Generalized Planning for the Abstraction and Reasoning Corpus

Chao Lei, Nir Lipovetzky, Krista A. Ehinger

School of Computing and Information Systems, The University of Melbourne, Australia
clei1@student.unimelb.edu.au, {nir.lipovetzky, kris.ehinger}@unimelb.edu.au

Abstract

The Abstraction and Reasoning Corpus (ARC) is a general artificial intelligence benchmark that poses difficulties for pure machine learning methods due to its requirement for fluid intelligence with a focus on reasoning and abstraction. In this work, we introduce an ARC solver, Generalized Planning for Abstract Reasoning (GPAR). It casts an ARC problem as a generalized planning (GP) problem, where a solution is formalized as a *planning program* with *pointers*. We express each ARC problem using the standard Planning Domain Definition Language (PDDL) coupled with *external functions* representing object-centric *abstractions*. We show how to scale up GP solvers via domain knowledge specific to ARC in the form of restrictions over the actions model, predicates, arguments and valid structure of planning programs. Our experiments demonstrate that GPAR outperforms the state-of-the-art solvers on the object-centric tasks of the ARC, showing the effectiveness of GP and the expressiveness of PDDL to model ARC problems. The challenges provided by the ARC benchmark motivate research to advance existing GP solvers and understand new relations with other planning computational models. Code is available at github.com/you68681/GPAR.

Introduction

Abstract visual reasoning tasks have been used to understand and measure machine intelligence (Małkiński and Mańdziuk 2023; Barrett et al. 2018; Moskvichev, Odouard, and Mitchell 2023). One of these tasks, the Abstraction and Reasoning Corpus (ARC) introduced by Chollet (2019), remains an open challenge. ARC tasks are challenging for machines because they require object recognition, abstract reasoning, and procedural analogies (Johnson et al. 2021; Acquaviva et al. 2022). ARC comprises 1000 unique tasks where each task consists of a small set (typically three) of input-output image pairs for training, and generally one or occasionally multiple test pairs for evaluation (Figure 1). Each image is a 2D grid of pixels with 10 possible colors. ARC tasks require inferring the underlying rules or procedures from a few examples based on *core knowledge* priors including objectness, goal-directedness, numbers and counting, topology and geometry.
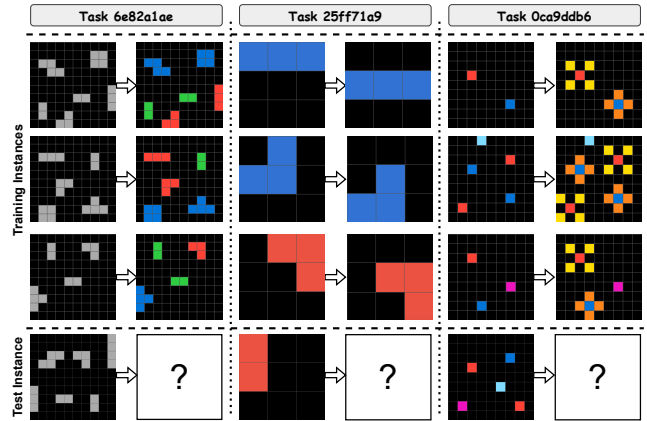
Figure 1: Three example tasks from the ARC. For a given task, each row contains an input-output image pair as a training instance, and the bottom row is the test instance. The goal of the solver is to learn from the training instances how to generate the output for the test instance.

Chollet (2019) suggested a hypothetical ARC solver that includes a program synthesis engine for candidate solutions generation within a "human-like reasoning Domain Specific Language (DSL)". Few successful solvers have followed this approach. Inspired by human strategies for solving ARC tasks (Johnson et al. 2021; Acquaviva et al. 2022), Xu, Khalil, and Sanner (2023) proposed an object-centric approach, Abstract Reasoning with Graph Abstractions (ARGA) that adopts a graph-based DSL representation and performs a constraint-guided search to find programs in the DSL solving the task. ARGA demonstrates considerable generalization ability and efficient search. However, due to the limited expressiveness of the DSL, its performance is worse for some ARC tasks than the Kaggle competition's first-place solution (top quarks 2020). This algorithm searches in a directed acyclic graph to synthesize program solutions over a hand-crafted DSL, where each search node represents an image transformation applied to its parent node.

