

# Abductive Symbolic Solver on Abstraction and Reasoning Corpus\*

Mintaek Lim<sup>1,†</sup>, Seokki Lee<sup>1,†</sup>, Liyew Woletemaryam Abitew<sup>1,†</sup> and Sundong Kim<sup>1,\*</sup>

<sup>1</sup>Gwangju Institute of Science and Technology

## Abstract

This paper addresses the challenge of enhancing artificial intelligence reasoning capabilities, focusing on logic within the Abstraction and Reasoning Corpus (ARC). Humans solve such visual reasoning tasks based on their observations and hypotheses, and they can explain their solutions with a proper reason. However, many previous approaches focused only on the grid transition and it is not enough for AI to provide reasonable and human-like solutions. By considering the human process of solving visual reasoning tasks, we have concluded that the thinking process is likely the abductive reasoning process. Thus, we propose a novel framework that symbolically represents the observed data into a knowledge graph and extracts core knowledge that can be used for solution generation. This information limits the solution search space and helps provide a reasonable mid-process. Our approach holds promise for improving AI performance on ARC tasks by effectively narrowing the solution space and providing logical solutions grounded in core knowledge extraction.

## Keywords

Abstraction and Reasoning Corpus, Abductive Reasoning, Knowledge Graph, Domain Specific Language

## 1. Introduction

Artificial intelligence nowadays exhibits impressive problem-solving skills in many domains. Though they provide valuable assistance, not all responses make sense due to the hallucination issue and lack of logical stability. According to Pan Lu et al., especially within the category of mathematical reasoning, logical reasoning, and numeric commonsense, AI agents underperformed compared to other areas such as scientific, statistical, and algebraic reasoning. Moreover, the "puzzle test" and "abstract scene" tasks showed averagely the biggest performance gap between current AI models and humans [1]. To enhance such weaknesses, various experiments have been conducted on logic and puzzle test datasets [2, 3, 4, 5]. Datasets corresponding to such categories that require complex logical capabilities with visual images are called Visual Reasoning tasks.

As an IQ test is one of the representative measurements of human intelligence, Abstraction and Reasoning Corpus (ARC) was invented by François Chollet to measure the intelligence of an AI [2]. The ARC dataset has 400 tasks in each training and evaluation set and each consists of multiple numbers of example pairs and a test pair as shown in Figure 1. The task is to formulate a pattern that applies to all the example pairs and then construct an answer with the given test input grid. All tasks are created based on four core knowledge priors, which are 1) objectness, including object cohesion, persistence, and its influence via contact, 2) goal-directedness, 3) numbers and counting, and 4) basic geometry and topology [2]. Due to these characteristics, solutions that have defined domain-specific languages (DSL) have emerged. Unlike other AI techniques, two representative solutions have utilized DSLs to make the essence of each not dissolved into a vector but preserved symbolically. Moreover, the performances have resulted in 1st place in the Kaggle ARC solving competition and ARCAthon 2022 [6, 7]. Therefore, this research focuses on the symbolic representation of the ARC by applying DSLs and synthesizing DSLs for the solution.

---

LNSAI 2024: First International Workshop on Logical Foundations of Neuro-Symbolic AI, August 05, 2024, Jeju, South Korea

\*Correspondance to Sundong Kim (sundong@gist.ac.kr)

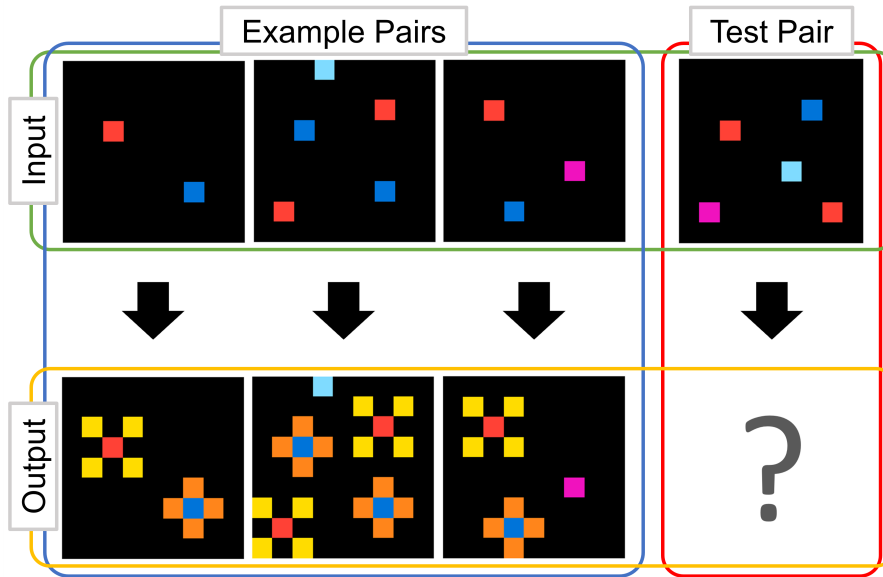
†These authors contributed equally.

✉ victorlim@gist.ac.kr (M. Lim); sklee1103@gm.gist.ac.kr (S. Lee); woleteml@gm.gist.ac.kr (L. W. Abitew); sundong@gist.ac.kr (S. Kim)

ORCID 0009-0001-4070-6927 (S. Lee); 0000-0001-9687-2409 (S. Kim)



© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).



**Figure 1:** Example ARC task. Solvers are supposed to formulate a pattern that applies to all the given example pairs and then construct an answer with the given test input grid.

Since the transformer-based models are considered the best-performing AI, various researchers have challenged solving ARC tasks with texts by providing additional descriptions [8], applying different prompting skills [9], or estimating hypotheses between the input and output grids [10]. However, such solutions can be improved in the following two ways, 1) by using a symbolic network to generate solutions that are understandable from the human perspective, and 2) by following human thinking processes to make solutions more reasonable and human-like. As humans explain their thoughts to verify their understanding, it is necessary to check both the solution and the answer for reasoning tasks. Thus, this research proposes a symbolic solver that returns understandable and reasonable solutions.

In visual reasoning, humans establish hypotheses based on their observation [11]. Inductive reasoning is well-known as a method to generate general solutions with sufficient observations, however, finding the best solution under limited observations is appropriate with abductive reasoning. Due to such property, the human thinking process of solving the ARC is more likely abductive reasoning. In each pair of the ARC task, the transition between two grids could be represented with multiple hypotheses including 1) what has changed, 2) how or how much it has changed, and 3) why it has changed in such a way. Considering the reason for the transition is the key to this research. In Figure 1, four orange pixels appeared around the blue pixel. With only the first pair, it is hard to guarantee a pattern for this task. Observing the second and third pairs provides more clues for formulating a solution. After checking all pairs, the reason for the orange pixel pattern can now be understood, ensuring that the target is blue. In other words, the color is the reason for the pattern not the other fundamental properties like position or counts.

