Graphs, Constraints, and Search for the Abstraction and Reasoning Corpus

Yudong Xu,[1] **Elias B. Khalil,**[1, 2] **Scott Sanner**[1]

[1] Department of Mechanical & Industrial Engineering, University of Toronto
[2] Scale AI Research Chair in Data-Driven Algorithms for Modern Supply Chains
wil.xu@mail.utoronto.ca, khalil@mie.utoronto.ca, ssanner@mie.utoronto.ca

Abstract

The Abstraction and Reasoning Corpus (ARC) aims at benchmarking the performance of general artificial intelligence algorithms. The ARC's focus on broad generalization and few-shot learning has made it difficult to solve using pure machine learning. A more promising approach has been to perform program synthesis within an appropriately designed Domain Specific Language (DSL). However, these too have seen limited success. We propose Abstract Reasoning with Graph Abstractions (ARGA), a new object-centric framework that first represents images using graphs and then performs a search for a correct program in a DSL that is based on the abstracted graph space. The complexity of this combinatorial search is tamed through the use of constraint acquisition, state hashing, and Tabu search. An extensive set of experiments demonstrates the promise of ARGA in tackling some of the complicated object-centric tasks of the ARC rather efficiently, producing programs that are correct and easy to understand.

In an attempt to better measure the gap between machine and human learning, the Abstraction and Reasoning Corpus (ARC) was created by Chollet in 2019. The dataset is a collection of 1000 image-based reasoning tasks, where each task asks for an output image given an input. To "learn" a procedure that produces said output, each task comes with 2–5 input-output image pairs as training instances; these training inputs are different from the actual test input, but can be solved by the same (unknown) procedure. Some examples are shown in Figure 1. A competition with over 900 teams was hosted on Kaggle to solve the ARC (Kaggle 2020). Despite a massive effort, the solutions only achieved 20% accuracy on the hidden test set, at best. In fact, the first-place solution could not solve two of the three examples shown in Figure 1 despite their simplicity to a human.

Recognizing objects, actions performed on them, and relationships between them makes up a large portion of human cognition core systems (Spelke and Kinzler 2007). The ARC embodies this notion in its tasks. In fact, Acquaviva et al. (2021) found that when humans attempt to solve ARC tasks through language, half of the phrases they use relate to object detection. Therefore, an object-centric approach to solving the ARC is highly promising. Surprisingly, this key insight is yet to be leveraged.

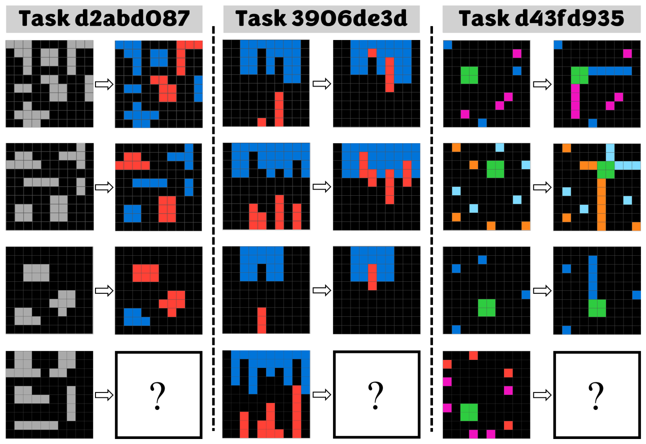

Figure 1: **Sample ARC Tasks.** Three tasks (each two consecutive columns) are shown. For a given task, each row contains one example input-output pair. The top three rows contain the "training" instances and the bottom row contains the "test" instance. The goal is to use the training instances to solve the test instance. The left task ("object recoloring") requires recoloring the size-6 grey objects to red and other grey objects to blue. The middle task ("object movement") requires moving the red columns up until they hit the blue object. The right task ("object augmentation") requires extending the size-1 objects directly above, below or to the sides of the green object towards it until they make contact.

ARGA: Abstract Reasoning with Graph Abstractions

Toward this goal, we propose Abstract Reasoning with Graph Abstractions (ARGA), an object-centric framework for solving ARC tasks. Our design rationale is to build a computationally efficient, extensible, object-aware ARC solver through careful integration of the following:

- **Representation:** Enabling object awareness requires a move from treating the input as pixels towards a *graph* of objects with spatial or other relations. We design a variety of such graph abstractions to cater to the diversity of the ARC and its different definitions of objects.
- **Structure:** Grounded in first-order logic, our graph-

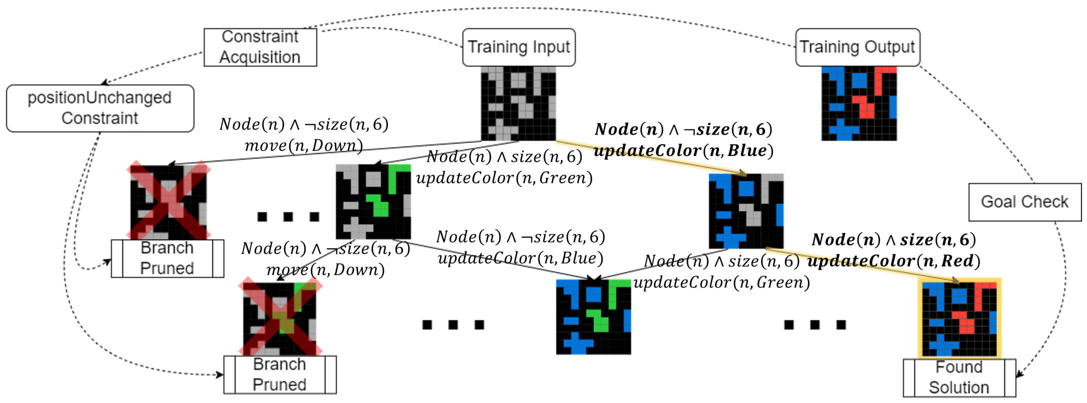
Constraint Acquisition
Training Input
Training Output
positionUnchanged Constraint
Node(n) ∧ ¬size(n, 6) move(n, Down)
Node(n) ∧ size(n, 6) updateColor(n, Green)
Node(n) ∧ ¬size(n, 6) updateColor(n, Blue)
Goal Check
Branch Pruned
Node(n) ∧ ¬size(n, 6) move(n, Down)
Node(n) ∧ ¬size(n, 6) updateColor(n, Blue)
Node(n) ∧ size(n, 6) updateColor(n, Green)
Node(n) ∧ size(n, 6) updateColor(n, Red)
Branch Pruned
Found Solution

