2023 IEEE 23rd International Symposium on Computational Intelligence and Informatics (CINTI) | 979-8-3503-4294-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/CINTI59972.2023.10382009

Solving ARC with non-procedural program induction

Norbert Neumann*, Ádám Pintér†

* John von Neumann Faculty of Informatics, Óbuda University, Budapest, Hungary
† John von Neumann Faculty of Informatics, Óbuda University, Budapest, Hungary, pinter.adam@nik.uni-obuda.hu

***Abstract*—This paper attempts to solve the Abstraction and Reasoning Corpus (ARC) [1] which was made to measure strong generalization in artificial intelligence systems. The existing program induction solutions have the disadvantage that the program tree defined by the used Domain-Specific Language (DSL), in which the search takes place, grows exponentially with the length of the program, making the search space too large to find the appropriate program. Our program induction-based solution attempted to eliminate the need for searching in the DSL's program tree. This requires that the induced program should not be procedural, i.e., it should not consist of a sequence of instructions built on top of each other. The induction of such programs is much simpler, as breaking down the instruction's interdependence causes the search time to grow only linearly with the program's length. This method successfully reduced the size of the problem's search space. Compared to previous methods, the average task-solving time was significantly lower than in any other publications. It was able to solve 41 tasks in the training set and 10 tasks in the public test set, which is the fourth-best result among previous publications.**

***Index Terms*—abstraction, reasoning, program induction**

I. Introduction

ARC contains 1000 image-based reasoning tasks. In a task, a certain number of (usually three) demonstration examples are given. An example consists of an input and its corresponding output, where the output is obtained from a complex transformation of the input. Both the input and output consist of a square grid ranging from 1×1 to 30×30, where each position contains a symbol that can be visualized as a color as an example is shown in Figure 1. There are a total of ten possible colors. The algorithm solving the task needs to recognize the transformation performed on the demonstration examples and then apply it to one (in some cases more than one) test example. The algorithm only receives feedback on the success of its answer, not on the specific errors made.

II. Related Work

All previous solutions attempted to solve the problem using program induction with the help of a domain-specific language specifically designed for this purpose. The number of programs that can be generated from the given DSL exponentially increases with the length of the program, so many existing solutions have tried to narrow down the search space using some kind of heuristics.

Fischer et al. [2] used an evolutionary algorithm to search for the appropriate solution within the program space of the DSL. Out of the 400 training samples, the approach achieved a correct solution rate of 7.68% (±0.61%), while for the 100 private test samples, the algorithm provided correct solutions in 3% of the cases.

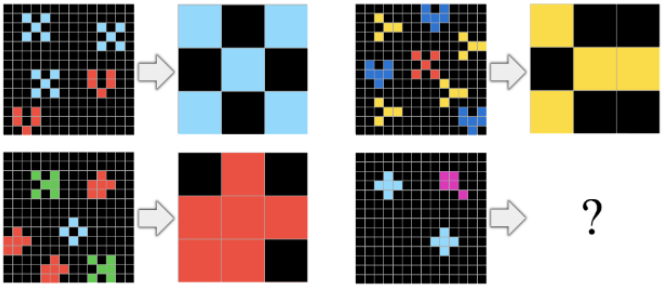

Fig. 1. Structure of an ARC task [1]. The output is the most frequently occurring object in the input.

Banburski et al. [3] attempted to solve 36 pre-selected tasks using the Dreamcoder architecture. The architecture was able to solve 22 tasks within 4 iterations.

Acquaviva et al [4] also employed the Dreamcoder architecture, which created a dataset containing natural language descriptions of the programs that can be found in ARC. In addition to the input-output pairs, this description was also provided as input to a neural network assisting in the traversal of the program tree. With the natural language descriptions, the architecture successfully solved 22 tasks in the public test set, and without them, it solved only 18 tasks.

The most successful solution [5] so far achieved a 20% success rate on the test tasks. The brute-force approach attempted to generate correct solutions using a combination of handcrafted image transformations consisting of approximately 7000 lines of code. According to the author, the significantly larger and faster image transformations compared to other approaches contributed to the success of his solution.

