F_{\mathit{Vector}} \\
T_{\mathit{Color}} & ::= & \mathit{black..brown} \;|\; F_{\mathit{Color}} \\
T_{\mathit{Mask}} & ::= & F_{\mathit{Mask}} \\
 & | & \mathit{ApplySym}(\mathit{sym}, F_{\mathit{Mask}}) \\
T_{\mathit{Shape}} & ::= & F_{\mathit{Shape}} \\
 & | & \mathit{ApplySym}(\mathit{sym}, F_{\mathit{Shape}}) \\
 & | & \mathit{Coloring}(F_{\mathit{Shape}}, F_{\mathit{Color}}) \\
T_{\mathit{Object}} & ::= & F_{\mathit{Object}} \\
 & | & \mathit{ApplySym}(\mathit{sym}, F_{\mathit{Object}}) \\
 & | & \mathit{Coloring}(F_{\mathit{Object}}, F_{\mathit{Color}}) \\
T_{\mathit{Grid}} & ::= & F_{\mathit{Grid}} \\
 & | & \mathit{ApplySym}(\mathit{sym}, F_{\mathit{Grid}}
\end{array}
$$

Fig. 4. Term-level expressions T_τ for each datatype τ, defined in terms of factor-level expressions F. Notation $a..b$ stands for a constant value between a and b.

$$
\begin{array}{lcl}
E_{\mathbb{N}} & ::= & T_{\mathbb{N}} \\
E_{\mathit{Vector}} & ::= & T_{\mathit{Vector}} \\
 & | & \mathit{Tiling}(T_{\mathit{Vector}}, 1..3, 1..3) \\
E_{\mathit{Color}} & ::= & T_{\mathit{Color}} \\
E_{\mathit{Mask}} & ::= & T_{\mathit{Mask}} \\
 & | & \mathit{Tiling}(T_{\mathit{Mask}}, 1..3, 1..3) \\
 & | & \mathit{FillResizeAlike}(\mathit{options}, P_{\mathit{Mask}}, T_{\mathit{Vec}}) \\
 & | & \mathit{SelfCompose}(T_{\mathit{Mask}}) \\
 & | & \mathit{UnfoldSym}(\mathit{sym}, T_{\mathit{Mask}}) \\
 & | & \mathit{CloseSym}(\mathit{sym}, T_{\mathit{Mask}}) \\
 & | & \mathit{ScaleTo}(T_{\mathit{Mask}}, T_{\mathit{Vector}}) \\
 & | & \mathit{And}(T_{\mathit{Mask}}, T_{\mathit{Mask}}) \ | \ \mathit{Or}(T_{\mathit{Mask}}, T_{\mathit{Mask}}) \ | \ \mathit{XOr}(T_{\mathit{Mask}}, T_{\mathit{Mask}}) \\
 & | & \mathit{AndNot}(T_{\mathit{Mask}}, T_{\mathit{Mask}}) \ | \ \mathit{Not}(T_{\mathit{Mask}}) \\
E_{\mathit{Shape}} & ::= & T_{\mathit{Shape}} \\
 & | & \mathit{Tiling}(T_{\mathit{Shape}}, 1..3, 1..3) \\
 & | & \mathit{FillResizeAlike}(\mathit{options}, P_{\mathit{Shape}}, T_{\mathit{Vec}}) \\
 & | & \mathit{SelfCompose}(F_{\mathit{Color}}, T_{\mathit{Shape}}) \\
 & | & \mathit{UnfoldSym}(\mathit{sym}, T_{\mathit{Shape}}) \\
 & | & \mathit{CloseSym}(\mathit{sym}, T_{\mathit{Shape}}) \\
 & | & \mathit{ScaleTo}(T_{\mathit{Shape}}, T_{\mathit{Vector}}) \\
E_{\mathit{Object}} & ::= & T_{\mathit{Object}} \\
 & | & \mathit{Tiling}(T_{\mathit{Object}}, 1..3, 1..3) \\
 & | & \mathit{FillResizeAlike}(\mathit{options}, P_{\mathit{Object}}, T_{\mathit{Vec}}) \\
 & | & \mathit{UnfoldSym}(\mathit{sym}, T_{\mathit{Object}}) \\
 & | & \mathit{CloseSym}(\mathit{sym}, T_{\mathit{Object}}) \\
E_{\mathit{Grid}} & ::= & T_{\mathit{Grid}} \\
 & | & \mathit{Tiling}(T_{\mathit{Grid}}, 1..3, 1..3) \\
 & | & \mathit{FillResizeAlike}(\mathit{options}, P_{\mathit{Grid}}, T_{\mathit{Vec}}) \\
 & | & \mathit{SelfCompose}(F_{\mathit{Color}}, T_{\mathit{Grid}}) \\
 & | & \mathit{UnfoldSym}(\mathit{sym}, T_{\mathit{Grid}}) \\
 & | & \mathit{CloseSym}(\mathit{sym}, T_{\mathit{Grid}}, T_{\mathit{Color}}) \\
 & | & \mathit{ScaleTo}(T_{\mathit{Grid}}, T_{\mathit{Vector}}) \\
 & | & \mathit{SwapColors}(T_{\mathit{Grid}}, T_{\mathit{Color}}, T_{\mathit{Color}})
\end{array}
$$

Fig. 5. Expressions E_τ used in grid models for each datatype τ. Expressions are decomposed into terms T, features F, and variables P of various datatypes. Notation $a..b$ stands for a constant value between a and b.

- $\mathit{ProjI}((i,j)) = (i,0)$;
- $\mathit{ProjJ}((i,j)) = (0,j)$;
- $\mathit{Average}((i_1,j_1),(i_2,j_2)) = (\frac{i_1+i_2}{2}, \frac{j_1+j_2}{2})$ (only valid fractions are integers);
- $\mathit{Min}((i_1,j_1),(i_2,j_2)) = (\mathit{min}(i_1,i_2), \mathit{min}(j_1,j_2))$;
- $\mathit{Max}((i_1,j_1),(i_2,j_2)) = (\mathit{max}(i_1,i_2), \mathit{max}(j_1,j_2))$;
- $\mathit{Area}(\mathit{shape})$ is the cardinal of its mask (how many cells belong to the shape);
- $\mathit{Size}(\mathit{grid})$ is the size of a grid, as a vector;
- $\mathit{Strip}(\mathit{grid})$ is a crop of the grid contents, considering the most used color as background;
- $\mathit{ColorCount}(\mathit{grid})$ is the number of non-black colors used in a grid;

- $\mathit{MajorityColor}(\mathit{grid})$ is the most used non-black color in a grid;
- $\mathit{Right}(\mathit{object}) = \mathit{object.pos.i} + \mathit{object.shape.size.i} - 1$;
- $\mathit{Center}(\mathit{object}) = \mathit{object.pos.i} + \mathit{object.shape.size.i}/2$ (only valid when $\mathit{size.i}$ is odd);
- $\mathit{Bottom}(\mathit{object}) = \mathit{object.pos.j} + \mathit{object.shape.size.j} - 1$;
- $\mathit{Middle}(\mathit{object}) = \mathit{object.pos.j} + \mathit{object.shape.size.j}/2$ (only valid when $\mathit{size.j}$ is odd);
- $\mathit{ApplySym}(\mathit{sym}, x)$ is the result of applying some symmetry to x (a mask, shape, object or grid);
- $\mathit{TranslationOnto}(o_1, o_2)$ is the vector that would move object o_1 in contact with object o_2, either straight onto a side or obliquely on a corner (not taking into account the objects' masks, only their bouding box);
- $\mathit{TranslationSym}(o_1, o_2)$ is the vector that would reflect object o_1 with respect to object o_2. If o_2 is a grid, the reflection is with respect to the median axes of the grid.
- $\mathit{PeriodicFactor}(\mathit{grid})$ is the smallest subgrid such that the whole grid is a periodic repeat of it;
- $\mathit{Coloring}(x, \mathit{color})$ is like x (a shape or object) but with a different color;
- $\mathit{Tiling}(x, k, l)$ is the repeat of x (a vector, mask, shape, object or grid) k times horizontally, and l times vertically;
- $\mathit{FillResizeAlike}(\mathit{options}, x, \mathit{size})$ resizes x (a mask, shape, object or grid) to the given size, filling as needed according to the periodic pattern of x;
- $\mathit{SelfCompose}(\mathit{color}, x)$ is the replacement of each cell of x with the given color by x itself (a mask, shape or grid);
- $\mathit{UnfoldSym}(\mathit{sym}, x)$ unfolds x (a mask, shape, object or grid) according to some symmetric pattern;
- $\mathit{CloseSym}(\mathit{sym}, x)$ is the closure of x (a mask, shape, object or grid) so that it matches some symetric pattern;
- $\mathit{ScaleTo}(x, \mathit{size})$ scales x (a mask, shape or grid) to the given size;
- $\mathit{SwapColors}(\mathit{grid}, c_1, c_2)$ is a recoloring of grid, exchanging colors c_1 and c_2.