Figure 2: **Illustration of ARGA's constraint-guided search.** Note that a reconstructed 2D image is used at each node for better visualization. Nodes in the actual search tree consists a set of abstracted graphs.

based DSL makes it possible to define complex but interpretable solution programs for tasks of the ARC. This is in contrast to pure neural network-type approaches that attempt to map input to output in an often black-box fashion.

- **Search:** With the representation and DSL in place, we opt for a complete tree search algorithm. Given a task, the search seeks a program in the DSL that produces the correct outputs for each of the task's training examples. Whenever a correct program for a task exists in our DSL, the search can find it given sufficient time.
- **Constraints:** Leveraging the observation that (solved) training examples not only tell us what a correct program does but also what it should not do (e.g., in Fig. 1 (left), objects should not move), we use constraint acquisition to simplify our combinatorial search space. Constraints are expressed in the very same graph DSL and may be acquired by an arbitrary algorithm.

Fig. 2 illustrates the DSL, Search, and Constraints components of ARGA; Fig. 4 illustrates the Representation. With ARGA, we hope to provide AI researchers who are interested in the ARC and similar few-shot reasoning situations with the first such system upon which they can build and explore the capabilities of graph and search-based reasoning. Our implementation is available on GitHub[1].

Because object-oriented abstraction and reasoning are major failure modes of state-of-the-art ARC solvers, we define criteria to select a subset of object-oriented ARC tasks as a testbed for the evaluation of our methods in comparison to other top solvers. The 160 tasks in question span a wide range of challenging problems that can be categorized as object recoloring, object movement, and object augmentation. We show how ARGA's design and performance are favorable in the following ways:

- **Extensibility and Modularity:** Every component of ARGA can be extended almost independently to target additional ARC tasks or optimize performance: novel graph abstractions can be added, additional object filters and transformations can be appended to the DSL, new search strategies can be tested, and faster constraint acquisition algorithms may seamlessly replace ours.
- **Computational Efficiency:** Our DSL contains a number of object-based selection filters as well as transformations (e.g., recoloring, moving, etc.). Because these can be composed together to form a candidate program for an ARC task, the resulting search space is combinatorially large. Nonetheless, through experiments on 160 object-based ARC tasks, we show that when ARGA finds a solution, it does so by exploring a minute number of possible solutions, effectively three orders of magnitude fewer than the winner of the Kaggle competition.
- **Effectiveness:** Our current DSL includes only 4 base filters and 11 transformations. Yet, we solve 57 of 160 tasks, only slightly behind the Kaggle winner's 64 of 160. The latter includes a much larger body of transformations that were obtained by examining many more ARC tasks.

[1]https://github.com/khalil-research/ARGA-AAAI23

System overview

We propose a two-stage framework that takes an object-centric approach to solving an ARC task. First, the *graph abstraction stage* inspired by work on Go (Graepel et al. 2001), where the 2D grid images are mapped to (multiple) undirected graph representations that capture information about the objects in the images at a higher abstracted level. Second, the *solution synthesis stage*, where a constraint-guided search is used to formulate the series of operations to be applied to the abstracted graphs that will lead to a solution. The space of possible operations is defined by an ARGA-specific relational Domain Specific Language (DSL).

Since the DSL defines operations on the abstracted graphs, we will first describe the graph abstraction stage and formally define the structure of the abstracted graphs. Then, the DSL will be defined in detail. Finally, the solution synthesis stage will be discussed.

Graph Abstraction

Graph abstraction allows us to search for a solution at a macroscopic level. In other words, we are modifying *groups*