A similar solution to the previous is the Visual Imagery Reasoning Language (VIMRL) [6] system, which also exhaustively explores the program space covered by the DSL using a brute-force approach. Compared to the previous solution, this system also incorporates *"high-level"* operations that use algorithms instead of elementary images or mathematical operations for solving ARC tasks. In its best run, the system solved 104 tasks in the training set and 26 tasks in the public

test set.

Xu et al. [7] followed the assumption that ARC tasks can be more easily solved by focusing on relations between objects extracted from individual images, rather than the images themselves. After transforming the tasks into graphs, this solution searches for the program solving the task within a DSL tailored to this data structure, like previous approaches. The method successfully solved 57 tasks in the training set.

All previous approaches have been unidirectional, meaning that the system tries to generate the expected output from the input. This is inefficient because it evaluates programs for which there is no evidence based on the output. This is what Alford et al. [8] aims to improve, where the authors' goal is for their system to solve ARC tasks in a way similar to human thinking [9]–[12]. According to the authors, this is achieved by jointly examining the input and output images to derive the correct transformation in multiple successive steps. The objective is to establish a *"binding"* between the objects in the output image and the property of the input image, meaning that there exists a sequence of operations that derives the currently examined object in the output from the properties of the input. This binding process is bidirectional, so the DSL designed for this purpose includes invertible operations. They formalized this process as a reinforcement learning problem: at each step, the agent's task is to decide which instruction from the DSL to apply, with what parameters (intermediate results), and in which direction (input-output or output-input). The algorithm was tested on 18 pre-selected tasks that involved symmetries, and it successfully solved 14 of them.

III. Methodology

A. Domain-Specific Language

Like many existing solutions, We try to interpret ARC tasks in an object-centric manner. According to this approach, the individual images consist of a background and one or more independent objects. The preprocessing module will perform the interpretation of images in this way. Therefore, the induction algorithm should not work with the pixels of the given image but with the extracted objects from the image. For this purpose, the DSL should include a description for representing objects. An object is represented by the DSL with the following seven properties:

1) $Shape$: The color-independent shape of the object. In practice, this is a two-dimensional matrix where values greater than zero indicate regions of the shape. In order to visually represent the shape, some coloring needs to be assigned to these regions. This task is handled by the $C : region \rightarrow color$ mapping. With this mapping, the shape remains color-independent, and different objects with different colorings can be visually generated from a single shape.
2) X: The vertical coordinate of the object.
3) Y: The horizontal coordinate of the object.
4) $Width$: The width of the object's shape.
5) $Height$: The height of the object's shape.
6) C: The color mapping of the object.
7) N: Noise objects for current objects.

Some ARC tasks require the existence of shapes that follow an infinitely repeating pattern. Due to their infinite extent, it is necessary to provide the *width* and *height* of the object.

In ideal cases, the first six properties can describe individual objects. However, in some ARC tasks, partial occlusion may occur, meaning that one object partially obscures another. For many tasks, it is necessary to know which objects partially occlude other objects. Set N provides this information. It is not necessary to consider cases where one object completely occludes another. If an object does not occlude anything, then N is the empty set.

The operations of the DSL produce one of the seven description properties. The structure of an operation is as follows:

$$operation\_name \quad [arg0] \quad [arg1] \tag{1}$$

where the second argument is only used in a few operations. In this paper, the following operations are used:

- x_of $[obj]$: returns the X coordinate of the given object.
- y_of $[obj]$: returns the Y coordinate of the given object.
- $width\_of$ $[obj]$: returns the width of the given object.
- $height\_of$ $[obj]$: returns the height of the given object.
- $shape\_of$ $[obj]$: returns the shape of the given object.
- $color\_of$ $[obj]$ $[n]$: returns the color of the n-th region of the given object.
- $color\_map\_of$ $[obj]$: returns the entire color map of the given object.
- $noise\_of$ $[obj]$: returns all noise objects of the given object.
- nth_noise_of $[obj]$ $[n]$: returns the n-th noise object of the given object.
- $const\_shape$ $[shape]$: an identity operation that returns the given shape.
- $const\_number$ $[num]$: an identity operation that returns the given integer.
- $const\_color$ $[c]$: an identity operation that returns the given color.
- $flatten$ $[shape]$: *"flattens"* the given shape data type, resulting in a shape that has the same size and shape as the operation parameter, but assigns the same region to each position.
- $dominant\_color$ $[obj]$: returns the color that appears most frequently in the color map of the given object.
- $find\_object$ $[predicate]$: returns the object for which the given predicate evaluates to true (there can only be one such object, and if the predicate is true for multiple objects, the operation will throw an error).