The environment of the output grid is the input grid, so that parts of the output grid can be computed based on the input grid contents. The input grid has no environment so that expressions are not useful in the input grid model. Re-using the above model for input grids, we can now define a correct and complete model for the task in Figure 1, using unknowns for the input grid, and expressions for the output grid:

$$
\begin{array}{l}
\mathbf{InOut}(\\
\quad \mathbf{Grid}(\mathbf{Vec}(12, ?), \mathit{black}, \\
\qquad [\ \mathbf{PosShape}(\mathbf{Vec}(?, ?), \mathbf{Rectangle}(\mathbf{Vec}(?, ?), ?, \mathbf{Full})), \\
\qquad\ \ \mathbf{PosShape}(\mathbf{Vec}(?, ?), \mathbf{Rectangle}(\mathbf{Vec}(?, ?), ?, \mathbf{Full})\ ])), \\
\quad \mathbf{Grid}(\mathit{layers}[1].\mathit{shape.size}, \mathit{layers}[0].\mathit{shape.color}, \\
\qquad [\ \mathbf{PosShape}(\\
\qquad\qquad \mathit{layers}[0].\mathit{pos} - \mathit{layers}[1].\mathit{pos}, \\
\qquad\qquad \mathbf{Rectangle}(\mathit{layers}[0].\mathit{size},\ \mathit{layers}[1].\mathit{color},\ \mathbf{Full}))\ ])).
\end{array}
$$

In words, this models says: *"find two stacked full rectangles on a black background in the input grid, then generate an output grid, whose size is the same as the bottom shape, and whose background color is the color of the top shape, and finally add a full rectangle whose size is the same as the top shape, whose color is the same as the bottom shape, and whose position is the difference between the top shape position and the bottom shape position (relative position)."*

A model is *well-formed* if it is well-typed (constructor arguments, operator operand types, etc.), and if all variables are valid paths in the environment data. More specifically, a task model is well-formed if the input grid model contains no variable, and if all variables in the output grid model are valid paths in the input grid model.

A task model is *definite* if its output grid model contains no unknown, so that it can be used to deterministically generate an output grid from a data structure describing the input grid.

3.2 Data According to a Model

In MDL-based approaches, it is essential to have a lossless encoding of data in order to have a fair comparison of different models. Therefore, it is important to identify the "data according to the model", i.e. the data that needs to be added to a model in order to capture all the information in the original data.

We first consider grid models, where data is a grid. An analogy can be made with formal languages and grammars. In this analogy, grids are sentences, grid models are grammars, and data according to the model is the parse tree. A parse tree explains how the grammar generates the sentence. Here, data structures as defined in Figure 2 play the role of parse trees for grid models, they are *grid parse trees* and we denote them with the greek letter π. A grid parse tree, the data according to the model, is a data structure that instantiates the grid model by replacing expressions by their value, and unknowns by values such that the resulting data structure correctly describes the grid. The description length will only count the data structures that instantiate the unknowns because the rest of the parse tree is already known from the model.

For input grid g_k^i (k-th example), π_k^i denotes one possible grid parse tree according to some fixed input grid model m^i. Similarly, for output grid g_k^o, π_k^o denotes one possible grid parse tree according to some fixed output grid model m^o and output environment ε^o.

In practice, a grid model may not explain all the cells of a grid. This happens in case of noise in the data, and also in the learning process where intermediate models are obviously incomplete. To account for those situations without losing information, we define "data according to the model" as a grid parse tree plus the list of cells that differ between the original grid g and the grid generated by the grid parse tree π. We call this additional information a *grid delta*, written δ.

An additional difficulty is when a test input grid does not match the input grid model learned from the train examples. For instance, all input grids in the train examples have size (10,10) but the test input grid has size (10,12). In this case, there is no grid parse tree that agrees with both the grid model and the grid. In many cases, this discrepancy does not alter the validity of the model, which only happens to be overspecific (a form of overfitting). Our approach to this difficulty is to allow for approximate parsing, i.e. to allow grid parse trees to diverge from the grid model. As those differences are present in the grid parse tree π, there is no loss of information. Of course, the description length will take into acount such differences to penalize them.

As an illustration, let us consider the following (incomplete) input grid model for the task in Figure 1:

$$\mathbf{Grid}(?, \mathit{black}, [\ \mathbf{PosShape}(?, \mathbf{Rectangle}(?, ?, \mathbf{Full}))\ ]).$$

There are mainly two "data according to the model" for the first input grid g_1^i (the upper index is i for input grids and o for output grids, the lower index identifies the example):

1. matching the outer red rectangle
   $\pi_1^i = \mathbf{Grid}(\mathbf{Vec}(12, 13), \mathit{black},$
   $[\ \mathbf{PosShape}(\mathbf{Vec}(1, 3), \mathbf{Rectangle}(\mathbf{Vec}(4, 4), \mathit{red}, \mathbf{Full}))\ ])$
   $\delta_1^i = \{(2, 4, \mathit{yellow}), (2, 5, \mathit{yellow}), (3, 4, \mathit{yellow}), (3, 5, \mathit{yellow})\}$
2. matching the inner yellow rectangle
   $\pi_1^i = \mathbf{Grid}(\mathbf{Vec}(12, 13), \mathit{black},$
   $[\ \mathbf{PosShape}(\mathbf{Vec}(2, 4), \mathbf{Rectangle}(\mathbf{Vec}(2, 2), \mathit{yellow}, \mathbf{Full}))\ ])$
   $\delta_1^i = \{(1, 3, \mathit{red}), (1, 4, \mathit{red}), (1, 5, \mathit{red}), (1, 6, \mathit{red}), (2, 3, \mathit{red}), (2, 6, \mathit{red}), (3, 3, \mathit{red}), ...\}$

The first solution looks better because it has a much smaller grid delta, and its grid parse tree has the same size as in the second solution.

3.3 Primitive Functions on Grids, Grid Models, Grid Parse Trees, and Grid Deltas

We here define the key primitive functions to manipulate grids, grid models, grid parse trees and grid deltas.

We first define *grid deltas* and simple arithmetics on them.

Definition 5 (grid delta). *Let $g_1, g_2 \in C^{h \times w}$ two same-size grids. The* grid delta *$\delta \subseteq \mathbb{N} \times \mathbb{N} \times C$ between the two grids, written $g_2 - g_1$, is defined as the set of cells that need changing color in g_1 in order to obtain g_2.*

$$\delta = g_2 - g_1 = \{(i, j, c) \mid g_2[i, j] \neq g_1[i, j], c = g_2[i, j]\}$$

A grid delta δ can be added to a grid g_1 to get a corrected same-size grid g_2.

$$g_2 = g_1 + \delta \iff \forall i \in [0, h[, j \in [0, w[\colon g_2[i, j] = \begin{cases} c & \text{if } (i, j, c) \in \delta \\ g_1[i, j] & \text{otherwise} \end{cases}$$

Then we define two functions that are directly related to the semantics of grid models. The first function converts a grid parse tree into a grid.

Definition 6 (grid drawing). *Let π be a grid parse tree, $draw(\pi)$ is the grid that results from the drawing following the instructions given by the grid parse tree.*

Version 2. Given a grid parse tree π as specified in Figure 2, the grid $draw(\pi)$ is obtained by first generating a grid with the specified size and background color, and then layers are drawn on that grid, bottom-up. Each layer, either a point or a rectangle, is drawn in the obvious way. For a point $\textbf{PosShape}(\textbf{Vec}(i,j), \textbf{Point}(c))$, the cell at position (i,j) is set to color c. For a rectangle $\textbf{PosShape}(\textbf{Vec}(i,j), \textbf{Rectangle}(\textbf{Vec}(h,w), c, m))$, each cell at position $(i+x, j+y)$, for any $x \in [0,h[$ and $y \in [0,w[$, is set to color c if the relative position (x,y) belongs to mask m.