Generalized planning (GP), a program synthesis approach that studies the representation and generation of solutions that are valid for a set of problems, is well suited to the ARC (Srivastava, Immerman, and Zilberstein 2008; Hu

and De Giacomo 2011; Jiménez, Segovia-Aguas, and Jonsson 2019). Solutions, known as generalized plans, can be formalized as *planning programs* with *pointers* (Segovia-Aguas, Jiménez, and Jonsson 2019) where conditional statements, and looping and branching structures allow the compact representation of solutions. Recent advances in GP solvers have significantly improved the search efficiency, enabling the applicability of GP over new challenging benchmarks (Lei, Lipovetzky, and Ehinger 2023).

In this work, we propose an ARC solver called Generalized Planning for Abstract Reasoning (GPAR), which models each ARC task as a generalized planning problem and adopts a state-of-the-art planner to perform program synthesis. We improve existing graph abstractions to promote greater object awareness and introduce a novel DSL based on the Planning Domain Definition Language (PDDL) (Haslum et al. 2019), where hybrid declarative and imperative modeling languages are combined to guarantee enough expressivity, and represent the transition function concisely. Our main contributions are: 1) a novel method to solve abstract reasoning tasks based on generalized planning, which achieves the state-of-the-art performance over the ARC benchmark; 2) an encoding based on PDDL which enables the adoption of alternative planning models for visual reasoning; 3) the usage of novel ARC domain knowledge that other ARC solvers can use to reduce the size of the solution space.

Background

Planning Domain Definition Language   PDDL, the *de facto* standard modeling language for several different classes of planning problems, allows the usage of automated planning solvers to find plans that map an initial state into one of the goal states of a transition system (Haslum et al. 2019). PDDL divides the representation of a planning problem into two parts, a *domain* $\mathcal{D}$ to define the *predicates* and *action schemes*, consisting of preconditions and effects, whose parameters can be instantiated with a typed-system of constant objects, and a *problem* or *instance* $\mathcal{I}$, defining the *objects*, *initial state*, and *goal formula* that entails a set of goal states. Different problems with the same domain can be created by changing any problem definition element: objects, initial state or goal conditions.

The induced transition system can be traversed through the application of actions. In fact, a plan is typically composed of a sequence of actions. To be applicable, the action preconditions need to be true in a state, and the resulting state is generated by incorporating the effects of the action, where some ground atoms of a predicate become true or false. Preconditions and effects are generally described through formulas in first-order logic. It is known that certain effects can be described more concisely by alternative languages or simulators, better equipped to reason over complex numeric operations, for example Dornhege et al. (2009). We use PDDL with *external functions* whose denotation is specified using imperative languages to express complex preconditions and effects of the ARC tasks (Frances et al. 2017).
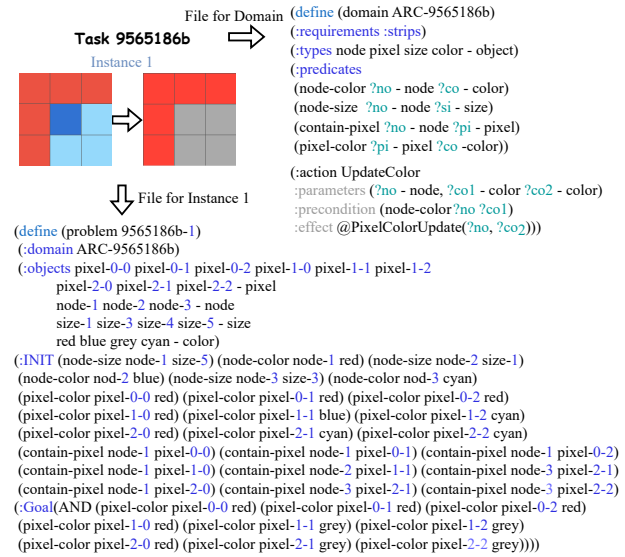

Figure 2: A PDDL example for a fragment of an ARC task.

Figure 2 presents a PDDL domain and instance file for a fragment of an ARC task. Parameters of action schemes and predicates are preceded by the "?" symbol, and external functions are preceded by the "@" symbol.