Many previous approaches missed such information and struggled to select a target object to apply the pattern in the solution generation step. By emphasizing the weight of repeated features, we propose an experiment that extracts core knowledge which are the candidate arguments for the solution, and finds common transformations that utilize the extracted information to estimate the result. Our paper's contribution is two-fold, 1) it delineates the conversion of ARC tasks into knowledge graphs and the subsequent extraction of core knowledge from these graphs, and 2) it presents an abductive symbolic solver that utilizes the extracted core knowledge.

## 2. Related Works

**Domain Specific Languages (DSL)** In tackling the ARC challenge, some researchers have designed DSLs by referencing specific ARC tasks and refining them after solving training tasks. While these DSLs prove their systematic stability through successful example pair augmentation based on handcrafted solutions, their adaptability to unseen tasks is limited [7, 12]. Recent studies have explored integrating neurodiversity-inspired methods with computational intelligence through DSLs. One such system, the Visual Imagery Reasoning Language (VIMRL), simulates human mental imagery processes in neurodivergent individuals but struggles to generalize across diverse ARC tasks [13]. Another study uses DreamCoder synthesis to create symbolic abstractions from solved tasks and design a reasoning algorithm, however, this approach heavily relies on previously solved tasks, making it less effective in novel situations [14].

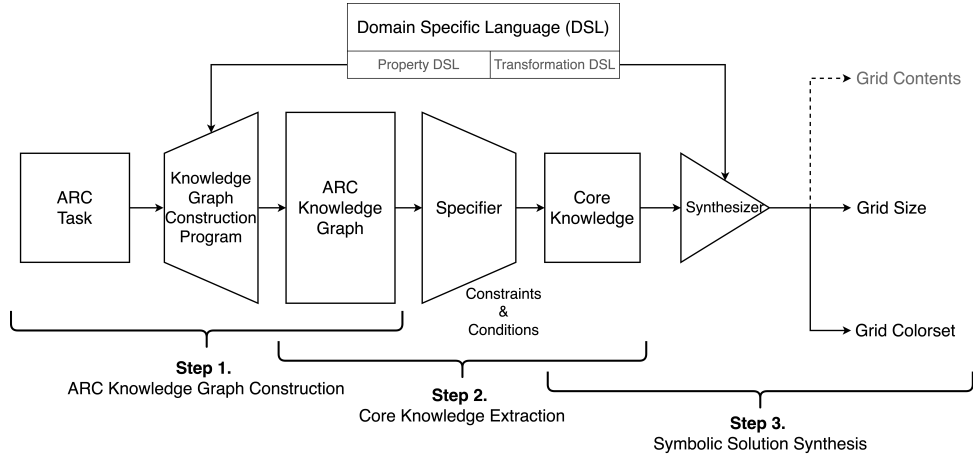
**Graph in ARC** The paper "Abstract Reasoning with Graph Abstractions (ARGA)" proposed using a graph-based representation to abstract input images into nodes and edges [15]. This method captures spatial and relational information but struggles with the complexity of real-world visual reasoning tasks due to its reliance on predefined graph structures and constraints, limiting its flexibility in diverse scenarios. Lastly, the paper "Mimicking Human Solutions with Object-Centric Decision Transformer" proposed an object-construction algorithm by transforming the ARC grid into a graph to cluster the nodes based on their distances [16]. Since the aim of the paper is limited to defining an object within only one layer of the graph, the current research gained motivation to expand the graph space by detecting multiple features.

**Abductive Reasoning** Abductive reasoning is a type of logical inference aimed at determining the simplest and most probable explanation from observations. It is used in fields like logistics, design synthesis, and visual reasoning [11, 17, 18, 19]. Liang et al. introduced a task to evaluate machine intelligence in visual scenarios through abductive reasoning. This approach, reflecting human cognition via Observation (O) and Explanation (H), influenced our understanding of the ARC task [11].

**Program Synthesis** Program synthesis has shown significant advancements in recent years, particularly in the context of the Abstraction and Reasoning Corpus (ARC). Various approaches have been proposed to tackle the complexities of synthesizing programs that can generalize well from a few examples. For instance, [20] developed techniques for automating string processing in spreadsheets using input-output examples, laying the foundational work for programming by example (PBE) systems. [21] extended this by synthesizing more complex programs beyond domain-specific languages. A notable contribution is the Semantic Interpreter by [22], which leverages large language models (LLMs) to translate natural language user utterances into executable programs within a domain-specific language (DSL) tailored for Microsoft Office applications. This approach highlights the effectiveness of combining natural language processing with program synthesis, particularly in productivity software. Building on the idea of leveraging structured domain knowledge, [23] introduced a Divide-Align-Conquer strategy for program synthesis. Their method addresses the exponential growth of the search space in program synthesis by decomposing tasks into smaller, manageable subtasks. This approach utilizes structural alignment to guide the search process, significantly improving the efficiency and accuracy of program synthesis in structured domains such as string transformations and visual reasoning tasks in the ARC. By employing analogical reasoning and the Structure-Mapping Theory (SMT), their agent, BEN, outperforms traditional inductive logic programming (ILP) methods, demonstrating the potential of decomposition-driven synthesis in handling complex program generation.

### 3. Method

This chapter will describe the overall structure of solving ARC tasks symbolically with abductive reasoning. The framework shown in Figure 2 can be divided into three main stages: 1) ARC Knowledge Graph (ARCKG) construction, 2) core knowledge extraction from the knowledge graph, and 3) solution searching using extracted core knowledge. Each of the steps is further described in Sections 3.1, 3.2, and 3.3, respectively.



**Figure 2:** Overall framework of Symbolic ARC Solver. To tackle ARC tasks from the symbolic perspective, the first step involves generating a corresponding knowledge graph using a construction program based on defined Domain Specific Languages (DSL). (Step 1, Chapter 3.1) Then, extract core knowledge from the knowledge graph using *Specifier*. (Step 2, Chapter 3.2) Since all the ARC tasks consist of multiples of example pairs and a test pair, we define *Specifier* to hold only the repeated conditions that appeared in all example pairs. Lastly, search solutions under given constraints using *Synthesizer*. (Step 3, Chapter 3.3) The information gained from the examples and proposing *Transformation DSLs* limits the solution search space and makes the search feasible.