The second function applies a grid model to an environment, resolving references to it, and replacing expressions by their evaluation result.

Definition 7 (model application). *Let m be a grid model, and ε an environment for the model. $apply(m,\varepsilon)$ is the model that results from the substitution of expressions in m by their evaluation over the environment ε. The resulting model has no expression but may still have unknowns. The operation is ill-defined if some variable in m is not defined in ε.*

Version 2. Environments are grid parse trees, and variables are paths into them.

We also define two non-deterministic functions that produce grid parse trees from a grid model, and hence grid descriptions. The first function corresponds to the parsing of a grid.

Definition 8 (grid parsing). *Let m be a grid model, and g be a grid. $parse(m,g)$ returns a ranked set of grid parse trees π at rank ρ that approximately agree with model m, and approximately draw as the grid g. The set allows for different interpretations of the grid by the model, and the ranking reflect the fact that some interpretations seem better than others.*

The description lengths that are used for guiding the learning process (see Section 4.1) are here used to rank the different parse trees.

Version 2. This is by far the most complex primitive function. Parsing is decomposed in three stages. First, the grid is partitioned into contiguous monocolor regions. Second, objects are looked for as single parts or as unions of parts to cover the case where some objects are overlaped by other objects. Small parts (less than 5 covered cells) are also considered as adjacent individual points. When an object's shape is not a point or a full rectangle, two shapes are considered: one shape with a full mask, considering the missing cells as noise on top of the rectangle; and another shape with a mask containing the cells in the rectangle box that belong to the object and have the same color. A regular mask is used when possible (e.g., borders, crosses), otherwise a bitmap is used. Third, matching superpositions of objects are looked for according to the grid model. For each layer, candidate objects are considered in a fixed ordering that includes some simple heuristics. Black objects are considered last because black is often the background color. Then, objects are considered in decreasing number of covered cells. Beyond those heuristics, a total ordering is defined because it makes experiments more reproducible, and also because it helps disambiguate some situations. For instance, if the model contains several points, they will be matched to points in the grid from left to right, top-down. Layers are parsed top-down. When done in a naive recursive way, the search space is not explored in fair way. Indeed, the second candidate object for the first layer is only considered when all candidates have been considered for the other layers in combination with the first candidate object for the first layer. Defining the rank of a parse tree for all layers as the sum of the rank of the chosen candidate object of each layer, we explore the search space in increasing rank to favor the first candidate objects across all layers. This improvement was introduced in Version 2.2. In Version 2.3, we set a maximal rank (e.g., 16) to avoid a systematic exploration when there are no valid parse tree.

As an example of parsing, the test input grid in Figure 1 has three contiguous parts: the light blue one, the green one, and the black one. Three rectangles can be identified from those parts: a small light blue square, a green rectangle overlaped by the blue one, and a black rectangle covering the whole grid and overlaped by the other rectangles. According to the above model, this input grid is parsed as a 2x2 light blue rectangle at coordinates (3,4), on top of a 6x6 green rectangle at coordinates (1,2), on top of a black background.

Compared to Version 1, parsing has been made non-deterministic in the sense that a sequence of grid parse trees is produced. We use the description lengths as defined below to order this sequence from the shorter DL to the longer DL. Indeed, the shorter the DL, the more plausible the grid parse tree is. Because of combinatorial problems in the parsing process, we use various bounds: maximum rank when parsing layers, maximum number of produced grid parse trees before sorting them, and maximum number of grid parse trees actually returned. Relative to approximate parsing, we also bound the number of differences w.r.t. the model, and produce the parse trees in increasing number of differences.

The second function corresponds to generation, where the grid parse tree is produced by filling in the holes of the grid model, i.e. guessing at the missing elements.

Definition 9 (grid generation). *Let m be a grid model without any expression. generate(m) returns a set of grid parse trees that are obtained by replacing unknowns in the model by type-compatible sub-trees.*

Version 2. We simply replace unknowns by default values. It makes it very unlikely to generate the right output grid by chance but it helps to visualize the intermediate states of the learning process. The default position is $(0, 0)$, the default size is $(10, 10)$ for grids, and $(2, 2)$ for rectangles, the default color is black for grids and grey for shapes, the default mask is the full mask, the default shape is a rectangle with default values in each field.

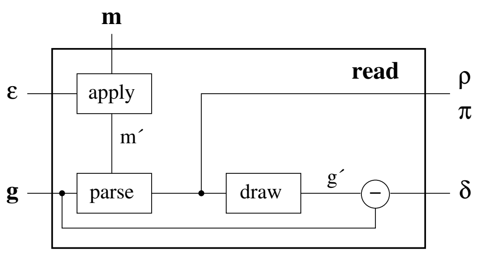

Fig. 6. The *read* component.

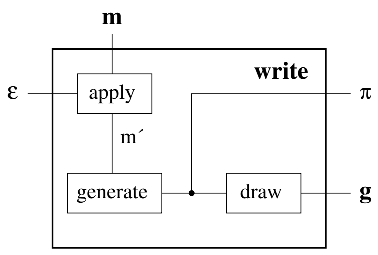

Fig. 7. The *write* component.

3.4 Reading and Writing a Grid with a Grid Model

We define two important components for reading and writing grids. They are simply composed from the above primitve functions, and are used as basic blocks for training and using models.

Definition 10 (grid reading). *Let m be a model. Function $read_m$ takes an environment and a grid as input, and returns a ranked set of grid parse tree and delta as output.*

$$read_m(\varepsilon, g) = \{(\rho, \pi, \delta) \mid m_\varepsilon = apply(m, \varepsilon), (\rho, \pi) \in parse(m_\varepsilon, g), \delta = g - draw(\pi)\}$$

Figure 6 gives a schematic description of the *read* component. When reading a grid g, the grid model is first applied to the input environment, in order to resolve any expression that the model may contain. The resulting model is then used to parse the input grid into a grid parse tree π. Finally, the grid delta between the grid specified by π and g is computed. The pair (π, δ) provides a lossless representation of the input grid, as expressed by the following lemma.

$$(\pi, \delta) \in read_m(\varepsilon, g) \implies g = draw(\pi) + \delta.$$

Definition 11 (grid writing). *Let m be a grid model. Function $write_m(\varepsilon)$ takes an environment as input, and returns a parse tree and a grid as output.*

$$write_m(\varepsilon) = \{(\pi, g) \mid m_\varepsilon = apply(m, \varepsilon), \pi = generate(m_\varepsilon), g = draw(\pi)\}$$

Figure 7 gives a schematic description of the *write* component. When writing a grid, the grid model is first applied to the input environment, in order to resolve any expression that the model may contain. The resulting model is then used to generate a grid parse tree, by instantiating the unknowns, from which a concrete grid can be drawn. The parse tree is output in addition to the grid in order to provide an explanation of how the grid was generated.

3.5 Task Models: Prediction, Training, and Creation

Using a task model $M = \textbf{InOut}(m^i, m^o)$ to predict the output grid from the input grid simply consists into the chaining of grid reading and grid writing.

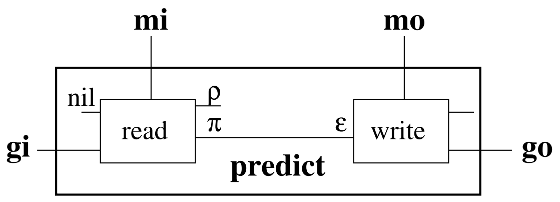

Fig. 8. Prediction mode: producing an output grid from the input grid

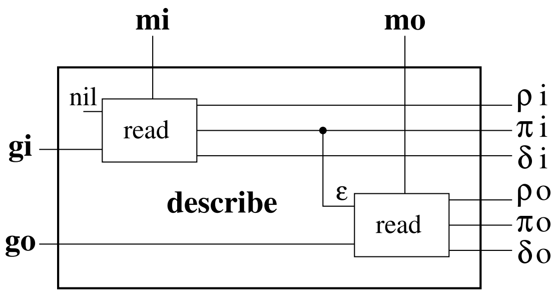

Fig. 9. Training mode: reading the input and output grids in chain

$$predict_M(g^i) = \{g^o \mid (\pi^i, \delta^i) \in read_{m^i}(nil, g^i), g^o \in write_{m^o}(\pi^i)\}$$

Figure 8 gives a schematic description of prediction. The input grid g^i is first read with the input grid model m^i, using an empty environment, which non-deterministically results in an input grid parse tree π^i and an input delta δ^i. Then, the predicted output grid is produced by writing the output grid model m^o using the input grid parse tree as environment.