During induction, it may happen that we cannot express a property of an object based on the input image using any operation. In such cases, we still need a method to access these values. For this purpose, We introduced the constant operations $const\_shape$, $const\_number$, and $const\_color$,

which are one-parameter functions with a return value equal to the parameter received.

The predicates are functions that return a Boolean value and have one or two parameters. They have the same structure as the operations. A predicate indicates whether an object has a certain property. It plays a role during the execution of the transformation program. The complete set of predicates is the followings:

- $color\ group$: two or more objects form a color group c if the color map of all such objects contains the color c.
- $visual\ group$: two objects visually match, if the shape, width, and length of the two objects match, visual groups can be created according to this relation, in a similar way to color groups.
- $belongs\_to\_color\_group\ [obj]$: true if the object received as a parameter belongs to at least one color group, false otherwise.
- $unique\_color\ [obj]$: true if the object received as a parameter is single-colored and does not belong to any color group, false otherwise.
- $belongs\_to\_specific\_color\_group\ [obj]\ [c]$: true if the object received as a parameter belongs to the color group c (so the color of the group is c), false otherwise.
- $belongs\_to\_visual\_group\ [obj]$: true if the object received as a parameter belongs to a visual group, false otherwise.
- $unique\_visual\ [obj]$: true if the object received as a parameter does not belong to a visual group, false otherwise.
- $belongs\_to\_specific\_visual\_group\ [obj]\ [V]$: true if the object received as a parameter belongs to visual group V, false otherwise.
- $const\_visual\ [obj][V]$: true if the object received as a parameter visually matches V, false otherwise.
- $has\_color\ [obj][c]$: true if the object colormap received as a parameter contains color c, false otherwise.
- $container\ [obj]$: true if the object received as a parameter contains another object, false otherwise.
- $contained\ [obj]$: true if the object received as a parameter is contained by another object, false otherwise.
- $parent\ [obj]$: true if the object received as a parameter has noise objects, false otherwise.
- $child\ [obj]$: true if the object received as a parameter is a noise object of another, false otherwise.
- $unicolor\ [obj]$: true if the object received as a parameter is monochrome, false otherwise.
- $multicolor\ [obj]$: true if the object received as a parameter is multi-colored, false otherwise.
- $is\_max\ [obj]$: true if the size of the object received as a parameter is the largest among all objects filtered from the image, false otherwise.
- $is\_min\ [obj]$: true if the size of the object received as a parameter is the smallest among all objects filtered from the image, false otherwise.
- $is\_max\_color\ [obj]\ [c]$: true if the object received as a parameter contains the color c and this color is found in this object most of the time.
- $contained\_by\_const\_visual\ [obj]\ [V]$: true if the object received as a parameter is contained by another object that visually matches V, false otherwise.
- $belongs\_to\_visual\_group\_contained\_by\_const\_visual\ [obj]\ [V]$: true if the given object belongs to a visual group and the group has at least one element contained by an object that visually matches V, false otherwise.

The transformation program, which receives the extracted objects from the input image as parameters, can be built from the previously described parts of the DSL. It generates the expected output for the given task. The transformation program consists of one or more sub-transformations. A sub-transformation is composed of a condition and a program, where the condition is a set of predicates, and the program is an expression that contains seven operations, each generating a descriptive property of the object.

The execution of the transformation program follows the following steps: Iterate over the set of input objects received as parameters. Evaluate the condition part of each sub-transformation for each object. If all predicates are true for a particular object, execute the program part of that sub-transformation. The result is an ARC object, which is added to the set of output objects. Repeat this process for each input object, resulting in a set of output objects. Later, this set is transformed into an image by the post-processing module. By comparing the generated output with the expected output, it can be determined whether the system successfully solved the given ARC task.