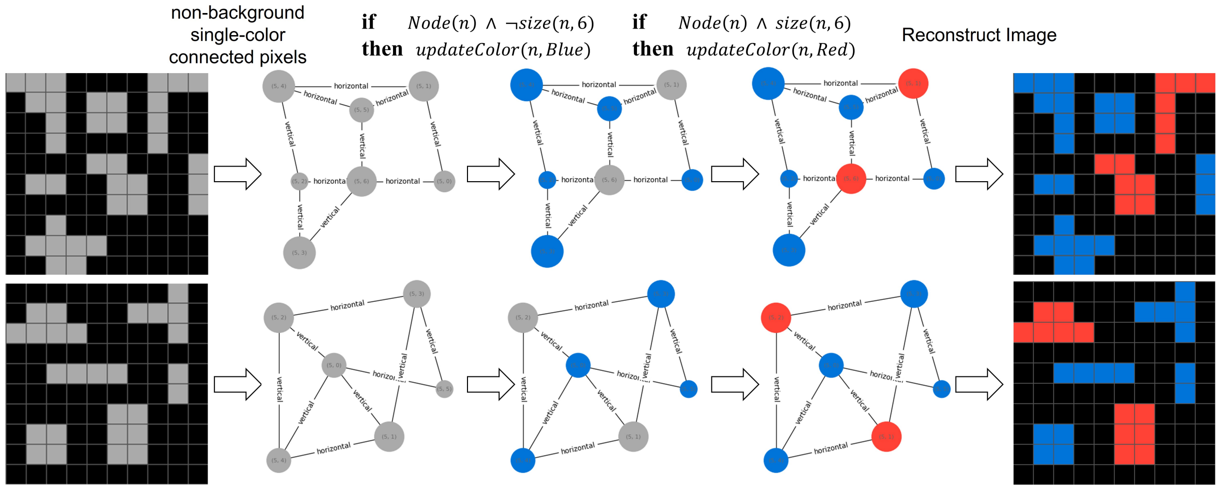
non-background single-color connected pixels
if $Node(n) \wedge \neg size(n,6)$ then $updateColor(n, Blue)$
if $Node(n) \wedge size(n,6)$ then $updateColor(n, Red)$
Reconstruct Image
horizontal
vertical

Figure 3: **Example solution generated by ARGA.** The input image is first abstracted into a graph in which each node represents a set of adjacent pixels that are not black. Two nodes share an edge iff there is at least one cell in each node with the same coordinate value along either axis. The solution here first colors in blue all nodes *not* containing exactly six pixels, then colors in red all nodes with exactly six pixels. The number of pixels contained in a node is defined as its "size", a node attribute that ARGA can use in its search for a correct program.

Object Type Set	Object Type Description
$i \in Image$	A 2D grid image
$p \in Pixel$	A pixel on an image
$g \in Graph$	An abstracted graph
$n \in Node$	A node in an abstracted graph
$e \in Edge$	An edge in an abstracted graph
$c \in Color$	Color (including *background*)
$s \in Size$	Size of a node (# pixels)
$d \in Direction$	Directions within the 2D image
$pa \in Pattern$	A pattern found on the image
$t \in Type$	Generic Type (any above)

Table 1: **Object Types** in ARGA.

Typed Object Binary Relations	Description
$containsNode(Graph, Node)$	Graph contains Node
$containsPixel(Node, Pixel)$	Node contains Pixel
$neighbor(Node, Node)$	An edge exists between two Nodes
$color(Node, Color)$	color of Node
$size(Node, Size)$	size of Node
$Rel(Type, Type)$	Generic Relation (any above)

Table 2: **Example Object Relations** in ARGA.

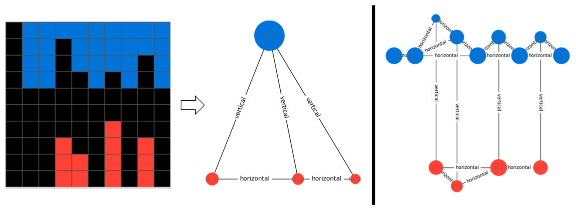

Figure 4: **Visualization of graph abstractions.** Applying two different graph abstractions to an image. Left: non-background single-color connected pixels. Right: non-background single-color vertically-connected pixels.

of pixels at once, instead of modifying each individual pixel separately. As a result, this approach has a smaller search space than its non-abstracted, raw image counterpart.

We now formally introduce terminology that aids in defining our abstracted graphs (such as those shown in Figure 3) that will be leveraged by the DSL of the next section. The language we use builds on first-order logic which provides a flexible and expressive language for describing typed objects and relations. Object types in our DSL are shown in Table 1 and can be used as unary predicates, e.g., $Node(n)$ is true *iff* $n \in Node$. Some example relations between objects are shown in Table 2 and the full set of relations can be found in Appendix Table 6.

Let i be any input or output 2D grid image from an ARC task. i can be completely specified by its set of pixels p. Let g be an abstracted graph with sets of abstracted nodes n. The relations that hold between these types are shown in Table 2. Each Node n represents an object that is detected in the original image i based on the rules of the abstraction (e.g., one graph abstraction is "non-black neighboring pixels of the same color form a node") and relations between the nodes represent relationships between these objects.

Therefore, the graph abstraction process executes a mapping that generates some abstracted graph G for image I. We note that there are multiple ways in which this mapping can be defined. Different graph abstractions can be used to identify objects in the image using different definitions of what an object is. Since the resulting abstracted graphs from different graph abstraction definitions share the same underlying structure, we are able to expand the solution space significantly without modifying the DSL.

The usefulness of having multiple definitions of an object can be observed in the example shown in Figure 1 (Middle). Upon first inspection, one may think that objects are defined as connected pixels with the same color. However, upon further inspection, we realize that the connected red pixels in different columns are in fact different objects as they do not share the same modification in the output images. Therefore, defining multiple abstraction processes improves ARGA's ability to correctly capture object information from the input images. The two different abstractions mentioned in the example are further discussed in the following:

non-background single-color connected pixels: In this abstraction, an object (or node) is defined as a set of connected pixels sharing the same color. The pseudocode for the abstraction algorithm is shown in Appendix Algorithm 1 and an illustration of this abstraction is in Figure 4.

non-background single-color vertically-connected pixels: In this graph abstraction, an object is defined as a set of vertically connected pixels that are not the background color. An illustration of this abstraction is in Figure 4.

Overlapping Objects

Note that our representation allows for a pixel to be associated with multiple nodes in the graph, as objects are modified. This can be intuitively understood by observing that objects may overlap with one another on the grid as one applies a sequence of transformations to solve a given task. Our graph abstraction ensures that although some objects may be partially obscured, they are still kept track of and considered to be a whole object. This allows the system to have the *object persistence* knowledge prior.