For instance, given the input grid parse tree returned by the reading of the test input grid in Figure 1, the output grid model generates a 2x2 green rectangle at coordinates (2,2), on top of a 6x6 light blue background, which is the expected grid.

During the training phase, both the input and output grids are known. However, computing a grid delta between the predicted output grid and the expected output grid is not enough to learn a better model. It is more useuful to have the grid parse tree for each grid, because it provides an explanation of how the model "sees" the grids. We therefore define the *describe* function that chains the reading of the input and ouput grids, and outputs a joint decription of the pair of grids.

$$\begin{aligned} describe_M(g^i, g^o) = \{(\rho^i, \pi^i, \delta^i, \rho^o, \pi^o, \delta^o) \mid & (\rho^i, \pi^i, \delta^i) \in read_{m^i}(nil, g^i), \\ & (\rho^o, \pi^o, \delta^o) \in read_{m^o}(\pi^i, g^o)\} \end{aligned}$$

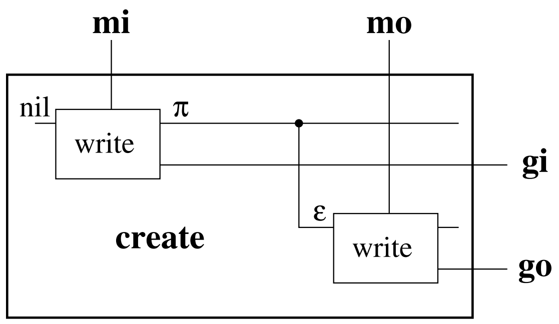

Fig. 10. Creation mode: writing input and output grids in chain

Figure 9 gives a schematic description of training. The first step is the same as prediction. The input g^i is first read with the input grid model m^i, using an empty environment, which non-deterministically resuts in an input grid parse tree π^i and an input grid delta δ^i at some rank ρ^i. In the second step, the output grid g^o is read with the output grid model m^o, using π^i as environment, which non-deterministically results in an output grid parse tree π^o and an output grid delta δ^o at rank ρ^o. The two grid parse trees and the two grid deltas collectively form a lossless description of the two grids according to the task model.

It is also possible to *create* new examples for the task, i.e. to create both the input and output grids from nothing else than the input and output grid models.

$$create_M() = \{(g^i, g^o) \mid (\pi^i, g^i) \in write_{m^i}(nil), (\pi^o, g^o) \in write_{m^o}(\pi^i)\}$$

Figure 10 gives a schematic description of the creation mode. The input grid model is used to write the input grid given an empty environment. The parse tree of the input grid is then used as an environment for the output grid model to generate an output grid that is consistent with the input grid. The process is highly non-deterministic as in general many grids can be generated for a given input grid model. Indeed, it generally contains several unknowns.

4 Learning Task Models

MDL-based learning works by selecting the model that compresses the data the best. This involves two main components: (a) a measure of compression so as to compare models, and (b) a strategy to explore the model space. The measure of compression is called *description length.*

4.1 Description Lengths

In two-part MDL, the description length (DL) $L(M, D)$ measures the number of bits required to communicate data D with model M. It is decomposed in two parts according to the equation

$$L(M,D) = L(M) + L(D|M)$$

where $L(M)$ is the DL of the model, and $L(D|M)$ is the DL of the data encoded with the model. In our case, a model M is a task model $\mathbf{InOut}(M^i, M^o)$, and data D is the part of a task $T = (E, F)$ that is accessible to the learner. As the objective is to learn a model that generalizes to new instances, only the train examples are accessible, so $D = E$.

The description length of the task model is the sum of the description lengths of its two grid models.

$$L(M) = L(m^i) + L(m^o)$$

We detail the description length of grid models below. The description length of the task data $D = E$ is the sum of the description length of each example.

$$L(D|M) = L(E|M) = \alpha \sum_{(g^i,g^o)\in E} L(g^i, g^o|M)$$

Factor α equals 10 by default to give more weights to the data, relative to the model, and hence allow the learning of more complex models. Indeed, ARC tasks have a very low number of examples (3 on average).

The DL of an example is based on the most compressive joint description of the pair of grids. The joint description is based on the chained reading of the two of grids according to the task model.

$$\begin{aligned} L(g^i, g^o|M) = \; & min_{\rho^i,\pi^i,\delta^i,\rho^o,\pi^o,\delta^o \in describe(M,g^i,g^o)} \\ & [\ L(\rho^i, \pi^i, \delta^i|m^i, nil) + L(\rho^o, \pi^o, \delta^o|m^o, \pi^i)\ ] \end{aligned}$$

The DL of a grid g according to a grid model m and an environment ε is computed by using the rank ρ, the parsed description π and delta δ as an intermediate grid representation.

$$L(\rho, \pi, \delta|m, \varepsilon) = [L_{\mathbb{N}}(\rho) - L_{\mathbb{N}}(0)] + L(\pi|apply(M, \varepsilon)) + L(\delta|\pi)$$

The first term measures the amount of information need to choose among the different descriptions. It is the universal encoding ($L_{\mathbb{N}}$, see below) of the rank relative to the preferred description (rank=0). The second term measures the amount of information that must be added to the model and environment to get the description, typically unknown bindings and divergences found with approximate parsing. The third term measures the differences between the original grid and the grid that is generated by the parsed description. A correct model is found when $L(\rho^o, \pi^o, \delta^o|m^o, \pi^i) = 0$ for all examples, i.e. when output grids can be perfectly predicted from the input grids.

Note that the description lengths of the grid parse tree π is relative to the grid model m applied to its environment. Indeed, we only need to encode the parts of the grid parse tree that are unknown in the grid model or that differ from the grid model. Its detailed definition depends on the definition of grid models. The description length of a grid delta δ is relative to a grid parse tree π. This is valid as the grid parse tree is encoded before the delta, and is therefore available when encoding the grid delta. Its detailed definition also slightly depends on grid models.

We also introduce normalized definitions of description lengths $\hat{L}$. Indeed, input and output grids may have very different sizes (e.g., the output grid has a single cell), and this can hinder the learning of a grid model for the smaller grid. Our normalization gives the same weight to the input grids and to the output grids, taking the initial model M_0 as a point of reference. We therefore define λ^i and λ^o as respectively the input and output contributions to $L(M_0, D) = \lambda^i + \lambda^o$:

$$\lambda^i = L(m_0^i) + \alpha \sum_{(g^i, g^o) \in E} L(g^i | M_0)$$
$$\lambda^o = L(m_0^o) + \alpha \sum_{(g^i, g^o) \in E} L(g^o | M_0),$$

where $L(g^i|M_0)$ is the input component of $L(g^i, g^o|M)$, and similarly for the output. From there, we define normalized description lengths as follows.

$$\hat{L}(M, D) = \hat{L}(M) + \hat{L}(D|M)$$
$$\hat{L}(M) = \frac{1}{\lambda^i} L(m^i) + \frac{1}{\lambda^o} L(m^o)$$
$$\hat{L}(D|M) = \alpha \sum_{(g^i, g^o) \in E} \frac{1}{\lambda^i} L(g^i|M) + \frac{1}{\lambda^o} L(g^o|M)$$

As the learning strategy only selects models giving shorter and shorter description lengths, the normalized description $\hat{L}(M, D)$ is in the interval $[0, 2]$, and the contribution of each part (input and output) is in the interval $[0, 1]$.

When defining description lengths for grid models and grid parse trees, we assume a few basic description lengths for elementary types:

- $L_{\mathbb{N}}(n) = 2\ \log(n + 1) + 1$: universal encoding of natural intergers (including zero), more precisely the gamma encoding shifted by one to include zero;
- $L_P(x \in X) = -\log P(x)$: encoding based on a probability distribution P over elements X;
- $L_X(x \in X) = \log |X|$: encoding based on a uniform distribution over elements X.