The DSL contains 4 data types: shape, color, number, and object, which are built from the first three types. The expressions that produce values for these types are:

$$\begin{aligned} ES(shape) = & \\ & \{expr \in DSL_{SHAPE} | eval(expr) = shape\} \\ ES(color) = & \\ & \{expr \in DSL_{COLOR} | eval(expr) = color\} \\ ES(number) = & \\ & \{expr \in DSL_{NUMBER} | eval(expr) = number\} \\ ES(object) = & \\ & \{expr \in DSL_{OBJECT} | eval(expr) = object\} \end{aligned} \tag{2}$$

where DSL_{SHAPE}, DSL_{COLOR}, etc. are sets representing parts of the DSL operation set that evaluate the shape, color, etc. data types. The operations for these four data types are as follows:

$$\begin{aligned} DSL_{SHAPE} = & \\ & \{group\_of, const\_group, flatten\} \\ DSL_{COLOR} = & \\ & \{color\_of, const\_color, color\_map\_of, \\ & dominant\_color\} \\ DSL_{NUMBER} = & \\ & \{x\_of, y\_of, width\_of, height\_of, \\ & const\_number\} \\ DSL_{OBJECT} = & \\ & \{noise\_of, nth\_noise\_of, find\_object\} \end{aligned} \tag{3}$$

B. Induction Algorithm

In the first step, for each $x \in X$ input object, two sets of predicates are generated.

First is $P(x)$ which contains the predicates that, when evaluated on the given x object, return true:

$$P(x) = \{p \in U_{predicate} | p(x) = 1\} \tag{4}$$

Second, is $P_{unique}(x)$ which contains the predicates that only evaluate to true for the given x object among the X objects:

$$P_{unique}(x) = \{p \in P(x) | \nexists x_i \in X - x \text{ where } p(x_i) = 1\} \tag{5}$$

In the next step, equivalence saturation is performed on each description property of each $y \in Y$ output object. This results in a set for each property, which contains all the DSL expressions that would evaluate the value of that property. Since the object representation consists of 7 such properties, the result is 7 sets for each output object:

$$ES(y) = (ES^S(y), ES^X(y), ..., (ES^N(y)) \tag{6}$$

The expressions generated by ES are built from DSL operations and object references. In the next step, We replace the object references with the unique predicates of the respective object. For example, if we have an expression $(rotatex180)$ and $P_{unique}(x) = \{P1, P2, P3\}$, the result of the substitution is:

$$\{(rotate\ P_1\ 180), (rotate\ P_2\ 180), (rotate\ P_3\ 180)\} \tag{7}$$

This substitution is performed for all elements in the sets of expressions. The resulting 7 sets (whose elements no longer contain object references) are denoted as $program_{pseudo}(y)$. These steps are repeated for all input-output pairs of objects in the given ARC task. The pseudo programs generated this way are collected into one set:

$$U_{programs} = \bigcup_{\forall Y_i \in T} \{program_{pseudo}(y) | \forall y \in Y_i\} \tag{8}$$

For pseudo programs, We define an "*intersection*" and an "*emptiness*" operation. The intersection of two pseudo programs is the intersection of the sets associated with each description property. A pseudo program is empty if any set associated with a description property is empty.

Currently, these are not actual executable programs since, instead of a single expression for each description property, we have sets of expressions. In the next step, a single expression will be selected for each description property, which will become part of the actual program.