#### 3.1. ARC Knowledge Graph Construction

In this knowledge graph construction step, each example pair in the task becomes one unit of ARC Knowledge Graph (ARCKG). For example, a problem in Figure 1 will have four ARCKGs (three examples and one test pair). ARCKG has four layers in total and it is to organize the nodes and edges well with their origin and characteristics. Based on this 4-layer structure, the construction rule is defined using DSL to apply human understanding to the ARC task and to form a database. DSL in the ARC domain could be categorized into two; *Transformation DSL* and *Property DSL*, and only the *Property DSLs* are used to construct ARCKG. *Transformation DSLs* are used in *Synthesizer* which is explained in Chapter 3.3. In the following three sub-chapters, the definition of DSL, the structural frame of ARCKG, and the detailed process of the construction are described.













































##### 3.1.1. Domain Specific Language Definition






When humans observe the ARC task, they don't only identify the changes or differences on the surface but also why such changes occurred. According to how Michael Hodel designed his DSL for the ARC [7], property, and util DSLs are the one that composes the reason for the transformation. In other words, for the complete solution of the ARC, such DSLs are supposed to be preceded before the transformation. Since this research proposes to use a knowledge graph as a source of core knowledge, ARCKG is designed to contain information that could be the key argument of the *Transformation DSL*. Thus, mainly the DSL which represents the property of an object or a pixel is used for the ARCKG construction.



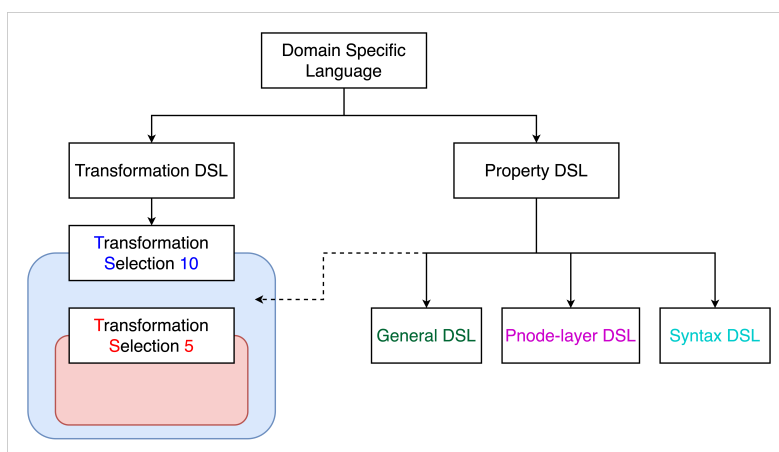
**DSL Categories - Property DSL** This research proposes DSLs that are classified into two categories based on their purpose. DSLs that symbolize the properties of nodes are referred to as *Property DSLs* and are primarily used to draw edges in the knowledge graph. Refer to the Figure 3 to see what properties are defined. There are several conditions to draw an edge, such as when two nodes have the same property, when a node has a specific property, or when one node is contained within another node by some property. This category is further divided into more specific categories: *General* and *Pnode layer*. The former applies to all layers, generating edges, while the latter applies only to the Pnode layer. *Syntax DSLs* handle the syntactical elements of DSLs and form the backbone of constructing the knowledge graph. They, in turn, are divided into DSLs for generating edges, creating nodes, and combining the two lists, ultimately resulting in the knowledge graph being stored in the form of *nodelist* and *edgelist*.

**DSL Categories - Transformation DSL** *Transformation DSLs* are utilized in the symbolic ARC solver and play a role in predicting the answer by applying transformations to the given nodes. Some of them belong to both *Property DSL* and *Transformation DSL* simultaneously, and the detailed classification is shown in Figure 3. The reason is due to the ARCKG structure that is defined to have only four types of nodes. The argument of a *Transformation DSL* is

Linear	 	get_number_of_nodes	  	get_background_color		Make_Edge_list	
get_identity_match	 	get_color_difference_set		get_dimension		Make_Pnode_list	
Onode_count	 	get_dimension_difference		get_color_of_node		Make_Onode_list	
get_number_of_color_set	 	get_dimension		get_coordinate		Make_Gnode_list	
get_subset_match		get_dominant_color		is_square		Make_Vnode_list	
get_superset_match		get_least_common_color		is_rectangle		Concat_list	
get_union		get_height_difference		is_ring		get_vertical_index	
get_height	 	get_width_difference		is_symmetric		get_manhattan_distance	
get_width	 	get_center_nodes		get_horizontal_index		get_polar_distance	

**Legend**  Transformation-S5  Transformation-S10  General DSL  Pnode-layer DSL  Syntax DSL

**Figure 3:** Overview of Domain-Specific Languages (DSLs) and their category tag.



**Figure 4:** The taxonomy of the Domain-Specific Language (DSL). The terms *Transformation DSL* and *Property DSL* are equivalent to the DSL used in *Synthesizer* and ARCKG construction respectively. In particular, *Transformation DSLs* do not follow the traditional ones, such as move, flip, or rotate due to the experimental setup of this research. *Transformation Selection 10 (TS10)* contains a selection of suitable DSLs for the experiment and *TS5* is a subset of it. *General DSL* takes the majority of the *Property DSL* and represents the characteristics of the object. Similarly, *Pnode-layer DSL* only appears in Pnode-layer and forms the fundamental feature of object forming. *Syntax DSL* contains node and edge list generation functions to store the information in the form of *NodeList* and *EdgeList*.

**Data Types** In the realm of Domain-Specific Language (DSL), data types form the backbone of how information is represented, manipulated, and interpreted. Table 1 provides an overview of the key data types utilized in our DSL, each tailored to facilitate the unique requirements of nodes and their symbolic relationships.

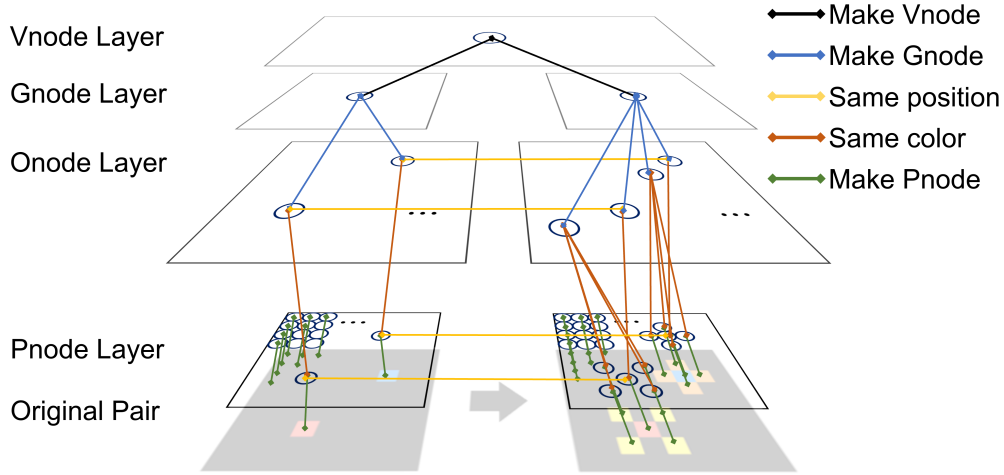
**Table 1**  
Description of Data Types Used in creating DSLs

<i>Data type</i>	<i>Description</i>
<i>Pnode</i>	Represent a single pixel in the grid. Stores grid coordinates.
<i>Onode</i>	Represent objects in the grid formed by a collection of <i>Pnode</i>
<i>Gnode</i>	Represent the entire grid holding <i>Pnode</i> and <i>Onode</i> as one node.
<i>Vnode</i>	Represent a pair of input and output into one node that holds two <i>Gnodes</i> .
<i>Xnode</i>	Represent any type of node above.
<i>Edge</i>	Represent relationship between <i>Pnode</i> , <i>Onode</i> , <i>Gnode</i> , and <i>Vnode</i> (provides connection in the graph).
<i>Color</i>	Represent a color of pixel by integer value.
<i>NodeList</i>	Represent a list of nodes.
<i>EdgeList</i>	Represent a list of edges.
<i>Coordinate</i>	Is used to represent coordinates which is a tuple of two integer values.
<i>ColorSet</i>	Is used to hold collections of color.