Version 2. In order to define the description lengths of grid models and grid parse trees, we first need to define the description lengths of primitive types, algebraic datatypes, expressions, and templates.

Primitive types. Integers are used for positions and sizes. For positions, we use a uniform distribution over the rows or columns of a grid. Hence for a row position i in a grid g, $L(i|g) = \log \mathit{height}(g)$; similarly, for a column position j in a grid g, $L(j|g) = \log \mathit{width}(g)$. When the grid size is not known, the maximum grid size is used (30 in the ARC challenge). For sizes, we use the universal encoding of natural numbers $L_{\mathbb{N}}$. For the color of shapes, we use the uniform distribution L_C. For the background color of grids, we use a probability distribution P_{BC} that gives a higher probability to color black ($P_{BC} = 0.91$) than to other colors ($P_{BC} = 0.01$). Indeed, in most tasks, grids have a black background. Hence, for a background color c, $L(c) = -\log P_{BC}(c)$. The description length of a rectangular bitmap is defined according to a prequential code, which favors unbalanced bitmaps (either nearly full or nearly empty).

Algebraic types. For each algebraic datatype (*Grid*, *Object*, *Shape*, *Vector*, *Mask*), each constructor determines which fields are to be encoded. Therefore, it suffices to encode the choice of a constructor among the constructors of the datatype, in addition to the encoding of the fields. For datatypes *Grid*, *Object*, and *Vector*, there is a single constructor, hence there is no choice to be made. Hence, $L(\mathbf{Grid}) = L(\mathbf{Object}) = L(\mathbf{Vec}) = 0$. For datatype *Shape*, there are two constructors, to which we give a uniform distribution. Hence, $L(\mathbf{Point}) = L(\mathbf{Rectangle}) = -\log 0.5 = 1$. For datatype *Shape*, there are 7 constructors, to which we give the following distribution: $P_{\mathit{Shape}} = \{\mathbf{Bitmap} : 0.3, \mathbf{Full} : 0.5, \mathbf{Border} : 0.1, \mathbf{EvenCheckboard} : 0.025, \mathbf{OddCheckboard} : 0.025, \mathbf{PlusCross} : 0.025, \mathbf{TimesCross} : 0.025\}$. For a given constructor, the encoding of some field may depend on other fields, thus inducing an ordering of fields. This is the case for constructor **Rectangle** where the mask field depends on the size field (see the encoding of masks above).[2] Grids have a list of shapes. The encoding of a list starts with the universal encoding of the length of the list, followed by the encoding of the list elements.

Expressions. There are two kinds of expressions (with distribution P_e): function (or operator) applications ($P_e = 0.3$), and variables ($P_e = 0.7$). Functions are encoded along a uniform distribution over the functions whose result type is the expected type. The expected type is determined by the syntactic context of the expression: e.g., a color if it is the color field of the grid or a shape, an integer if it is a component of the position of a shape, or an operand of an arithmetic operator. Variables are encoded according to their context of insertion in the model, and hence according to its datatype. The encoded information tells how to transform the context path into the variable path. Therefore, the most similar they are, the shorter the encoding is. We first compute the longest common prefix of the two paths, then we encode by how much the context path must be shortened, and finally we encode what has to be added to the common prefix to reconstruct the variable path, taking into account the suffix of the context path.

Templates. There are three kinds of templates (with distribution P_t): primitive values and constructors ($P_t = 0.2$), expressions ($P_t = 0.7$), and unknowns ($P_t = 0.1$). Each field of a constructor and each argument of a function is a

[2] We could also consider that the position field depends on the size field. Indeed, given a grid size and a rectangle size, not all positions are possible if we assume that the rectangle must fit into the grid.

template, thus allowing in principle arbitrary mixing of values, constructors, functions, and unknowns. In Version 2, we never use unknowns for grids, objects, shapes, and function arguments. Expressions are given higher probability because they explain some part of the grid parse tree from the environment. unknowns are given the lower probability because they correspond to unexplained parts.

Grid models ($L(m)$) are templates over type *Grid*, and are encoded as such.

Grid parse trees ($L(\pi|m)$) are only made of primitive values and constructors, and could be encoded as such. However, as grid parse trees are obtained by parsing grids according to a grid model, parts of the grid parse tree are already encoded in the grid model. Actually, the very purpose of grid models is precisely to factor out what all example grids have in common, and avoiding to encode it repeatedly. The only parts of grid parse trees that need to be encoded are (a) the parts corresponding to unknowns in the grid model, and (b) the parts that differ from the grid model due to approximate parsing.

(a) About the unknowns, a fixed ordering can be derived from the model. Therefore, it is enough to concatenate the encodings of the grid parse subtrees that correspond to each unknown.
(b) About the differing parts, we have no a priori information, neither on their number, nor on their location in the grid parse tree. We first encode their number with universal encoding. We then encode each differing part by encoding its location in the grid parse tree, using a uniform distribution, and the subtree at that location.

The encoding of the location of the differing parts could probably be improved in several ways. First, some differences are more likely than others: e.g., a different rectangle width vs a completely different shape. Second, not all subsets of locations make sense as a difference location should not be inside another difference location as it would entail to encode the same part twice.

Grid delta ($L(\delta|\pi)$). For comparability with grid models and grid parse trees, we encode a grid delta δ as a set of points. The description length of a grid delta is hence defined as:

$$L(\delta|\pi) = L_{\mathbb{N}}(|\delta|) + \sum_{(i,j,c)\in\delta} L(\mathbf{PosShape}(\mathbf{Vec}(i,j), \mathbf{Point}(c))|draw(\pi)).$$

Note that the description length of points is made relative to the grid drawn from the grid parse tree. In particular, this makes the grid size available to the encoding of the point position.

For the task in Figure 1, Table 1 compares the description lengths of the initial model and the solution model given as example above, for $\alpha = 10$ (each example counts as 10). The tables detail the split between input grid and output grid on one hand, and the split between the model ($L(M)$) and the data according to the model ($L(D|M)$) on the other hand. We observe that the increase in the model DL is largely compensated by the decrease of the data DL. The resulting

Table 1. Comparison of description lengths of the initial model (top) and the solution model given above (bottom).

	$L(M)$	$L(D\|M)$	$L(M,D)$	$\hat{L}(M,D)$
input	15.6	13695.3	13710.9	1.000
output	15.6	2124.4	2140.0	1.000
chained	31.2	15819.7	15850.9	2.000

	$L(M)$	$L(D\|M)$	$L(M,D)$	$\hat{L}(M,D)$
input	81.2	2429.8	2511.1	0.183
output	79.5	0.0	79.5	0.037
chained	160.7	2429.8	2590.5	0.220

compression gain is 16.3%, a six-fold reduction. In particular, with the solution model, the description length of the ouput grids has fallen to zero, which means that output grids are entirely explained by the model and the input grid parse tree.

4.2 Learning Strategy

The model space is generally too vast for a systematic exploration, and heuristics have to be employed to guide the search for a good model. The learning strategy consists in the iterative refinement of models, starting with an initial model, in search for the best model, i.e. the most compressive model. Given a class of models, a search strategy is therefore entirely specified by:

1. an initial model M_0 (*initial model*), generally the simplest one;
2. and a function R mapping each model M to an ordered list $M_1, M_2, \ldots$ of refined models (*refinements*), generally taking into account the data according to the model (here, grids, grid parse trees and grid deltas).

We say that a refinement M_i is *compressive* if it reduces the description length compared to the original model M. The ordering of refinements is important because the number of possible refinements could be huge, and it has a cost to evaluate each refinement by measuring its description length. Indeed, this involves the parsing of both input and output grids by each refined model. This ordering is therefore a key part of the heuristics.

Another part of the heuristics consists in bounding the number of compressive refinements per model that are considered, and the number of models that are kept after each iteration. A *greedy search* selects the first compressive refinement, and has therefore a single model at each iteration. A *beam search* provides a wider exploration by selecting K_r compressive refinements per model, and then selecting among them the K_m best models for the next iteration. K_m is the beam width. The search halts when no compressive refinement can be found.