Generalized Planning   GP aims to solve a finite set of classical planning problems $\mathcal{P}$ over the same domain $\mathcal{D}$, where each instance $\mathcal{I}$ may differ in the initial state I, goal conditions G, or objects Δ. A GP solution is a single *program* that produces a valid plan for every classical planning instance.

Planning Programs with Pointers   *planning programs* with *pointers* Z, where each pointer indexes a type of object in $\mathcal{P}$, compactly describe a scalable solution space for GP (Segovia-Aguas et al. 2022). A planning program Π, is a sequence of programmable instructions, i.e. $\Pi = \langle w_0, \ldots, w_{n-1} \rangle$, with a given maximum number of program lines n.

An instruction w_i can either be a planning action instantiated from the action scheme over pointers or constant objects, a RAM action to manipulate pointers, a test action to return the interpretation of a predicate, a goto instruction for non-sequential execution, or a special end instruction for termination that is always programmed in the last line. A goto instruction is a tuple goto(i', y), where i' is the destination line, and y is a proposition that captures the result of the last execution of RAM or test action. We refer the reader to Segovia-Aguas, Jiménez, and Jonsson (2019) for a full specification.

The upper part of Figure 3 illustrates a planning program discovered by our solver that updates the color of any size-1 node (a collection of pixels) to black using two pointers, no and co, to iterate over node and color objects. The bottom part illustrates how a single planning action can represent a large set of object-instantiated action executions. The inner loop, lines 0 to 4, updates a size-1 node no with color co to black. If the precondition of the action *UpdateColor* is false (see Fig 2), then the effect of the action will not be
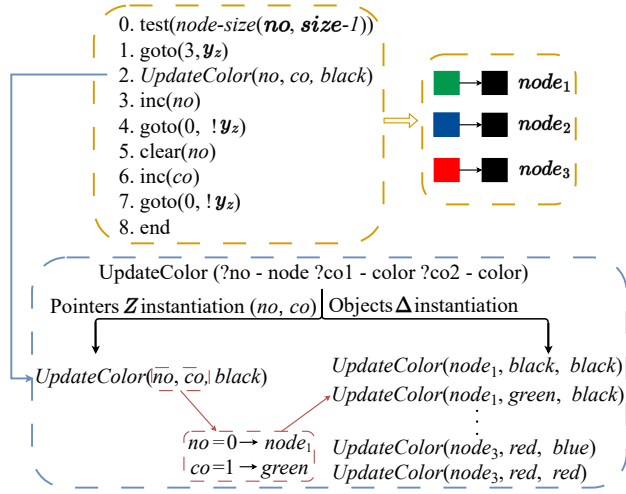
Figure 3: A planning program Π to alter size-1 nodes with different colors to black. Pointers no and co index node and color objects.

executed. When no points to the last node object, line 3 fails to increment the pointer, and lines 5-7 are executed to set no to the first node, and let co point to the next color. When all colors have been tried, then the program will end.

Abstraction over ARC

Abstraction enables object awareness in GPAR to allow actions to modify a group of pixels at once rather than individually, resulting in a smaller search space. Object cohesion is central to human visual understanding (Spelke and Kinzler 2006), and humans doing ARC tasks seem to come up with solutions that involve objects and object relations (Acquaviva et al. 2022; Johnson et al. 2021). However, part of the challenge of the ARC tasks comes from the fact that there are multiple ways to interpret the images, and different tasks may require different "objects". Therefore, we consider multiple possible abstract representations. As in Xu, Khalil, and Sanner (2023), we represent an image as a graph of *nodes* representing objects and their spatial relations.