### 3.1.2. ARC Knowledge Graph Structure Definition

The original ARC data is provided in the form of a two-dimensional array, where each element of the array contains information corresponding to colors, ranging from 0 to 9. Therefore, it is challenging for machines to understand and infer rules from this data due to its limited information content. Thus, we propose a method to convert the 2D grid into a knowledge graph that captures information perceived by humans when viewing ARC problems. The knowledge graph is formed as units of one input-output example pair. A single knowledge graph consists of four layers, each characterized by the attributes of the nodes included in it. When representing the original ARC task's example pairs as  $Task = \{(I_1, O_1), (I_2, O_2), \dots, (I_n, O_n)\}$  the corresponding knowledge graphs are expressed as  $ARCKG = \{g_1, g_2, \dots, g_n\}$ , where  $g_n$  is further represented as  $g_n = \{NodeList_n, EdgeList_n\}$ . Each  $NodeList_n$  in  $g_n$  is a data structure containing all nodes found in the four layers, and  $EdgeList_n$  is a data structure containing all edges found in the respective knowledge graph. The detailed description of each of the four layers is as follows.

- **Pnode layer:** This first layer converts each pixel into a single node named *Pnode* and captures the relationships between these *Pnodes*, representing them as edges.
- **Onode layer:** This second layer contains nodes representing sets of one or more pixels forming objects. It captures the relationships between objects as edges. Nodes in this layer, which is named *Onode*, are connected to the *Pnodes* with edges.
- **Gnode layer:** This third layer represents the entire input or output grid as a single node named *Gnode*. Nodes in this layer are connected to all nodes in the lower layers including the first and second with edges.
- **Vnode layer:** This fourth layer combines the input and output grid into a single node. Each example pair is ultimately represented by one fourth-layered *Vnode*, which is connected to two *Gnodes* from the third layer through edges.



**Figure 5:** An example of a straightforward, and almost backbone-structured knowledge graph of the first pair of Figure 1. In practice, the ARCKG generated by Algorithm 1 can contain up to millions of edges. The graph consists of four layers, with edges freely drawn between layers as well as between input and output by the *Property DSL*. The yellow edges represent connections between two nodes at the same position. The other (black, blue, green) indicate edges signify that nodes in the lower layer constitute nodes in the upper layer.

### 3.1.3. ARC Knowledge Graph Construction Program

Algorithm 1 is an example pseudo-code for building an ARCKG using defined DSLs and graph structure. This program takes a task as input and returns the corresponding knowledge graphs. It consists of two stages: one for generating node lists from the grid and another for creating edges. The algorithm is composed of a nested loop structure. The outer loop iterates over each example pair of the input task. Then, it iterates over the input and output grids of each pair to create node lists. At the end of line 7, two node lists are generated as a result of lines 3 to 7, named *node\_list\_input* and *node\_list\_output*, respectively. Lines 8 to 9 merge these two node lists and create the very top-layer Vnode, appending it to the *node\_list*. The loop from lines 10 to 12 applies all possible *Property\_DSL* to the *node\_list*, drawing edges using *Make\_Edge\_list*.

---

#### Algorithm 1 ARC Knowledge Graph Construction Program

---

**Require:** ARC Task

```

1:  $KG \leftarrow$  empty set
2: for each pair in Task do
3:   for each grid in pair do
4:      $node\_list \leftarrow Make\_Pnode\_list(grid)$ 
5:      $node\_list \leftarrow Make\_Onode\_list(node\_list)$ 
6:      $node\_list \leftarrow Make\_Gnode\_list(node\_list)$ 
7:   end for
8:    $node\_list \leftarrow Concat\_list(node\_list\_input, node\_list\_output)$ 
9:    $node\_list \leftarrow Make\_Vnode\_list(node\_list)$ 
10:  for each Property_DSL do
11:     $edge\_list \leftarrow Make\_Edge\_list(node\_list, Property\_DSL)$ 
12:  end for
13:  add ( $node\_list\_pair, edge\_list$ ) to  $KG$ 
14: end for
15: return  $KG$ 

```

---

## 3.2. Core Knowledge Extraction

In this step, the goal is to extract information that is considered useful for the solution. A unit named *Specifier* takes a knowledge graph as input and returns objects that satisfy the constraints. It plays a role in filtering out relatively less helpful knowledge graph components to narrow down the search space in *Synthesizer*. The conditions of this filter are gathered by analyzing all the given example pairs. Since the ARC is a few-shot task, humans are supposed to formulate a solution from given pairs and apply it to the test grid after only observing a single grid. Conversely, the solution must be appropriately applied to all the examples. Therefore, the solution that we are aiming to find is the intersection of possible solutions of all the given pairs. Moreover, the components of the solution could be found from the ARCKGs established in the previous step, by counting the nodes with identical properties. The following sub-chapters describe the concept of the *Specifier* unit and how it operates differently on example pairs and a test grid.

### 3.2.1. Specifier

*Specifier* is designed to select candidate objects from the test input grid and by doing so, the afterward solution search space decreases substantially. The object selection in the test grid must be done based on a rule, driven from given example pairs. For instance, the task in Figure 1 has three example pairs and we can detect the grid changes around a specific pixel. Since the red and blue pixels appear in all three pairs, those pixels in the test grid can become a candidate component for the solution. Accordingly, due to the changes in the grid around those pixels also appearing three times, the solution is concluded to apply such transformations with the corresponding target pixels. Here, in the task given with  $n$  example pairs, selecting the objects, features, or changes observed  $n$  times is critical in *Specifier*. Abductive reasoning in one sentence is to make the best prediction from incomplete observations. Suppose there always is an absolute solution in every ARC task. Due to the incompleteness of the ARC said by the creator François Chollet, given example pairs may or may not express the rule of the task precisely. Thus, the solution-, object-, and constraint-finding process based on the abduction is employed in this research.

**Core Knowledge** The term "core knowledge" refers to an output of the *Specifier*. The narrow range of meaning relates only to the candidates of objects that satisfy the conditions, while the wider range of meaning includes their intrinsic properties. On the surface, *Specifier* unit appears to return only the object on the grid. In contrast, since an object is equivalent to a node in ARCKG, edges that either originated from or ended with the node are also the information on the table. By utilizing the features of selected objects, the *specifier* concludes constraints for object selection in the test grid.