A Graph DSL for the ARC

We now introduce a lifted relational DSL for ARGA built upon the objects and relations defined in the previous section. The DSL is used to formally describe the filter language used to match node patterns, determine graph transformation parameters, and carry out transformations on abstracted graphs as described in the following. An example solution expressed using the DSL is shown in Figure 3.

Filters

Filters are used to select nodes from the graph. The fundamental *grammar* is a subset of first-order logic:

$$\begin{aligned} Filter(x) &::= Type(x) \\ &::= Filter(x) \wedge Filter(x) \\ &::= Filter(x) \vee Filter(x) \\ &::= \neg Filter(x) \\ &::= \exists y\, Rel(x,y) \wedge Filter(y) \\ &::= \exists y\, Rel(y,x) \wedge Filter(y) \\ &::= \forall y\, Rel(x,y) \implies Filter(y) \\ &::= \forall y\, Rel(y,x) \implies Filter(y) \\ &::= Rel(x,c) \text{ [} c \text{ is a constant]} \\ &::= Rel(c,x) \text{ [} c \text{ is a constant]} \end{aligned}$$

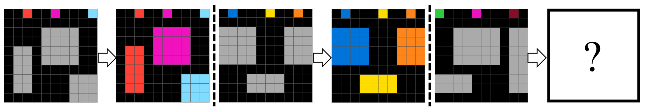

Figure 5: **Example Task from the ARC that requires dynamic transformation parameter.** The target color of a grey node is determined dynamically based on the input.

The following example filters match nodes with 6 pixels, with grey as their color, and whose neighbors are all blue, respectively:

$$\begin{aligned} \mathit{filterBySize6}(n) &\equiv \mathit{Node}(n) \wedge \mathit{size}(n,6) \\ \mathit{filterByColorGrey}(n) &\equiv \mathit{Node}(n) \wedge \mathit{color}(n,\mathit{grey}) \\ \mathit{neighborsAllBlue}(n) &\equiv \mathit{Node}(n) \\ \wedge \forall y\, \mathit{Neighbor}(n,y) &\implies \mathit{color}(y,\mathit{blue}) \end{aligned}$$

Transformations

Transformations are used to modify nodes selected by filters. They do so by modifying the values of object relations. Table 3 describes a few of the transformations; the full list can be found in Appendix Table 8.

Transformation	Description
$\mathit{updateColor}(\mathit{Node}, \mathit{Color})$	Update color of Node to Color
$\mathit{move}(\mathit{Node}, \mathit{Direction})$	Update pixels of Node to move in Direction
$\mathit{rotate}(\mathit{Node})$	Update pixels of N to rotate it clockwise
$\mathit{extend}(\mathit{Node}, \mathit{Direction})$	Add additional pixels to Node in Direction
$\mathit{transform}(\mathit{Node}, v_1, \ldots, v_k)$	Generic transformation with k parameter values $(v_1, \ldots, v_k)$

Table 3: **Example Transformations**.

An example transformation definition is shown below.

$$\begin{aligned} &\mathit{updateColor}(n : \mathit{Node}, c : \mathit{Color}) \\ &\longrightarrow \mathit{color}(n,c) \wedge \neg \mathit{color}(n,c') \quad \forall c' \in \mathit{Color} \text{ s.t. } c' \neq c \end{aligned}$$

In this example, the transformation *updateColor* updates ($\longrightarrow$) the color of the Node n to c. It does so by assigning $\mathit{color}(n,c)$ to true and $\mathit{color}(n,c')$ to false for all other colors c' in the abstracted graph representation.

Dynamic Parameter Transformations

In the example shown in Figure 1 (Left), we can "statically" identify the color that the nodes should be updated to. However, this does not work for Figure 5, because the target color of a transformed grey object is that of its neighboring size-1 object. Therefore, we define parameter binding functions which allow us to dynamically generate parameters for transformations. The grammar for parameter binding as well as its interpretation and an example are provided next:

$$\begin{aligned} Param(x, v) &::= v = c\,[c \text{ is a constant}] \\ &::= Rel(x, v) \\ &::= Rel(v, x) \\ &::= \exists y\, Rel(x, y) \land Filter(y) \land Param(y, v) \\ &::= \exists y\, Rel(y, x) \land Filter(y) \land Param(y, v) \end{aligned}$$

While it shares a similar grammar to filters, the $Param(x, v)$ has special semantics that we pause to discuss. First, the goal of $Param(x, v)$ is to find possible matching parameters for an object x, hence we never apply a filter to x in the grammar since we are not aiming to restrict it — x is assumed to be given. Second, we can interpret $Param(x, v)$ as providing all possible parameter values v that make $Param(x, v)$ true. However, we need a unique parameter v; if no v matches for a given x then $Param(x, v)$ fails to return a parameter and we cannot apply the transformation (it is considered a *noop*). If multiple v match, then we deterministically order and return the first matching v. While this is generally undesirable behavior, we note that our search over dynamic parameter bindings $Param(x, v)$ most often only succeeds when $Param(x, v)$ represents a *functional* matching such as $Param(x, v) \equiv Color(x, v)$ since we know that Color is an injective relation. Hence, we do not *a priori* restrict the grammar search to functional parameter bindings, but find in practice that successful $Param(x, v)$ bindings found in search tend to recover functional mappings from $x \mapsto v$ based on invariant properties inherent in the training examples.

We remark that this dynamic parameter grammar includes static cases such as $Param(x, v) \equiv v = blue$, which would ignore the node x and always return the parameter $blue$.