To do this, We utilize the fact that multiple demonstration examples are given for a specific task, where the same target program is executed on different inputs. Therefore, it can be assumed that the expressions that frequently appear in a specific description property of the pseudo program are likely to be parts of the target program. To find this, a counter is assigned to each expression. If We want to find the program for the output object y, We need to iterate through all pseudo programs and check if the pseudo program of y and the current pseudo program have a common intersection. If they do, the counters of the expressions present in both sets are incremented. Thus, the counter for an expression $expr$ associated with the object y can be written as follows:

$$count(y, expr) = | \{p \forall p \in U_{programs} | expr \in program_{pseudo}(p) \text{ and } program_{pseudo}(y) \cap p \neq \} | \tag{9}$$

The final program for y consists of the expressions whose counter for the respective description property is maximal:

$$program(y)^I = \underset{\forall expr \in program^I_{pseudo^{(y)}}}{argmax} count(y, expr) \tag{10}$$

where I in the upper index indicates one of the properties of the 7-element object description. We perform this for each output object and then group these output objects based on the generated program. If there are multiple expressions, the first one is selected for non-integer data structure properties (shape, color map, noise). For the others, the *"best matching"* expression is selected. This is always the query expression corresponding to the respective description property: for the vertical coordinate x, it is x_of; for the horizontal coordinate y, it is y_of; for the width property, it is $width\_of$, and for the height property, it is $height\_of$. If there is no such expression, the first one is selected, similar to the other description properties. If there happen to be multiple such expressions, the first one among the "*best matching*" expressions is selected.

The last step of induction is to find the predicates that determine which group a given object belongs to. The grouping predicates for a particular class c can be obtained by taking the anchor objects of all output objects belonging to class c and determining the predicates that are true for all these objects but false for all other input objects:

$$\begin{aligned} predicates(c) = \{ & p \in U_{predicates} | \\ & \forall y^+ \in c \\ & p(anchor(y^+)) = 1 \text{ and} \\ & \forall y^- \in \frac{C}{c} \\ & p(anchor(y^-)) = 0\} \end{aligned} \tag{11}$$

IV. Evaluation

The results of the system are shown in Table I. The system successfully solved 41 tasks from the public training set of 400 tasks and 10 tasks from the public test set. Compared to other publications, this ranks as the fourth-best result (Fig 2). In terms of execution time, the average task-solving time is 0.48 and 1.67 seconds on the training and test sets respectively. This metric has been mentioned in several other solutions as well, but due to the non-uniform execution environment, comparing them does not provide an accurate picture of the actual speed of each algorithm. Nevertheless, it can be

observed that the presented method's average execution time is orders of magnitude smaller than the others (Fig 3).

TABLE I
THE ALGORITHM'S SCORE AND AVERAGE TASK SOLVING TIME

	Training set	Public set	Private set
Tasks solved	41	10	0
Average task time (seconds)	0.48	1.67	-

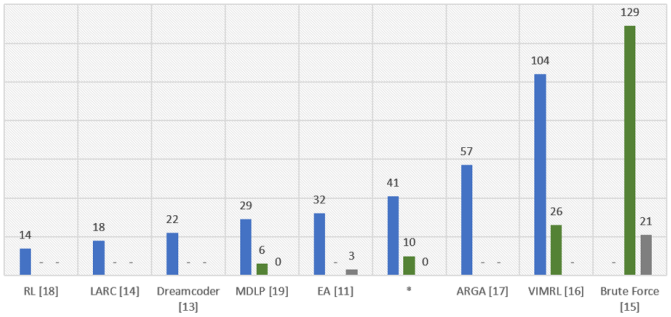

Fig. 2. The performance of different solutions on the training set (blue), public test set (green), and private test set (gray). The results of our solution are indicated in the "*" column.

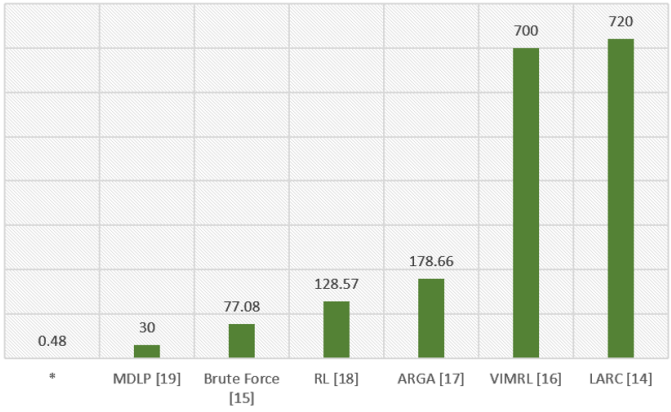

Fig. 3. The average task-solving time in seconds for different solutions. The time corresponding to our solution is indicated in the "*" column.