### 3.2.2. Train and Test of the Specifier

**Train of the Specifier** The training phase of the *Specifier* means the constraint update process for specifying the objects during the example pair observation. Specifically, the update begins from the second pair. During the first pair, there are no objects that can be specified for the solution due to the absence of constraint. Thus, the entire objects and the input grid itself which refer to Onodes and a Gnode become the candidates. This set of objects is then exported to the *Synthesizer*. Regardless of whether the *Synthesizer* discovers the solution path, *Specifier* in the following example starts to filter out the object without any feature in common compared to the object candidates from the previous iteration. The conditions of an object are either a property or a relationship with other components which respectively refer to the term feature and edge. From the second iteration, the constraints of the *Specifier* gathered based on the features and edges of the object are modified. Since no ARC task contains less than two example pairs, this abduction process of updating constraints occurs at least once. After the final update in the last iteration, the constraints are fixed and further used for the test phase.

**Test of the Specifier** During the test phase, the module processes the ARCKG of the test grid. Due to the absence of the output grid, some edges that connect nodes across the grid are not considered. The core knowledge should be driven only from half of the knowledge graph. The trained constraints allow *Specifier* to achieve such a goal under the concept of the task. In short, *Specifier* in the test phase searches for nodes that satisfy the conditions gathered from the example pairs and returns candidate components that can be the material for the solution.

### 3.3. Symbolic Solution Synthesis

In this step, a solution of an ARC task is discovered by synthesizing *Transformation DSLs* and core knowledge driven from the ARCKG by the *Specifier* unit. A module named *Synthesizer* takes the role of searching through all the combination spaces. Since the solution-finding process follows the brute-force search, theoretically it is solvable under the assumption that the provided DSLs completely cover the task. Moreover, as the *Synthesizer* unit exploits the syntheses of *Transformation DSL* from the example pairs when solving the test case, the following paragraph explains the operation based on the train and test phase.

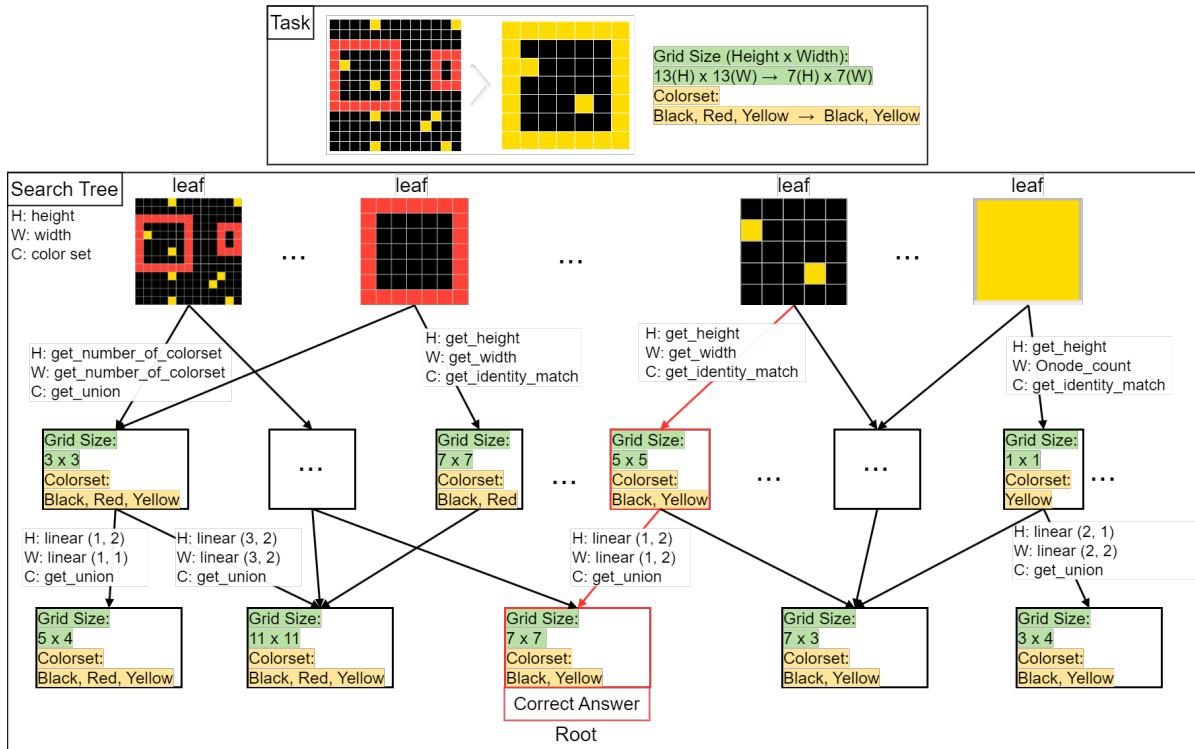
#### 3.3.1. Synthesizer

The *Synthesizer* unit takes core knowledge and *Transformation DSLs* as input and finds the combination of them to be the desired answer. While humans formulate hypothetical solutions and update them during the example pair observation, *Synthesizer* narrows down the number of solutions. Similar to the object node abduction in the *Specifier*, only the solution that is applicable for all the examples remains after the training phase. Further, when the component reaches the test grid, it exploits the exact solution from the train and returns the answer. Figure 6 below depicts the initial solution search space of the first example pair of a task.

#### 3.3.2. Abductive Symbolic Solver

The *Solver*, refers to a union of *Specifier* and *Synthesizer* unit, has an equivalent meaning with the term "abductive symbolic solver" or "symbolic ARC solver" in this paper. It utilizes the concept of abductive reasoning for the learning stage which unfolds in reverse order of inference, starting from the *Synthesizer*. The process begins with each node of the input graph treated as a leaf and extends up to the root of the output grid, exploring all possible paths through the search tree. The edges of the search tree are composed of *Transformation DSLs*, originating from the leaves and branching towards the root by applying each *Transformation DSL*. The search halts when the tree reaches a certain depth, at which point the paths connected to the root become candidates for core knowledge used in the inference stage. At this stage, it identifies all possible (*node, path*) pairs, where the path represents the sequence of *Transformation DSLs*. The term "path" refers to the sequence of *Transformation DSLs* (Domain-Specific Languages) applied within the search tree during the process of abductive reasoning to reach a solution for a given ARC task. Applying this path to the nodes in the pair yields the desired output targeted during the process. An example of the *Synthesizer*'s training can be found in Figure 6, which corresponds to the setup in Section 4.1 and is an example of *Synthesizer-10*. The *Synthesizer* starts with nodes representing parts of the input grid. It applies transformations like *get\_height*, *get\_width*, *get\_number\_of\_colorset*, etc., in sequence. The path through these transformations is formed until the output node, representing the desired solution, is reached.