Version 2. The initial model is made of two minimal grid models, reduced to a background with unknown size and color, and no shape layer.

$$\mathbf{InOut}(\mathbf{Grid}(?, ?, [\,]), \mathbf{Grid}(?, ?, [\,]))$$

Two kinds of refinements are used: (1) the addition of a new object to a grid model (input or output), and (2) the replacement of a part of a template (typically some unknown) by another template (typically a value or an expression). In the first kind of refinement, the new object can be inserted at any position in the list of layers. The inserted object can be a fully unspecified point or rectangle: $\mathbf{PosShape}(?, \mathbf{Point}(?))$ or $\mathbf{PosShape}(?, \mathbf{Rectangle}(?, ?, ?))$. In the output grid model, the inserted object can also be an object or a shape from the input grid model, using paths as variables to reference them: p_{object} or $\mathbf{PosShape}(?, p_{shape})$. Indeed, it is common for the output grid to reuse shapes from the input grid, either at the same position or at a different position to be determined. Inserting such references to the input grid is not necessary because they could be learned from unspecified objects but they accelerate learning.

The second kind of refinements enables to replace part of a model at path p by a template t. The replacing template is either a *pattern*, i.e. a combination of primitive values, constructors, and unknowns; or an *expression* using environment variables. For the input grid model, it can only be patterns because the environment is empty. For the output grid model, every path into the input grid parse tree is available as an environment variable. In this version, we only consider patterns made of a single primitive value or a single constructor with all fields being unknowns. By following a principle of invariance in ARC tasks respective to scale and color, we do not allow for the insertion of specific integer and color values in the input model, and we only insert small integer values in 1..3 in the output model. We consider all expressions in Figure 5, starting from the environment variables. When the replaced part p is an unknown, it enables to make the model more specific to the task. When the replaced part is a pattern in the output grid model, and it is replaced by an expression, it enables to better define the output grid as a function of the input grid. To summarize, unknowns may be replaced by patterns and expressions, and patterns in the output grid model may be replaced by expressions.

Let E be the set of examples (pairs of grids), and $M = (m^i, m^o)$ be the current model. We generate the following refinements:

1. An input unknown at $p \in U(m^i)$ can be replaced by pattern t if

$$\forall (g^i_k, g^o_k) \in E : \exists (\pi^i_k, \delta^i_k, \pi^o_k, \delta^o_k) \in \mathit{train}_M(g^i_k, g^o_k) : \pi^i_k.p \sim t$$

2. An output unknown at $p \in U(m^o)$ can be replaced by pattern t if

$$\forall (g^i_k, g^o_k) \in E : \exists (\pi^i_k, \delta^i_k, \pi^o_k, \delta^o_k) \in \mathit{train}_M(g^i_k, g^o_k) : \pi^o_k.p \sim t$$

3. An output grid model part at $p \in m^o$ can be replaced by expression e if

$$\forall (g^i_k, g^o_k) \in E : \exists (\pi^i_k, \delta^i_k, \pi^o_k, \delta^o_k) \in \mathit{train}_M(g^i_k, g^o_k) : \pi^o_k.p = \mathit{apply}(e, \pi^i_k)$$

A tentative ordering of refinements is the following: input shapes, output shapes, input expressions, output expressions.

The sequence of compressive refinements that leads to the above model about task in Figure 1 takes 0.6s to compute, and is shown in the table below. The normalized description length is the one reached after applying the refinement. Each refinement is described as an equality between a model path (left-hand side) and a template (right-hand side). Refinements defining a layer by a shape actually insert a new layer for that shape.

step	$\hat{L}(M, D)$	refinement
0	2.000	
1	1.250	$in.layer[1] = \mathbf{PosShape}(?, \mathbf{Rectangle}(?, ?, ?))$
2	0.960	$out.layer[0] = \mathbf{PosShape}(?, \mathbf{Rectangle}(?, ?, ?))$
3	0.757	$out.size = layer[1].shape.size$
4	0.678	$out.layer[0].shape.color = in.layer[1].shape.color$
5	0.626	$in.layer[0] = \mathbf{PosShape}(?, \mathbf{Rectangle}(?, ?, ?))$
6	0.505	$out.color = in.layer[0].shape.color$
7	0.390	$out.layer[0].shape.pos = in.layer[0].shape.pos - in.layer[1].shape.pos$
8	0.282	$out.layer[0].shape.size = in.layer[0].shape.size$
9	0.235	$out.layer[0].shape.mask = in.layer[1].shape.mask$
10	0.228	$in.layer[0].shape.mask = \mathbf{Full}$
11	0.220	$in.layer[1].shape.mask = \mathbf{Full}$

At steps 1 and 5, two rectangle objects are introduced in the input. At step 2, a rectangle object is introduced in the output. Combined with the background grid, this is enough to cover all cells in both grids. As of step 5, there are therefore empty grid deltas. At steps 3 and 6, the size and color of the output grid are respectively equated to the bottom input shape ($in.layer[1].shape.size$) and to the top input shape ($in.layer[0].shape.color$). At steps 4 and 8, the size and color of the output shape are equated to respectively to the top input shape ($in.layer[0].shape.size$), and to the bottom input shape ($in.layer[1].shape.color$). At step 7, the position of the output shape is defined as the difference between the two input objects. At step 9, the mask of the ouput shape is equated to the mask of the bottom input shape (always **Full** in the examples). At this stage, the model has solved the task as it can correctly generate the output grid for any input grid. Steps 10-11 finds that all input shapes are *full* rectangles.

5 Evaluation

Table 2 reports on the performance of our approach on both training and evaluation tasks (400 tasks each), for successive versions and timeouts. The timeout sets a limit to the learning time for each task. On both collections of tasks, we provide performance for train examples, for which output grids are known, and for test examples. In each case, we give two measures n_1/n_2: n_1 is the number

Table 2. Performance on training and evaluation tasks as the number of tasks solved, per version ($\alpha = 10$)

			training			evaluation		
version	date	timeout	time	train	test	time	train	test
v1.0	2020-10-21	20s	7s	8 / 9.5	5 / 5.5	12s	0 / 0.3	0 / 0.0
v1.1	2021-03-01	2s	1.5s	16 / 17.3	5 / 5.0	1.5s	4 / 4.3	4 / 4.0
v2.0	2021-05-01	30s	20s	28 / 34.6	19 / 19.5	24s	6 / 7.8	5 / 5.0
v2.0	2021-05-01	60s	35s	33 / 41.2	22 / 22.5	46s	7 / 9.9	6 / 6.0
v2.0	2021-05-01	120s	57s	35 / 43.5	24 / 24.5	82s	8 / 10.9	7 / 7.0
v2.1	2021-09-12	30s	18s	32 / 40.0	25 / 25.5	23s	7 / 10.9	7 / 7.0
v2.1	2021-09-12	60s	32s	36 / 44.9	27 / 27.5	43s	8 / 12.5	7 / 7.0
v2.1	2021-09-12	120s	52s	38 / 46.9	28 / 28.5	77s	10 / 14.0	8 / 8.5
v2.2	2021-10-22	30s	14s	41 / 50.1	29 / 29.5	20s	8 / 12.7	6 / 6.5
v2.3	2022-01-29	30s	14s	44 / 56.2	32 / 33.0	19s	11 / 16.3	7 / 7.5
v2.4	2022-02-12	30s	13s	49 / 59.6	42 / 42.5	19s	13 / 21.5	6 / 6.5
v2.5	2022-03-08	30s	15s	59 / 71.6	52 / 52.5	20s	15 / 20.9	9 / 9.5
v2.6	2022-11-07	30s	13s	97 / 107.8	94 / 94.5	20s	34 / 38.3	23 / 23.5
v2.6	2022-11-07	60s	21s	97 / 107.8	94 / 94.5	35s	35 / 40.1	24 / 24.5

of tasks for which *all* examples are correctly solved, and n_2 is the sum of partial scores over tasks. For instance, if a task has two test examples and only one is solved, the score is 0.5.