V. CONCLUSION

The aim of this paper was to present an induction algorithm that eliminates the infeasibly large search space by abandoning proceduralism, thereby achieving equal or better results than existing algorithms on the ARC problem. The presented system successfully reduced the size of the problem's search space. Compared to previous methods, the average task-solving time is the lowest. The algorithm was able to solve 41 tasks in the training set and 10 tasks in the public test set, which is the fourth-best result among previous publications. However, none of the private test set tasks were solved. The primary reason for the low number of solved tasks relative to the dataset's size is the number of instructions in the domain-specific language. Our main goal was to create an algorithm that avoids searching in the DSL's program tree by abandoning proceduralism not to write a complete DSL containing all operations required for solving every task. Therefore, the primary direction for further development is to expand the DSL's operation set, which will likely make it possible to solve most of the tasks in the public sets and achieve non-zero result on the private test set.

VI. ACKNOWLEDGMENTS

The authors would like to thank the Hungarian National Talent Program (NTP-HHTDK-22) for its valuable support.

REFERENCES

[1] Chollet, F., "On the measure of intelligence", *ArXiv - arXiv:1911.01547v2*, 2019.
[2] Fischer, R., Jakobs, M., Mücke, S., & Morik, K., "Solving Abstract Reasoning Tasks with Grammatical Evolution", *Proceedings of the LWDA 2020 Workshops: KDML, FGWM, FGWI-BIA, and FGDB*, pp. 6-10, 2019.
[3] Banburski, A., Ghandi, A., Alford, S., Dandekar, S., Chin, P., & Poggio, T., "Dreaming with ARC. Center for Brains, Minds and Machines", *CBMM*, 11 2020.
[4] Acquaviva, S., Pu, Y., Nye, M., Wong, C., Tessler, M. H., & Tenenbaum, J., "LARC: Language annotated Abstraction and Reasoning Corpus", *In Proceedings of the Annual Meeting of the Cognitive Science Society*, Vol. 43, No. 43, 2021.
[5] top-quarks/ARC-solution [Online], Available at: https://github.com/top-quarks/ARC-solution [Accessed: 28 July 2023].
[6] Ainooson, J., Sanyal, D., Michelson, J. P., Yang, Y., & Kunda, M., "An approach for solving tasks on the Abstract Reasoning Corpus", *ArXiv - arXiv:2302.09425*, 2023.
[7] Xu, Y., Khalil, E. B., & Sanner, S., "Graphs, Constraints, and Search for the Abstraction and Reasoning Corpus", *ArXiv - arXiv:2210.09880*, 2022.
[8] Alford, S., Gandhi, A., Rangamani, A., Banburski, A., Wang, T., Dandekar, S., ... & Chin, P., "Neural-Guided, Bidirectional Program Search for Abstraction and Reasoning", *In Complex Networks & Their Applications X: Volume 1, Proceedings of the Tenth International Conference on Complex Networks and Their Applications COMPLEX NETWORKS 2021 10*, pp. 657-668, 2022.
[9] Valeria Diaz, "Machine Learning for Detection of Cognitive Impairment", *Acta Polytechnica Hungarica*, Vol. 19, No. 5, 2022.
[10] Piroska Biró, Tamás Kádek, "The Mathability of Computer Problem Solving with ProgCont", *Acta Polytechnica Hungarica*, Vol. 19, No. 1, 2022.
[11] Man-Wen Tian, Ardashir Mohammadzadeh, Jafar Tavoosi, Saleh Mobayen, Jihad H. Asad, Oscar Castillo, Annámaria R. Várkonyi-Kóczy, "A Deep-learned Type-3 Fuzzy System and Its Application in Modeling Problems", *Acta Polytechnica Hungarica*, Vol. 19, No. 2, 2022.
[12] Zoltán M. Balogh, Alexandru Kristály, "Sharp isoperimetric and Sobolev inequalities in spaces with nonnegative Ricci curvature", *Mathematische Annalen*, Vol. 385, pp. 1747–1773, 2023.