Inspired by Xu, Khalil, and Sanner (2023), we consider the following abstractions: 1) *4-connected*, which treats 4-connected components as nodes, excluding the background; 2) *8-connected*, which treats 8-connected components as nodes, excluding the background; 3) *same-color*, which treats all pixels of the same color as a node, regardless of their connectivity; 4) *multi-color*, which treats all non-background colors as the same for the purposes of forming 4-connected and 8-connected components (thus allowing the creation of multi-colored nodes); 5) *vertical* and *horizontal*, which form nodes of columns or rows, respectively, of same-colored non-background pixels; 6) *pixels*, which treats each pixel as a node; 7) *image*, which treats the entire image as a single node; 8) *max-rectangle*, which recognizes the maximum rectangle that can be inscribed within a 4-connected component as a node and subsequently processes the remaining pixels as 4-connected components, in both non-background and background regions.
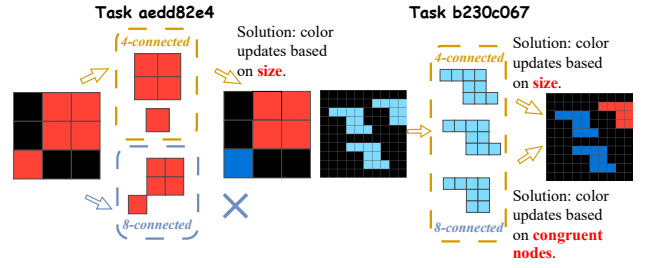

Figure 4: An example task for which the 4- vs. 8-connected abstractions produce different nodes (left), and an example task where identified nodes are the same (right).

Different node definitions can compensate for the limitations of a certain abstraction, such as in Figure 4 (left), where only the 4-connected abstraction is reasonable. However, two abstractions may produce identical nodes for a given ARC task, as in Figure 4 (right). To avoid duplication, we only consider an abstraction if it produces a different node representation in terms of size, color, or shape for at least one training instance.

Node Attributes. Each identified node is associated with basic attributes, including color, size, and shape. Shape can be a *single pixel*, *square*, *rectangle*, *horizontal line*, *vertical line*, *left diagonal line*, *right diagonal line* or *unknown*. To address tasks involving counting or sorting objects, nodes with the largest and smallest size, odd and even size, and most and least frequently occurring color are also indicated.

For some abstractions, the aforementioned attributes are inappropriate, and alternative attributes are used. For multi-color nodes, the color attribute is omitted. When considering either pixel nodes or image nodes, only the most and least frequent color are identified. For pixel nodes, we use additional attributes to represent image geometry, denoting which nodes are on the image borders, centric-diagonal, middle-vertical and middle-horizontal lines and to detect and remove pixels that are potentially noise (defined as 4-connected components with a size of 1 pixel).

Relations between Nodes We define three types of node relations, *spatial*, *congruent*, and *inclusive*, applicable to all node definitions except for pixel and image nodes. Spatial relations, right, left, up, and down, exist between two nodes iff there is at least one pixel in each node with the same coordinate value along either axis. Diagonal spatial relations are considered for two nodes whose shapes are not *unknown* and whose corners align on the same diagonal axis. Congruent relations are defined between nodes with identical shapes and sizes, and nodes with the same color. Inclusive relations specify which nodes contain or partially contain other nodes. A node contains another node if all the pixels of the contained node lie within the borders of the containing node. A node partially contains another node if the above relation holds and the borders of the contained node touch the boundaries of the image. Node attributes and relations are sourced from *core knowledge* priors and extracted through standard image processing approaches.

Object Types	Possible Associated Objects
NODE	*node-0, ..., node-n.*
PIXEL	*pixel-0-0, ..., pixel-29-29.*
COLOR	*color-0, ..., color-9.*
SIZE	*size-1, ..., size-9000.*
STEP	*one, max.*
ROTATION	$90°, 180°, 270°$.
F-DIRECTION	*vertical, horizontal, left-diagonal, right-diagonal.*
M-DIRECTION	*left, right, up, down, left-up, left-down, right-up, right-down.*
SHAPE	*single-pixel, square, rectangle, vertical-line, horizontal-line, left-diagonal-line, right-diagonal-line, unknown.*

Table 1: Object types and associated objects in our DSL.

Domain-Specific Language

PDDL leverages a subset of first-order logic, a powerful tool to represent knowledge for reasoning purposes. It provides a structured and concise way to express relations between objects and properties (Genesereth and Nilsson 1987; Levesque 1986). PDDL describes each ARC task through a single domain file and a finite set of instance files, one for each input-output image pair. The domain file contains the relations between nodes and their attributes, modeled as predicates, and node transformations, modeled by action schemes. Action schemes and predicates are instantiated through the objects specified in the instance file, where the conjunctive formula of instantiated predicates describes the initial state representing an input image, and a goal state modeling the target image configuration.