Version 1. This first version was designed by looking at tasks `ba97ae07` and `b94a9452`. Evaluating it on training tasks has shown that this simple model manages to solve 3.5 additional tasks: `6f8cd79b`, `25ff71a9` (2 correct examples), `5582e5ca`, `bda2d7a6` (1 correct example out of 2). This version even manages to explain all train examples of 3 additional tasks, hence 8 tasks in total, although they fail on the related test examples: tasks `a79310a0` (3 examples), `bb43febb` (2 examples), `694f12f3` (2 examples), and `a61f2674` (2 examples). For task `a79310a0`, v1 recognizes that a shape is moved, and changes color but, whereas all train shapes are rectangles, the test shape is not a rectangle. For tasks `bb43febb` and `694f12f3`, v1 finds unplausible expressions to replace some unknowns, e.g. replacing some height by a position instead of a height decremented by a constant, or replacing some unknown by a constant that happens to be valid for train examples but not for the test example. For task `a61f2674`, v1 is not powerful enough but given that there are only two train examples, it manages to find some ad-hoc regularity.

Despite those encouraging result on training tasks, no evaluation task could be solved with this version. We wonder whether this is a statistical fluke or the fact that evaluation tasks are harder than the training tasks.

Version 1.1. Table 3 compare results with Version 1.1 for different strategies concerning model refinements. For example, the first line consider expressions with variables before constants, and consider refinements in this order: input

Table 3. Performance on training tasks as the number of tasks solved, for different strategies, in Version 1.1 with timeout 2s, and $\alpha = 10$. C = constants first, V = variables first, Ei = input expressions, Si = input shapes, Eo = output expressions, So = output shapes.

strategy	train	test
V/Si-Ei-So-Eo	11 / 11.7	4 / 4.0
V/Ei-Si-So-Eo	14 / 15.0	5 / 5.0
V/Ei-Si-Eo-So	16 / 17.3	5 / 5.0
C/Ei-Si-Eo-So	16 / 17.3	5 / 5.0

shapes, input expressions, output shapes, output expressions. The best strategy among those tested is Ei-Si-Eo-So. Refining the input model first provides useful parameters when refining the output model. Inserting expressions before inserting new shapes helps to disambiguate the parsing process, which often happens when a model contains several unspecific shapes.

The successful training tasks are `ba97ae07`, `bda2d7a6`, `6f8cd79b`, `e48d4e1a`, `25ff71a9`. Compared to Version 1, we improve on `bda2d7a6` by succeeding on the the two test instances, and gain `e48d4e1a`. The former is explained by the use of masks, and the latter is explained by the improved extraction of rectangles from grids. However, we loose on `b94a9452` and `5582e5ca`. In Version 1, the latter was successful by chance, and the former because test inputs were considered for training. Finally, 17.3 training instances are successful compared to 9.6 in the previous version. Hence, all in all, we consider this new version as a valuable improvement.

This is confirmed by Table 2 that shows that this new version succeeds in 4 evaluation tasks, instead of none for the previous version.

Version 2. Table 2 reports results for Version 2 for different timeouts. It uses a greedy search ($K_m = 1$) but at each step, it look for up to 20 compressive refinements ($K_r = 20$), selecting the best one among them. Grid parsing looks for up to 512 grid parse trees, and keep only the three best ones. Approximate parsing is not activated on train examples, and limited to 3 differences on test examples. Table 2 shows a sharp improvement w.r.t. previous versions, especially on the training tasks. The number of successful tasks jumps from 5 to 24 in the training dataset, and from 4 to 8 in the evaluation dataset. This seems to confirm that evaluation tasks are intrisically more complex than training tasks. As the language of models is the same as in Version 1.1, the improvement come mostly from the way those models are used and modified:

– the use of paths in grid parse trees rather than variable names,
– the introduction of non-determinism in grid parsing,
– the ordering of candidate grid parse trees and refinements,
– the ability to replace any part of a template, not only unknowns,
– better definitions of description lengths.

Higher timeouts are required because of the non-determinism that significantly increases computation complexity.

For a timeout of 60s, the 22 successful training tasks are: `1bfc4729`, `1cf80156`, `1f85a75f`, `25ff71a9`, `445eab21`, `48d8fb45`, `5521c0d9`, `5582e5ca`, `681b3aeb`, `6f8cd79b`, `a1570a43`, `a79310a0`, `a87f7484`, `aabf363d`, `b1948b0a`, `b94a9452`, `ba97ae07`, `bda2d7a6`, `bdad9b1f`, `e48d4e1a`, `e9afcf9a`, `ea32f347`. For a timeout of 120s, the additional successful tasks are: `23581191`, `91714a58`. Task `48d8fb45` is successful although the learned model does not exhibit a proper understanding of it. The system found a correlation that was probably not intended by the creator of the task. The models learned on other tasks are adequate, and are rather diverse despite the simplicity of the model language. We informally describe a few learned models to illustrate this diversity.

– `a79310a0`. Find a cyan rectangle and move it one row down. Unexpectedly, the test instance has an irregular shape instead of a rectangle. Thanks to approximate parsing, the input grid parse tree contains a mask of the irregular shape, which is used to generate the output grid.
– `aabf363d`. Find an arbitrary shape and a colored point in the bottom left corner, and output the same shape except for the color, which is the same as the point.
– `b94a9452`. Find an arbitray shape on a pink background, simply change the background color to red.
– `ba97ae07`. Find two full rectangles, reverse the list of layers: bottom rectangle on top, top rectangle at bottom.
– `bdad9b1f`. Find two segments, a 2x1 cyan rectangle, and 1x2 red rectangle. Extend each segment into a line spanning the whole grid, then add a yellow point at the crossing of the two lines.
– `e48d4e1a`. Find two colored lines, a 1x10 rectangle and a 10x1 rectangle, on a 10x10 black grid. Find also a grey rectangle in the top right corner with width=1. Generate the same lines at different positions, according to the height H of the grey rectangle. The vertical line is moved H columns left (substraction), and the horizontal line is moved H rows down (addition). This involves arithmetic expressions for computing the new positions.
– `ea32f347`. Find three grey full rectangles, implictly ordered from the larger (top layer) to the smaller (bottom layer) according to the parsing ordering heuristics. Generate the same rectangles with different colors: the top one gets blue, the middle one gets yellow, and the bottom one gets red.
– `1cf80156`. Crop an arbitrary colored shape on a black grid.
– `1f85a75f`. Crop an arbitrary colored shape on a black grid that also contains many points of other colors at random positions (a kind of noise).
– `6f8cd79b`. Starting with a black grid of any size, generate a cyan grid of same size, and add a black rectangle at position (1,1) whose size is two rows and two columns less than the grid. The effect is to add a cyan border to the input grid.
– `445eab21`. Find two shapes, the larger one implicitly on top. Generate a 2x2 grid whose color is the same as the top shape.

– `681b3aeb`. Find two shapes on a black background. Generate a 3x3 grid with the color of one shape, and add at position (0,0) the other shape. Actually, the two shapes complement each other and pave the 3x3 grid. The learned model works because the evaluation protocol allows for three predicted output grids, and there are only two possibilities as the top left cell must belong to one of the two shapes. Moreover, as the larger shape is tried first and it has a higher probability to occupy the top left cell, there is more than 50% chance being correct on first trial.
– `5521c0d9`. Find three full rectangles: a yellow one, a red one, and a blue one. Move each of them upward in the grid by as much as their height.
– `5582e5ca`. Find two shapes on a 3x3 colored grid. Generate the same colored grid without the shapes. Actually, the task is to identify the majority color in the input grid. Even without a notion of counting and majority in the models, the MDL principle implicitly identifies the majority color as the one that compresses the most.

For 11 training tasks, the approach was successful on all train instances although it failed on the test instances. This means that a working model was found but that it does not generalize to the test instances. Here are some causes for the lack of generalization:

– wrong choice between constants and variables, and finding accidental arithmetic equalities (e.g., `0962bcdd`);
– wrong segmentation of shapes, e.g. prefering a full rectangle to an irregular shape (e.g., `1caeab9d`);
– the transformation should be performed several times in the test instance whereas once is enough in all train instances (e.g., `29c11459`);
– there is an invariance by grid rotation or grid transposition (e.g., `4522001f`).

Versions 2.1, 2.2. This version brings only a few changes to the model but it significantly improves the efficiency of parsing grids, which is the major cost of learning a model. Table 2 shows that we achieve similar results in 30s as we did in 120s. The limit of the number of parse trees per grid was lowered from 512 down to 64 thanks to a better parsing strategy for stacks of layers.

For the record, the main changes relative to Version 2 are the following:

– distinction between shapes and objects to better represent object moves,
– richer set of masks (borders, checkboards, and crosses),
– zero as a nullary operator to avoid expressions like $e_1 - e_1$,
– small parts can be seen as adjacent points,
– better strategy when parsing a stack of layers,
– normalized description length to balance inputs and outputs w.r.t. grid size,
– insertion of input shapes and objects into the output model,
– optimizations to reduce timeout.