The *Specifier* generates a function that identifies the minimal features in the knowledge graph that uniquely designate the node, returning them as constraints. The objective of this process is to traverse the knowledge graph and find the smallest subset that satisfies the criteria of the given node, such as "same color," "adjacent pixels," and "largest." Consequently, the constraint becomes a function that extracts node(s) in the knowledge graph, ultimately generating a hypothesis in the form of a pair (*constraints, path*). This hypothesis can be applied to all knowledge graphs of the same task by the



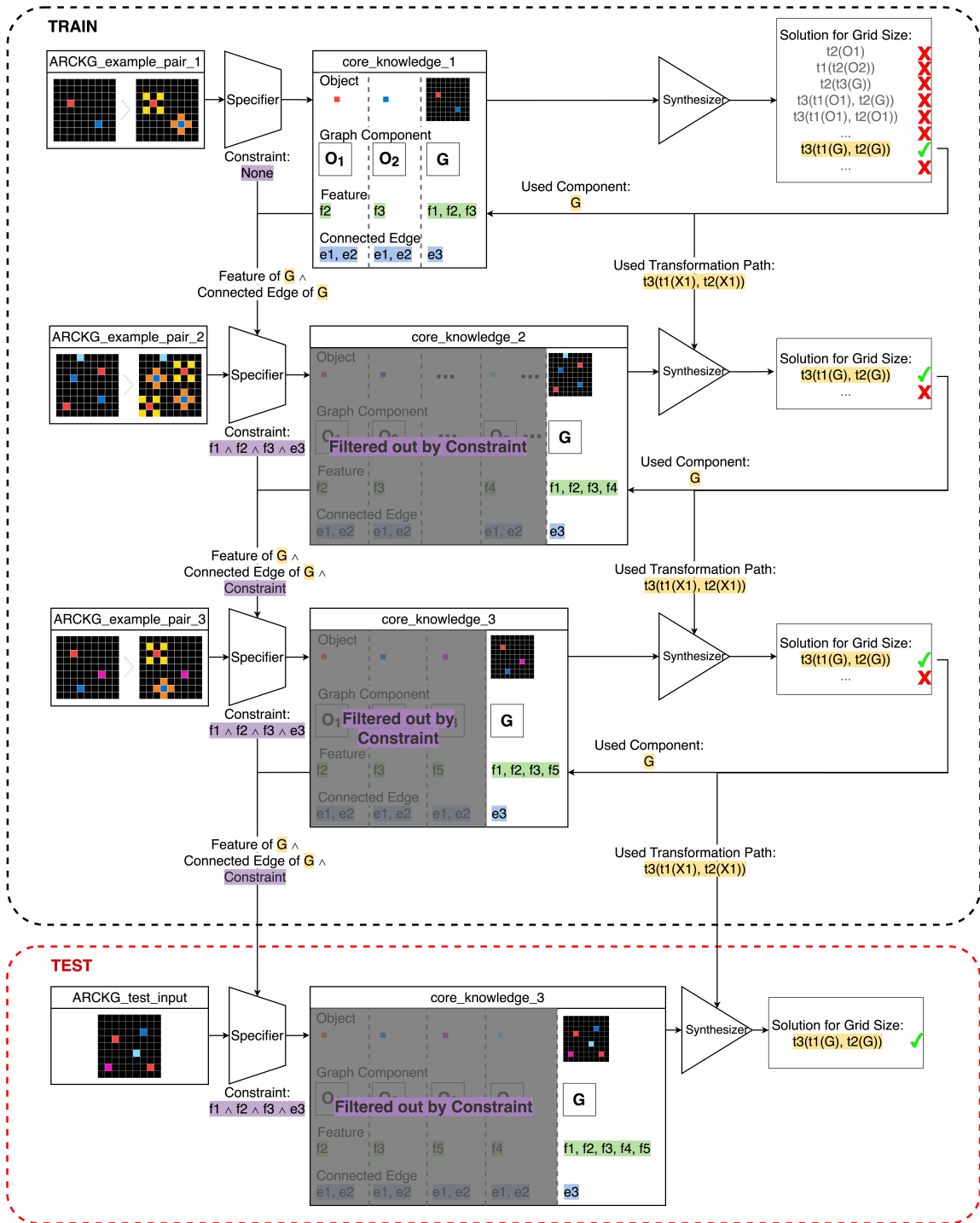
**Figure 6:** Training session of the *Synthesizer* and its expanded search tree. The task is to find the largest rectangle in the input and change the color to its interior single-pixel color. First, all nodes generated from the input are placed at the top (leaf) of the search tree, with the output node at the bottom (root), commented as Correct Answer in the figure. Then, *Transformation DSLs* are applied to draw paths. This example shows *Synthesizer-10* targeting grid size and color set. Among the DSLs used, *get\_height* returns the height of the node, *get\_width* returns the width, *get\_number\_of\_colorset* returns the number of colors other than the background, and *Onode\_count* returns the number of included objects. The *linear(a, b)* DSL performs the transformation  $ax + b$  on the previous value  $x$ . *get\_union* returns the union of colors between the previous node and the target node, while *get\_identity\_match* returns the color set of the previous node. The path that reaches the root is highlighted in red, forming a pair with the corresponding leaf.

following method:

$$path(constraints(KG)) \Rightarrow prediction$$

It means that by applying the sequence of transformations defined by the "path" to the nodes and information extracted from the knowledge graph (based on the given constraints), the model can generate a prediction or solution for the task. Essentially, the constraints filter and guide the application of transformations, ensuring that only relevant parts of the knowledge graph are used to derive the final prediction. After obtaining a set of possible hypotheses from the observations of the first example pair, the final solution is adopted through the process of evaluating whether these hypotheses can consistently explain other observations. Due to the nature of ARC problems, observations are highly limited by the number of example pairs and exhibit characteristics of few-shot learning. By applying hypotheses to the given pairs and iteratively selecting only those hypotheses that correctly derive the answers, the remaining hypotheses are adopted as the final solution for this task. This solution ensures that our observations are well explained.





**Figure 7:** Overall demonstration of proposing symbolic ARC solver. The process consists of two steps, the train phase with given example pairs and the test phase with test input. The starting node indicates the ARCKG constructed using the respective example pair. The initial state of the *Specifier* has no constraint and makes the entire set of detected objects become core knowledge. The *Synthesizer*, can only refer to the graph components which are either Onode or Gnode, thus the candidates do not include other types of node. Throughout the entire combination space, only a few solution paths satisfy the answer and are further utilized for constraint updating. Since the output grid is considered the answer, verification of the result is feasible. The intrinsic core knowledge of the used component, which refers to the feature and connected edge in this diagram, affects the constraint of the following step. The combination of *Transformation DSLs*, the path that leads to the answer, is also transferred to the *Synthesizer* in the next step with generalized form using any node  $X1$ ). From the second iteration, *Specifier* and *Synthesizer* follow the conditions of the past. After the training phase, the final conditions are then applied to each unit, and utilize the ARCKG made of the test input grid to yield the answer.