Given an ARC task, Table 1 shows the available objects and their types, while Table 2 presents the available predicates to model node attributes and their relations. We differentiate between predicates that can be interpreted by the `test` action to condition a `goto` instruction, indicated by the test_p column, and predicates whose main purpose is to encode knowledge in our DSL. Table 3 introduces a subset of the main action schemes included in our DSL, where the preconditions or effects are implemented by external functions, either to check the applicability of certain actions or facilitate node transformations. We encode a mix of low-level and high-level actions, where some high-level actions encode complex transformations that would otherwise require several low-level actions. This enables the solver to reason at the appropriate level of abstraction and lower the program complexity when possible. E.g., *SwapColor* and *CopyColor* can be realized by the ground action *UpdateColor* with additional program logic to manipulate pointers, but this would require increasing the number of program lines encoding a solution.

Each abstraction is associated with its respective set of actions and predicates and a full description is available in the supplementary materials. We also consider two additional abstractions to enable complicated movement, extension, and congruent node operations, where both node definitions are the same as the 4-connected abstraction. These are only tried if no solution can be found in the simpler abstractions.

		Predicates (? Parameters)
Attributes	test_p	*color-most(color), color-least(color),* *color-max(node), color-min(node), size-min(node),* *size-max(node), odd(node), even(node),* *up-border(node), down-border(node),* *left-border(node), right-border(node),* *left-diagonal(node), right-diagonal(node),* *horizontal-middle(node), vertical-middle(node),* *node-color(node, color), node-shape(node, shape),* *node-size(node, size), denoising-color(node, color).*
	-	*background(color).*
Relations	test_p	*node-diagonal(node, node), same-color(node, node),* *congruent(node, node), contain-node(node, node),* *partially-contain-node(node, node),* *relative-position(node, node, m-direction).*
	-	*node-spatial(node, node, m-direction).*
Pixel	-	*pixel-color(pixel, color), contain-pixel(node, pixel).*

Table 2: Predicates in our DSL. `test_p` indicates following predicates can be interpreted by the `test` action; the symbol "- "denotes predicates can not be interpreted.

Action Pruning Abstractions can introduce irrelevant actions in a domain. E.g., for the first task in Figure 1, actions that involve changing node positions should not be included, and in the second task, actions related to color updates should be avoided. A similar idea is discussed by Xu, Khalil, and Sanner (2023), where a newly generated node will be pruned while searching if it fails to satisfy a set of constraints generated by comparing the nodes defined by each abstraction. GPAR supports all their constraints. However, we acquire and use action constraints to prune irrelevant action schemes when generating the domain file instead of pruning generated nodes.

We consider mainly three constraints based on whether all nodes' positions, colors, or sizes remain unchanged across training input and output images. If any of the properties above hold true in the training sets, then the related actions involving movement, color, or size updating will be pruned. Some additional constraints are included to prune actions not applicable to a given abstraction; e.g., *InsertNode* is avoided when no consistent pattern (nodes with the same color, size, and shape) exists among input images. See the supplementary materials for the full list of actions associated with each constraint.

Program Synthesis

We use and improve the PGP(v) solver (Lei, Lipovetzky, and Ehinger 2023) to search in the space of planning programs over training instances of each ARC task. Once the solver returns a program that solves all training instances, we use the test instances to evaluate the solution. The solver core engine is a heuristic search algorithm that starts with an empty program and tries to program an instruction one line at a time until a solution is found.

Action Schemes (? Parameters)	Effects
UpdateColor(*node color₁ color₂*)	Change the node color from $color_1$ to $color_2$.
SwapColor(*node₁ node₂*)	Swap colors of $node_1$ and $node_2$.
CopyColor(*node₁ node₂*)	Copy the color of $node_1$ to $node_2$.
MoveNode(*node₁ node₂*)	Move $node_1$ to the boundary of $node_2$.
MoveNodeDirection(*node m-direction step*)	Move node with the given direction and step.
ExtendNode(*node₁ node₂*)	Extend $node_1$ until it hits the $node_2$.
ExtendNodeDirection(*node m-direction*)	Extend node in a given direction.