Following is a more complex parameter binding:

$$\begin{aligned} &bindSize1NeighborColor(x, v) \equiv \\ &\qquad \exists y\, neighbor(x, y) \land size(y, 1) \land Color(y, v)) \end{aligned}$$

Here, $bindSize1NeighborColor(x, v)$ matches (and returns) the color v of any neighbor of x with a size of 1 pixel. In the example shown in Figure 5, suppose we have grey Node n selected by filters; we can then find the color to update it by calling $bindSize1NeighborColor(n, Color)$.

Full Operation

With the filters, transformations and parameter bindings formally defined, we may now combine them to perform a full modification to the abstracted graph. Given a filter, a transformation, and k parameter bindings $Param_i(x, v)$ ($i \in \{1 \ldots k\}$) for each parameter taken by the transformation (possibly none if $k = 0$):

```
for each  $n \in Node$
      if  $filter(n)$
    then  $v_i \hookleftarrow \{v | Param_i(n, v)\}$ for $i \in \{1 \ldots k\}$
          $transform(n, v_1, \ldots, v_k)$
```

We assume that $\hookleftarrow$ deterministically selects a unique value v_i if $|\{v\}| \neq 1$. The set of operations required for solving the example in Figure 5 are $filterByColorGrey$, $updateColor$ and $bindSize1NeighborColor$. We note that tasks such as the example shown in Figure 3 do not require dynamic parameters; in those instances, the parameter binding found in the solution simply returns a static value $v = c$.

Solution Synthesis

With a DSL clearly defining the solution space and the input images successfully abstracted, a search will be conducted to synthesize a solution. Many ARC tasks have very complicated logic with multiple detectable objects, which means that even with our high-level graph abstraction, the search space is too large to be explored exhaustively. Therefore, the key objective in developing our algorithm is to reduce the search space. To achieve this, we introduce a *constraint acquisition* module which acquires constraints that are used to prune unpromising branches of the search tree, i.e., sequences of transformations which cannot possibly result in a correct solution to the training tasks. Other tricks such as hashing and Tabu List are also used to speed up the search. An illustration of a search tree is shown in Figure 2

Search Strategy

We implement a greedy best-first search. Suppose ARC task t has m training instances, with input-output images $\{input_i, output_i\}$ for $i \in \{1, \ldots, m\}$. Each node in our search tree contains a set of graphs $\{g_{input\_i}\}$ for $i \in \{1, \ldots, m\}$. g_{input_i} represents $input_i$ after the abstraction process and the application of a sequence of operations $(o_1, \ldots, o_j, \ldots, o_k)$ for $j \in \{1, \ldots, k\}$, where each o_j is a full operation as defined previously. The special case of $k = 0$ corresponds to no operations applied, i.e., the root node of the search tree.

To expand a node with abstracted graphs $\{g_{input\_i}\}$, we first identify the set of all valid full operations O. Then, for each $o \in O$, we apply it to $\{g_{input\_i}\}$ and obtain updated abstracted graphs $\{g'_i\}$ for $i \in \{1, \ldots, m\}$. We add the new abstracted graphs $\{g'_i\}$ into the search tree as a new node and update the sequence of operations that led to it by appending operation o to obtain $(o_1, \ldots, o_k, o)$.

Heuristic Function

To determine the node to be expanded in each iteration of the search, our primary metric measures how close the node is to the target training output. For each node, we reconstruct the corresponding 2D image for each of the abstracted graphs $\{g_{input\_i}\}$. We then compare the reconstructed images with the training outputs $\{output_i\}$ and calculate a penalty score based on pixel-wise accuracy, as detailed in Appendix Table 9. Large mismatch in pixels between the "predicted" and the actual output results in a large penalty. The node in the search tree with the lowest score is selected for expansion.

Constraint-Guided Search

We illustrate this concept with an example. All objects in Figure 1 (Left) should not change in position. We can therefore define the constraint *positionUnchanged*, which is satisfied when a node and the updated version of that node share the same set of pixels, thus making sure that the node's position on the image remains unchanged through the transformation. All transformations that modify a node's pixels can therefore be pruned by this constraint in the search tree. A visualization is shown in Figure 2.

The constraints can be defined using the same language we've introduced earlier. For instance, *positionUnchanged* is defined as:

$$positionUnchanged(n : Node, n' : Node) \equiv$$
$$\forall p \in P \;\; containsPixel(n, p) \equiv containsPixel(n', p)$$

which holds if for all pixels $p \in P$, $containsPixel(n, p)$ and $containsPixel(n', p)$ return the same value. Constraints defined for ARGA can be found in Appendix Table 10.

Given a set of constraints C that must be satisfied and a node in the search tree with a set of graphs $\{g_{input\_i}\}$ for $i \in \{1, \ldots, m\}$, the search space is pruned as follows. Suppose we have a full operation o that selects n from g_{input_i} with filter operation f and transforms it with operation t to produce updated node n'. If $\exists c \in C \wedge c(n, n') = False$, then the branch in the search tree created by applying o to $\{g_{input\_i}\}$ is pruned, as it does not satisfy constraint c.

Constraint Acquisition

To obtain a set of constraints to prune the search space, we introduce a simple constraint acquisition algorithm inspired by the ModelSeeker (Beldiceanu and Simonis 2012) and Inductive Logic Programming (Lallouet et al. 2010).

We have the generic constraint $constraint(n : Node, n' : Node)$ where n, n' can be understood as a node before and after modification by a transformation. To determine the constraints that must hold for a particular ARC task from the set of all possible constraints, we compare the training output images to the corresponding input images.