## 4. Experiment & Result

The primary objective of this experiment is to leverage a knowledge graph (KG) and *Domain Specific Languages* to solve tasks within the Abstraction and Reasoning Corpus (ARC). Below are the hypotheses raised in this paper:

- **H1:** The knowledge graphs effectively encapsulate symbolic knowledge, facilitating human-like problem-solving and enhancing performance.
- **H2:** The number of *Transformation DSLs* is positively correlated with the performance of the symbolic ARC solver.

### 4.1. Experimental Setup

To evaluate the performance of the DSL-based symbolic Arc solver, we conducted experiments with two distinct setups: one utilizing a knowledge graph and another without it. This comparison aims to assess the impact of knowledge graphs on the solver’s accuracy in predicting the ARC task outputs (grid size and color set).

**Target Elements:** The answers (outputs) of ARC problems consist of three elements: 1) the size of the grid, 2) the color set of the grid, and 3) the contents of the grid. Though all three are crucial, predicting and modifying the target values of the first two hold significant importance as they represent steps inherent in human problem-solving of ARC tasks. Therefore, we prioritized these aspects in our experimental setup, focusing primarily on them and enabling the utilization of minimal and straightforward *Transformation DSLs* during the synthesis process. For the color set, all colors appearing in the correct grid must be matched with the predicted value to be considered as the correct answer, while for the grid’s size, separate integer values for height and width were predicted.

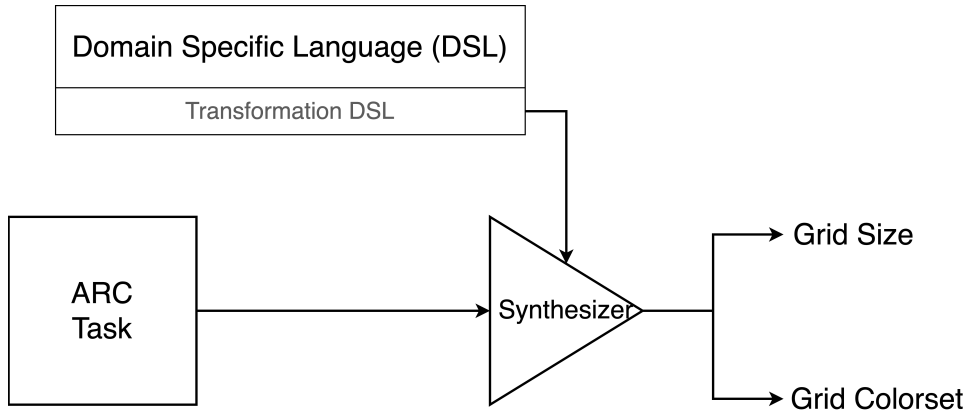
**Approach using Knowledge Graphs:** In this experiment a total of 22 *Property DSLs* were employed to build a graph encapsulating the symbolic information of the grid elements. Based on the transformations defined in the DSLs the solver generates potential solutions followed by the *Synthesizer* selecting the most accurate solutions by leveraging the information from the target node extracted by the *Specifier* from the knowledge graph.

**Approach without Knowledge Graphs:** This setup is similar to the experiment with the knowledge graph construction, but without the intermediate step of graph construction. The transformations DSLs are directly applied to the grid elements to generate potential solutions. Thus In this experimental setup, no *Specifier* is needed since the goal of a *Specifier* is to extract the unique characters of nodes from the knowledge graph. The overall flow of how we experimented without the knowledge graph is depicted in Figure 8.

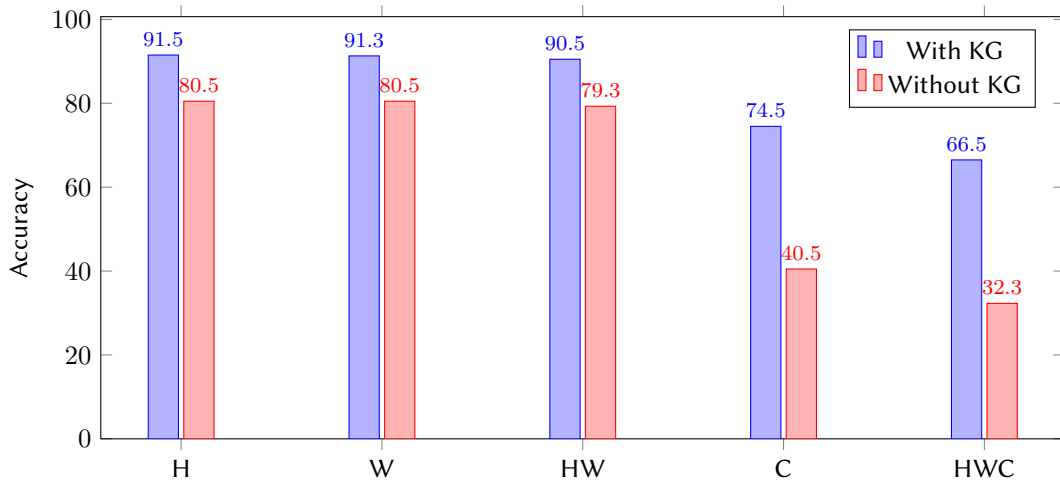
**Experimental Procedure:** A set of ARC tasks (400 tasks) was selected for the experiments ensuring a diverse range of grid sizes and color sets. For the KG approach graphs were constructed for each task using the 22 *Property DSLs*. Then both solvers (KG-based solvers and non-KG-based solvers) run on the tasks to predict the grid size and color set. The accuracy of the solvers was measured based on the correctness of the grid size (height and width) and color set.

### 4.2. Result

**Comparison of Solver Performance with and without the Use of Knowledge Graph** Figure 9 presents the accuracy scores of a solver’s performance on different target values, comparing the use of a knowledge graph against not using one. For each target on the x-axis, the solver’s accuracy is consistently higher when utilizing the knowledge graph. In particular, when not utilizing the knowledge



**Figure 8:** Systematic schema of the experiment without knowledge graph. Since the knowledge graph is not used, the process of graph construction and core knowledge extraction are omitted. Accordingly, only the *Transformation DSLs* are used.



**Figure 9:** Accuracy score comparison of solver with and without utilizing knowledge graph on each target. Here, "KG" refers to the knowledge graph. The targets assessed are Height (H), Width (W), Color (C), and their combinations: Height and Width (HW), and Height, Width, and Color (HWC).

graph, a significant decrease in the prediction performance of C and HWC can be observed. This indicates the crucial role of symbolic information contained in the knowledge graph in predicting the color set. The solver achieves nearly perfect accuracy for the H, W, and HW with the knowledge graph. These results confirm that the use of knowledge graphs effectively enhances performance, supporting **H1** by demonstrating their capability to encapsulate symbolic knowledge and facilitate human-like problem-solving.