The 7 newly solved training tasks w.r.t. Version 2 at 60s are: `08ed6ac7`, `23581191`, `72ca375d`*, `7e0986d6`, `a61ba2ce`, `be94b721`, `ddf7fa4f`*. In the two tasks with a star, the learned model is not precise enough but because the system is allowed three attempts and there are only a few reading alternatives, it finds the correct answer (without really understanding why). For example, in task `72ca375d`, the input grid has three objects, and the output grid has only one of them, the one that has a symmetry. Our system only understands that the output selects an input object, and makes an attempt for each object.

Version 2.3. The main change in this version is the introduction of various operators/functions for integers, vectors, and masks. Unfortunately, this only brings 3 more successful tasks. There are 5 new successful tasks: `b230c067`, `d631b094` (uses the area of a shape), `d9fac9be`*, `de1cd16c`*, `ea786f4a`. There are two lost tasks: `bda2d7a6`, `ddf7fa4f`*. The number of considered refinements had to be raised from 20 to 100 to account for the many new expressions. Various optimizations were applied to maintain a similar runtime on average.

Version 2.4. Here are the main changes in this version:

– a more generic parsing of the background color (chosen before parsing the layers, excluding candidate shapes of that color);
– taking into account hidden cells when generating candidate mask models, i.e. partially visible shapes;
– more fine-grained generation of candidate expressions by considering path roles, not only path kinds: e.g., adding a position vector and a size vector gives a position vector;
– more expressions: `Incr/Decr` on ints and vectors, `Min/Max/Average` on ints, `Span` on ints and vectors, translation vectors between objects;
– new parameter for limiting the number of considered expressions, to prevent cases of combinatorial explosion and to control runtime (set to 10,000);

Those changes result in 10 additional successful training tasks, 42 (10.5%) in total. Unfortunately, the number of successful evaluation tasks does not progress, even loosing one task (with the allocated timeout).

Version 2.5. Here are the main changes in this version:

– when parsing layers, discarding the remaining parts that contain the currently selected part (optimization);
– sorting candidate expressions (and values) by increasing DL gain estimate rather than ad-hoc measures. This enables to reduce the number of refinements to be evaluated (now set to 20 instead of 50);
– an additional role for ints and vectors: "move" in addition to "position" and "size". For example, a difference between two positions is now a move rather than a size. Moves can have negative values unlike positions and sizes;
– added a number of new expressions:
  • vector computations: `ConstVec` for small constant vectors, `ProjI` and `ProjJ` for projecting a vector on either axis;
  • `Coloring` to change the color of an object or shape;

  - scaling and resizing objects, shapes, masks and vectors: `ScaleUp` and `ScaleDown` with some constant factor, `ScaleTo` to rescale to some vector as size, `ResizeAlikeTo` to extend any pattern up to some vector as size, `Tiling` to repeat some pattern a constant number of times on each axis;
  - expressions based on the 8 grid symmetries (`flipHeight`, `flipWidth`, `flipDiag1`, `flipDiag2`, `rotate180`, `rotate90`, `rotate270`): `ApplySym` applies a symmetry to an object, shape or mask, `TranslationSym` measures the translation of an object relative to some pivot object and some symmetry, and `UnfoldSym` that unfolds an object, shape or mask according to a symmetry pattern.

Those changes result in 10 additional successful training tasks, 52 (13%) in total.

Version 2.6. Here are the main changes in this version:

- many new functions:
  - mask/grid functions: *Size*, *Strip*, *Crop*, *PeriodicFactor*, *ApplySym*, *TranslationSym*, *Tiling*, *FillResizeAlike*, *SelfCompose*, *UnfoldSym*, *CloseSym*, *ScaleTo*;
  - coloring functions: *ColorCount*, *MajorityColor*, *Coloring*, *SwapColors*.
- the description length of parsed description includes a penalty for the parse rank;
- a pruning stage at the end of the learning process to remove unnecessary elements in the built model, in order to make it more general;
- various optimizations, in particular the memoization of grid and mask transformations.

Those changes result in a sharp increase of the number of solved tasks, from 52 to 94 training tasks (23.5%).

6 Discussion and Perspectives

We have demonstrated that combining descriptive grid models and the MDL principle is a viable and encouraging approach to the ARC challenge. In contrast to other approaches, it focuses on the contents of grids rather than on transformations from input grids to output grids. Actually, our learning algorithm does not even try to predict the output grid from the input grid, only to find a short joint description of the pair of grids, only assuming that the output grid may depend on the input grid. Prediction of the output grid only comes as a by-product. The learned model can also be used to create new examples. We think it makes our approach more robust and scalable, and more cognitively plausible.

The progress through the different versions has more to do with improvements in the general approach than with the class of models. Indeed, the model has not changed much from the first version (stacks of points and boxed shapes over a colored background, plus very simple arithmetics). The key to improvements is *flexibility*. Flexibility lies mostly in the parsing procedure through non-determinism and approximation. To tackle the complexity coming from those, the MDL principle appears as essentiel to select the most plausible parsings, i.e. those that have the shortest description length. The MDL principle is therefore used at two levels: at a low level for parsing grids, and at a high level for choosing model refinements.

We were surprised to see that results on the evaluation tasks are much lower than on the training tasks. The same observation has been made previously [4]. We refrained to look at the evaluation tasks, as advised by the ARC designer, to avoid the inclusion of ARC-specific priors in our approach. We suspect that the evaluation tasks have larger or more complex grids because the runtime per task is higher. We also suspect that they require more difficult generalizations as this is a key factor of the measure of intelligence proposed by F. Chollet.

We are confident that our approach can solve many more tasks because the current grid models are quite simple. First, only a few of the core human priors are covered. Second, a number of tasks involve relatively simple whole-grid transformations (e.g. rotations, symmetries). Those are typically the tasks that are solved by transformation-based approaches but are not yet solvable by our model. Another important limit is when a grid contains a variable-sized collection of objects, each of which must be treated in the same way. This requires some form of loop in the model.

References

1. Bariatti, F., Cellier, P., Ferré, S.: Graphmdl: Graph pattern selection based on minimum description length. In: Berthold, M.R., Feelders, A., Krempl, G. (eds.) Advances in Intelligent Data Analysis - 18th International Symposium on Intelligent Data Analysis, IDA. LNCS, vol. 12080, pp. 54–66. Springer (2020), `https://doi.org/10.1007/978-3-030-44584-3_5`
2. Chollet, F.: On the measure of intelligence. arXiv preprint arXiv:1911.01547 (2019)
3. Dietterich, T.G., Domingos, P., Getoor, L., Muggleton, S., Tadepalli, P.: Structured machine learning: the next ten years. Machine Learning 73(1), 3–23 (2008)
4. Fischer, R., Jakobs, M., Mücke, S., Morik, K.: Solving Abstract Reasoning Tasks with Grammatical Evolution. In: LWDA. pp. 6–10. CEUR-WS 2738 (2020)
5. Goertzel, B.: Artificial general intelligence: concept, state of the art, and future prospects. Journal of Artificial General Intelligence 5(1), 1 (2014)
6. Grünwald, P., Roos, T.: Minimum description length revisited. arXiv preprint arXiv:1908.08484 (2019)
7. Koopman, A., Siebes, A.: Characteristic Relational Patterns. In: ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining. pp. 437–446. KDD '09, ACM (2009)
8. Proen ca, H.M., van Leeuwen, M.: Interpretable multiclass classification by MDL-based rule lists. Information Sciences 512, 1372–1393 (Feb 2020), `https://www.sciencedirect.com/science/article/pii/S0020025519310138`
9. Rissanen, J.: Modeling by shortest data description. Automatica 14(5), 465–471 (1978)

10. Tatti, N., Vreeken, J.: The long and the short of it: Summarising event sequences with serial episodes. In: Int. Conf. on Knowledge Discovery and Data Mining (KDD). pp. 462–470. ACM (2012)
11. Vreeken, J., Van Leeuwen, M., Siebes, A.: Krimp: mining itemsets that compress. Data Mining and Knowledge Discovery 23(1), 169–214 (2011)