While expanding a node in the search tree, we apply the same abstraction process for the output images $\{output_i\}$ as the input images to obtain $\{g_{output\_i}\}$ for $i \in \{1, \ldots, m\}$. Then, for each full operation $o \in O$, we apply its filter operation f to g_{input_i} and g_{output_i} to obtain pairs of n_{in} and n_{out}. For each constraint c, if $c(n_{in}, n_{out})$ evaluates to True for all pairs found by f, we say that constraint c must be satisfied for all nodes selected by filter f. Therefore, all full operations o with filter f and transformation t that violate constraint c can be pruned.

Hashing

It is highly likely that different transformations or sequences of transformations result in the same abstracted graph. To avoid duplicate search efforts, we hash each node in the search tree so that equivalent nodes are only explored once. The search tree therefore has the structure of a Directed Acyclic Graph. An example of this is shown in Figure 2.

Tabu List

In our current implementation, abstracted graphs from different abstractions share the same search tree. It is therefore possible that greedy best-first search will get stuck in unpromising local solutions. To avoid this, we implement a simple Tabu List, which keeps tracks of the performance of each abstraction. If an abstraction is generating increasingly worse results, we temporarily place it on the Tabu List so that no nodes with this abstraction will be explored.

Experiments

Chollet (2019) states that the ARC aims to evaluate "Developer-aware generalization", and all ARC tasks are unique and do not assume any knowledge other than the core priors. Therefore, implementing and evaluating ARGA on a subset of ARC tasks should provide useful insight into the effectiveness of our method without the need for extensive development of transformation functions, which are not the focus of our contribution.

We focus on a subset of 160 object-centric tasks from the ARC and categorize them into three groups: (1) *Object Recoloring* tasks, which change colors of some objects in the input image; (2) *Object Movement* tasks, which change the position of some objects in the input image; (3) *Object Augmentation* tasks, which expand or add certain patterns to objects from the input images. An example task from each of the three sub-categories is shown in Figure 1.

For comparison, we evaluated the Kaggle Challenge's first-place model (top quarks 2020) on the same subset of tasks. The model was executed without the time limitation enforced by the competition and the highest-scored candidate produced by the model was used to generate the final prediction.

Results

The performance of ARGA and the Kaggle competition's first-place solution are shown in Table 4. With the exception of Object Movement tasks, our model performed slightly worse than the Kaggle winner in terms of accuracy. This is likely due to the solution space spanned by our DSL not being expressive enough, as it was developed using only a subset of the 160 tasks. On the other hand, the DSL used in the Kaggle solution was developed by first manually solving 200 tasks from the ARC (top quarks 2020).

Despite lower accuracy, ARGA achieves much better efficiency in search as we are able to reach the solution with 3 order magnitude fewer nodes explored. This suggests that with a more expressive DSL and a more efficient implementation, ARGA should be able to solve more tasks with much less search effort (ARGA is currently implemented in Python while the Kaggle solution is implemented in C++).

Furthermore, the gap between the number of tasks for which all training instances are solved (# Training Correct) and the number of tasks for which the single test instance is solved (# Testing Correct) is much smaller for ARGA. This suggests that ARGA is better at finding solutions which generalize correctly while the Kaggle solution often overfits to the training instances.

Ablation Study Table 5 shows the performance of different variations of ARGA; the accuracies are reported on all 160 tasks. We see that the use of constraint acquisition is very effective in reducing the search space, resulting in 38% lower average nodes explored before reaching the solution.

Model	Task Type	# Training Correct	# Testing Correct	Average Nodes	Average Time (sec.)
ARGA	movement	18/31 (58.06%)	17/31 (54.84%)	3830.35	89.75
	recolor	25/62 (40.32%)	23/62 (37.10%)	12316.87	326.83
	augmentation	20/67 (29.85%)	17/67 (25.37%)	4668.82	67.09
	all	63/160 (39.38%)	57/160 (35.62%)	7504.81	178.66
Kaggle First Place	movement	21/31 (67.74%)	15/31 (48.39%)	2176777.67	62.45
	recolor	23/62 (37.10%)	28/62 (45.16%)	2290441.32	93.19
	augmentation	35/67 (52.24%)	21/67 (31.34%)	2248151.10	66.07
	all	79/160 (49.38%)	64/160 (40.00%)	2249924.92	77.08

Table 4: **Results on subset of ARC.** *# Training correct* is the number of tasks that got all the training instances exactly right. *# Testing correct* is the number of tasks that got the testing instance exactly right. *Average Nodes* is the average number of unique nodes added to the search tree before finding a solution for correctly solved tasks. *Average Time (sec.)* is the average time in seconds to reach solution for correctly solved tasks.

Model	# Training Correct	# Testing Correct	Average Nodes	Average Time (sec.)
ARGA	63 (39.38%)	57 (35.62%)	7504.81	178.66
-CA	62 (38.75%)	55 (34.38%)	12114.25	227.62
-SF	60 (37.50%)	54 (33.75%)	8530.17	197.54
-TL	64 (40.00%)	57 (35.62%)	7702.53	169.52
-H	62 (38.75%)	57 (35.62%)	26107.58	172.77

Table 5: **Ablation study.** ARGA is the complete system. -CA is ARGA without constraint acquisition. -SF is ARGA using a breadth-first search strategy for abstractions. -TL is ARGA without Tabu List. -H is ARGA without hashing.

Furthermore, the results show that Tabu List, hashing, as well as the proper searching strategy are all important for the best performance. We note that as seen in Appendix Table 11, there are no significant differences in the sets of tasks solved by variations of ARGA.