**Differences in Performance by Size of Synthesizer** To explore the relationship between the number of *Transformation DSLs* and accuracy, two *Synthesizers* of different sizes were prepared, both with a depth limit of 2 for the search tree. The results show that *Synthesizer-10* consistently achieves higher accuracy across all categories compared to *Synthesizer-5*. Notably, in the HWC category, *Synthesizer-10* outperforms *Synthesizer-5* by over three times. These findings support H2, confirming that the number of *Transformation DSLs* is positively correlated with the performance of the symbolic ARC solver. Additionally, this suggests that employing more sophisticated and diverse *Transformation DSLs* enhances the model's accuracy and its potential to predict content.

**Table 2**

The comparison presented here delves into the accuracy scores of solvers utilizing different *Synthesizer* sizes. *Synthesizer-10*, employing 10 *Transformation DSLs*, is contrasted with *Synthesizer-5*, which utilizes only 5. For details on DSL adopted by each *Synthesizer*, see Figure 3.

	Synthesizer-10			Synthesizer-5		
	Correct	Incorrect	Accuracy (%)	Correct	Incorrect	Accuracy (%)
<b>H</b>	366	34	91.5	209	191	52.25
<b>W</b>	365	35	91.25	203	197	50.75
<b>C</b>	299	101	74.75	176	224	44
<b>HW</b>	362	38	<u>90.5</u>	197	203	<u>49.25</u>
<b>HWC</b>	266	134	<u>66.5</u>	84	316	<u>21</u>

## 5. Conclusion

We introduced a framework for ARC problem-solving, integrating knowledge graph conversion and abductive reasoning learning with a symbolic ARC Solver. This approach, inspired by human thought processes, offers systematic, interpretable, and scalable solutions. Leveraging knowledge graphs, we decode ARC tasks symbolically, providing crucial insights for inferring problem rules. Impressively, even with a naive *Synthesizer* using limited *Transformation DSLs*, our framework achieves high accuracy in predicting grid sizes (90.5%) and color sets (74.5%). Furthermore, as DSLs increase, we anticipate significant performance improvement, potentially extending to grid content prediction.

## Acknowledgments

This work was supported by the IITP (RS-2023-00216011, No. 2019-0-01842), AICA (HPC-AI) and the GIST (HPC-AI) funded by the Ministry of Science and ICT, Korea.

## References

- [1] P. Lu, H. Bansal, T. Xia, J. Liu, C. Li, H. Hajishirzi, H. Cheng, K.-W. Chang, M. Galley, J. Gao, Mathvista: Evaluating mathematical reasoning of foundation models in visual contexts, arXiv preprint arXiv:2310.02255 (2023).
- [2] F. Chollet, On the measure of intelligence, 2019. arXiv:1911.01547.
- [3] J. Raven, Raven progressive matrices, in: Handbook of nonverbal assessment, Springer, 2003, pp. 223–237.
- [4] S. Antol, A. Agrawal, J. Lu, M. Mitchell, D. Batra, C. L. Zitnick, D. Parikh, Vqa: Visual question answering, in: Proceedings of the IEEE international conference on computer vision, 2015, pp. 2425–2433.
- [5] D. Ghosal, V. T. Y. Han, C. Y. Ken, S. Poria, Are language models puzzle prodigies? algorithmic puzzles unveil serious challenges in multimodal reasoning, arXiv preprint arXiv:2403.03864 (2024).
- [6] Top-quarks, Arc-solution, <https://github.com/top-quarks/ARC-solution>, 2021.
- [7] M. Hodel, arc-dsl, <https://github.com/michaelhodel/arc-dsl>, 2022.
- [8] Y. Xu, W. Li, P. Vaezipoor, S. Sanner, E. B. Khalil, Llms and the abstraction and reasoning corpus: Successes, failures, and the importance of object-based representations, arXiv preprint arXiv:2305.18354 (2023).
- [9] S. Lee, W. Sim, D. Shin, S. Hwang, W. Seo, J. Park, S. Lee, S. Kim, S. Kim, Reasoning abilities of large language models: In-depth analysis on the abstraction and reasoning corpus, arXiv preprint arXiv:2403.11793 (2024).
- [10] R. Wang, E. Zelikman, G. Poesia, Y. Pu, N. Haber, N. D. Goodman, Hypothesis search: Inductive reasoning with language models, arXiv preprint arXiv:2309.05660 (2023).

- [11] C. Liang, W. Wang, T. Zhou, Y. Yang, Visual abductive reasoning, 2022. arXiv:2203.14040.
- [12] M. Hodel, Addressing the abstraction and reasoning corpus via procedural example generation, arXiv preprint arXiv:2404.07353 (2024).
- [13] J. Ainooson, D. Sanyal, J. P. Michelson, Y. Yang, M. Kunda, A neurodiversity-inspired solver for the abstraction & reasoning corpus (arc) using visual imagery and program synthesis, arXiv preprint arXiv:2302.09425 (2023).
- [14] S. Alford, A. Gandhi, A. Rangamani, A. Banburski, T. Wang, S. Dandekar, J. Chin, T. Poggio, P. Chin, 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*, Springer, 2022, pp. 657–668.
- [15] Y. Xu, E. B. Khalil, S. Sanner, Graphs, constraints, and search for the abstraction and reasoning corpus, arXiv preprint arXiv:2210.09880 (2022). Available: <https://arxiv.org/abs/2210.09880>.
- [16] J. Park, J. Im, S. Hwang, M. Lim, S. Ualibekova, S. Kim, S. Kim, Unraveling the arc puzzle: Mimicking human solutions with object-centric decision transformer, arXiv preprint arXiv:2306.08204 (2023).
- [17] G. Kovács, K. M. Spens, Abductive reasoning in logistics research, *International journal of physical distribution & logistics management* 35 (2005) 132–144.
- [18] S. C.-Y. Lu, A. Liu, Abductive reasoning for design synthesis, *CIRP annals* 61 (2012) 143–146.
- [19] P. Thagard, C. Shelley, Abductive reasoning: Logic, visual thinking, and coherence, in: *Logic and Scientific Methods: Volume One of the Tenth International Congress of Logic, Methodology and Philosophy of Science*, Florence, August 1995, Springer, 1997, pp. 413–427.
- [20] S. Gulwani, Automating string processing in spreadsheets using input-output examples, *ACM Sigplan Notices* 46 (2011) 317–330.
- [21] X. Chen, C. Liu, D. X. Song, Towards synthesizing complex programs from input-output examples. arxiv, *Learning* (2018).
- [22] A. Gandhi, T. Q. Nguyen, H. Jiao, R. Steen, A. Bhatawdekar, Natural language commanding via program synthesis, arXiv preprint arXiv:2306.03460 (2023).
- [23] J. Witt, S. Rasing, S. Dumančić, T. Guns, C.-C. Carbon, A divide-align-conquer strategy for program synthesis, arXiv preprint arXiv:2301.03094 (2023).