Related Work

Current ARC Solvers There have been many attempts at solving the ARC. Most of those that have shown some success leverage a DSL within the program synthesis paradigm (Kaggle 2020). It has been shown that humans are able to compose a set of natural language instructions that are expressive enough to solve most of the ARC tasks, which suggests that the ARC is solvable with a powerful enough DSL and an efficient program synthesis algorithm (Johnson et al. 2021). Indeed, this is the approach suggested by Chollet (2019) when introducing the dataset: “A hypothetical ARC solver may take the form of a program synthesis engine” that “generate candidates that transform input grids into output grids.”

Solutions using this approach include the winner of the Kaggle challenge, where the DSL was created by manually solving ARC tasks and the program synthesis algorithm is a search that utilizes directed acyclic graphs (DAG). Each node in the DAG is an image, and edges between the nodes are transformations (top quarks 2020). The second-place solution introduces a preprocessing stage before following a similar approach (de Miquel Bleier 2020). Many other Kaggle top performers share this approach (Golubev 2020; Liukis 2020; Penrose 2020). Fischer et al. (2020) propose a Grammatical Evolution algorithm to generate solutions within their DSL. Alford et al. (2021) utilize an existing program synthesis system called DreamCoder (Ellis et al. 2020) to create abstractions from a simple DSL through the process of compression. The program then composes the solution for new tasks using neural-guided synthesis.

Other approaches to solving the ARC include the Neural Abstract Reasoner, which is a deep learning method that succeeds in a subset of the ARC’s problems (Kolev, Georgiev, and Penkov 2020). Assouel et al. (2022) developed a compositional imagination approach which generates unseen tasks for better generalization. Ferré (2021) develops an approach based on descriptive grids. However, these approaches have not achieved state-of-the-art results.

Constraint Acquisition (CA) is a field that aims to generate Constraint Programming (CP) models from examples (De Raedt, Passerini, and Teso 2018). State-of-the-art CA algorithms may be active, requiring interaction from the user (Bessiere et al. 2013; Arcangioli, Bessiere, and Lazaar 2016), or passive, requiring only initial examples (Bessiere et al. 2005).

The passive CA algorithm used for ARGA was influenced by ModelSeeker (Beldiceanu and Simonis 2012), which finds relevant constraints from the global constraint catalog (Beldiceanu, Carlsson, and Rampon 2005) as well as the system developed by Lallouet et al. (2010) which uses Inductive Logic Programming (ILP) and formulates constraints from logical interpretations.

Conclusion

We proposed Abstract Reasoning with Graph Abstractions (ARGA), an object-centric framework that solves ARC tasks by first generating graph abstractions and then performing a constraint-guided search. We evaluated our framework on an object-centric subset of the ARC dataset and obtained promising results. Notably, the efficiency in reaching the solution within the search space shows that with further development of the DSL, our method has the potential to solve far more complicated problems than state-of-the-art methods.

References

Acquaviva, S.; Pu, Y.; Kryven, M.; Sechopoulos, T.; Wong, C.; Ecanow, G. E.; Nye, M.; Tessler, M. H.; and Tenenbaum, J. B. 2021. Communicating Natural Programs to Humans and Machines. arXiv:2106.07824.

Alford, S.; Gandhi, A.; Rangamani, A.; Banburski, A.; Wang, T.; Dandekar, S.; Chin, J.; Poggio, T.; and Chin, P. 2021. Neural-Guided, Bidirectional Program Search for Abstraction and Reasoning. In *International Conference on Complex Networks and Their Applications*, 657–668. Springer.

Arcangioli, R.; Bessiere, C.; and Lazaar, N. 2016. Multiple constraint aquisition. In *IJCAI: International Joint Conference on Artificial Intelligence*, 698–704.

Assouel, R.; Rodriguez, P.; Taslakian, P.; Vazquez, D.; and Bengio, Y. 2022. Object-centric Compositional Imagination for Visual Abstract Reasoning. In *ICLR2022 Workshop on the Elements of Reasoning: Objects, Structure and Causality*.

Beldiceanu, N.; Carlsson, M.; and Rampon, J.-X. 2005. Global Constraint Catalog. https://hal.archives-ouvertes.fr/hal-00485396. Accessed: 2022-08-01.

Beldiceanu, N.; and Simonis, H. 2012. A model seeker: Extracting global constraint models from positive examples. In *International Conference on Principles and Practice of Constraint Programming*, 141–157. Springer.

Bessiere, C.; Coletta, R.; Hebrard, E.; Katsirelos, G.; Lazaar, N.; Narodytska, N.; Quimper, C.-G.; and Walsh, T. 2013. Constraint acquisition via partial queries. In *IJCAI: International Joint Conference on Artificial Intelligence*, 475–481.

Bessiere, C.; Coletta, R.; Koriche, F.; and O'Sullivan, B. 2005. A SAT-based version space algorithm for acquiring constraint satisfaction problems. In *European Conference on Machine Learning*, 23–34. Springer.

Chollet, F. 2019. On the Measure of Intelligence. arXiv:1911.01547.

de Miquel Bleier, A. 2020. ARC_Kaggle. https://github.com/alejandrodemiquel/ARC_Kaggle. Accessed: 2022-08-01.

De Raedt, L.; Passerini, A.; and Teso, S. 2018. Learning Constraints From Examples. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1).

Ellis, K.; Wong, C.; Nye, M.; Sable-Meyer, M.; Cary, L.; Morales, L.; Hewitt, L.; Solar-Lezama, A.; and Tenenbaum, J. B. 2020. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. arXiv:2006.08381.

Ferré, S. 2021. First Steps of an Approach to the ARC Challenge based on Descriptive Grid Models and the Minimum Description Length Principle. arXiv:2112.00848.

Fischer, R.; Jakobs, M.; Mücke, S.; and Morik, K. 2020. Solving Abstract Reasoning Tasks with Grammatical Evolution. In *LWDA*, 6–10.

Golubev, V. 2020. ARC-kaggle-3rd-Place. https://www.kaggle.com/c/abstraction-and-reasoning-challenge/discussion/154305. Accessed: 2022-08-01.

Graepel, T.; Goutrié, M.; Krüger, M.; and Herbrich, R. 2001. Learning on Graphs in the Game of Go. In *Proceedings of the International Conference on Artificial Neural Networks*, 347–352.

Johnson, A.; Vong, W. K.; Lake, B. M.; and Gureckis, T. M. 2021. Fast and flexible: Human program induction in abstract reasoning tasks. arXiv:2103.05823.

Kaggle. 2020. ARC-kaggle-main. https://www.kaggle.com/c/abstraction-and-reasoning-challenge. Accessed: 2022-08-01.

Kolev, V.; Georgiev, B.; and Penkov, S. 2020. Neural Abstract Reasoner. arXiv:2011.09860.

Lallouet, A.; Lopez, M.; Martin, L.; and Vrain, C. 2010. On Learning Constraint Problems. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, volume 1, 45–52.

Liukis, A. 2020. ARC-kaggle-5th-Place. https://www.kaggle.com/c/abstraction-and-reasoning-challenge/discussion/154377. Accessed: 2022-08-01.

Penrose, A. 2020. ARC-kaggle-8th-Place. https://www.kaggle.com/c/abstraction-and-reasoning-challenge/discussion/154384. Accessed: 2022-08-01.

Spelke, E. S.; and Kinzler, K. D. 2007. Core knowledge. *Developmental science*, 10(1): 89–96.

top quarks. 2020. ARC-solution. https://github.com/top-quarks/ARC-solution. Accessed: 2022-08-01.

Technical Details

Algorithm 1: non-background single-color connected pixels graph abstraction

Input: Grid Image *I*
Output: Abstracted Graph *G*

```
1:  Identify background color background-color
2:  Construct 2D grid graph I' for image I with node for each pixel in the image and edge between each adjacent pixel.
3:  Initialize abstracted graph G
4:  for color in all available non-background color do
5:      Find sub-graph SG of I' with node.Color == color
6:      Find connected components C of SG
7:      for component in C do
8:          Add node to G
9:          G.pixels = component
10:         G.color = color
11:     end for
12: end for
13: Add edges between nodes based on image I with relation vertical or horizontal
14: return G
```

Typed Object Binary Relations	Description
$containsNode(Graph, Node)$	Graph contains Node
$containsPixel(Node, Pixel)$	Node contains Pixel
$edgeSource(Node, Edge)$	Node is the source node for Edge
$edgeTarget(Edge, Node)$	Node is the target node for Edge
$direction(Edge, Direction)$	direction of Edge
$overlap(Node, Node)$	Two Nodes are overlapping
$neighbor(Node, Node)$	An edge exists between two Nodes
$color(Node, Color)$	color of Node
$size(Node, Size)$	size of Node
$Rel(Type, Type)$	Generic Relation (any above)

Table 6: **Full List of Object Relations** in ARGA. Further quantitative information can be introduced by implementing new relations. For example, to account for distance between two nodes, we can introduce new relation $distance(Edge, Distance)$ where Edge is the edge between the two nodes.

Filter	Description
$filterByColor(Node, Color)$	return True if Node has Color
$filterBySize(Node, Size)$	return True if Node is of Size
$filterByNeighborColor(Node, Color)$	return True if Node has neighbor with Color
$filterByNeighborSize(Node, Size)$	return True if Node has neighbor with Size

Table 7: **Base Filters**.

Transformation	Description
$updateColor(Node, Color)$	Update color of Node to Color
$move(Node, Direction)$	Update pixels of Node to move 1 pixel in Direction
$moveMax(Node, Direction)$	Update pixels of Node to move in Direction until it collides with another node
$rotate(Node)$	Update pixels of Node to rotate it clockwise
$fillRectangle(Node, Color)$	Fill background nodes in rectangle enclosed by the node with Color
$hollowRectangle(Node, Color)$	Color all nodes in rectangle enclosed by the node with Color
$addBorder(Node, Color)$	Add additional pixels to Node in Direction
$insertPattern(Node, Pattern)$	Insert Pattern at Node
$mirror(Node, Pixel, Direction)$	Mirror Node toward Direction around Pixel
$extend(Node, Direction)$	Add additional pixels to Node in Direction
$flip(Node, Direction)$	Flip Node in place in some direction
$transform(N, [k])$	Generic transformation with k parameters.

Table 8: **Full List of Transformations**.

Actual	Predicted	Penalty
Background	Non-background	2
Non-background	Background	2
Non-background	Non-background wrong color	1
Non-background	Non-background right color	0
Background	Background	0

Table 9: **Heuristic Function used in Search**

Constraint	Description
$positionUnchanged(Node, Node)$	Node does not change position after update
$colorUnchanged(Node, Node)$	Node does not change color after update
$sizeUnchanged(Node, Node)$	Node does not change in size after update
$constraint(Node, Node)$	Generic constraint

Table 10: **Example Constraints**.

	ARGA	-CA	-SF	-TL	-H
ARGA	57	54	53	56	56
-CA	54	55	51	53	54
-SF	53	51	54	53	52
-TL	56	53	53	57	55
-H	56	54	52	55	57

Table 11: **Solved Tasks Overlaps**.