Table 3: Example action schemes designed in our DSL with external functions. Whole action descriptions, including preconditions and effects, are available in the supplementary materials.

Predicate and Argument Constraints Predicate constraints limit the allowed arguments of the test action. This action returns the interpretation of a predicate in a program, subsequently used to condition a goto instruction. Predicate constraints are determined before the search starts to ensure only relevant test actions are programmed. We restrict a predicate, describing a node attribute, which can be interpreted by the test action, iff there are two nodes, among all training and test input images, with a distinct value of that attribute. If all image nodes have the same attribute value described by a predicate, then the interpretation of that predicate will not be a helpful condition for a goto instruction, as the interpretation value is always true. For example in the third task of Figure 1, a valid condition should be the interpretation of the node color predicate rather than node size predicate since all nodes in the input images are of size 1.

Argument constraints make sure that if a node color or size predicate is used in a test action, then the arguments chosen describe attributes that exist in all training and test input images. These constraints prevent overfitting programs to work only on a subset of input instances, increasing the generalizability of the solution programs. For example, conditioning over nodes with size 3 in the second task of Figure 1 would not lead to a valid plan as the node size in the test instance is 2. In this case, other conditions should be used to create a solution moving down every node for one step.

Structural Restrictions Restrictions over the structure of programs are valid strategies to reduce symmetries in the search space (Lei, Lipovetzky, and Ehinger 2023). We adopt structural restrictions by separating a planning program into two sections: the *Application Section* and the *Looping Section*. The application section can be programmed with planning actions, test, and goto instructions, and the looping section has a sequence of pointer manipulations and goto instructions to ensure the iteration of all possible combinations of pointer values, followed by an end instruction for termination.
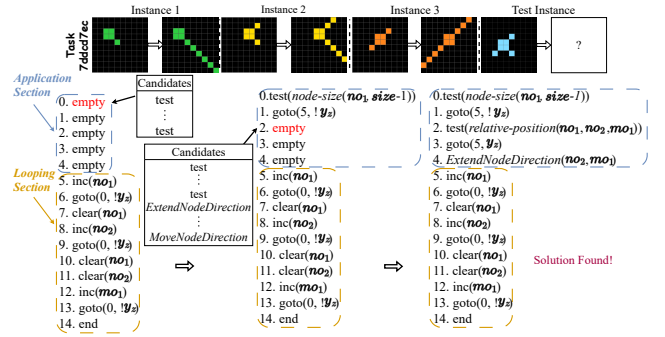
We program the looping section before the search starts

Figure 5: An illustration of the planning process with the application section and the looping section. Lines 0 and 1 ensure no_1 indexes the square node, and lines 2 and 3 constrain the no_2 to point to the single-pixel node, while mo_1 indexes the correct spatial relation between no_1 and no_2.

based on the given pointers. In the application section, the instruction sequence is constrained with the following rules: 1) a test action must be followed by a goto instruction; 2) the first line can only be programmed with a test action; 3) once an action from the DSL is programmed, the subsequent lines must either be programmed with actions from the DSL or followed by the looping section. To address a scenario where conditions are unnecessary, we include a dummy test(true) action whose interpretation is always true. In GPAR, the number of program lines n refers to the number of lines in the application section rather than in the full program.

Figure 5 illustrates the planning process of a planning program with a complex logic: the planning action *ExtendNodeDirection* is executed only when the first tested attribute is false (line 0) and the second tested spatial relation is true (line 2), using three pointers in total. These restrictions come with the caveats of making the solver incomplete. Even if no restrictions are used, existing approaches for ARC are already incomplete, as the expressivity of their DSLs limits the type of solution that can be found.

Heuristic Function Benefiting from the pixel-related predicates, we exploit the pixel information to guide the search. Every time a new program is generated, we execute it, and introduce a heuristic function h_p that goes beyond the goal-count heuristic by counting the number of pixels that differ from the goal state and penalizing further pixels whose values have been changed from the initial state and do not match yet the values in the goal state. This is very similar to the idea in means-ends analysis (Newell and Simon 1963), preferring programs that bring the current state abstraction closer to the goal state. We use h_p to guide the search algorithm used by the solver, and break ties with h_{ln} heuristic, which promotes the application of action schemes defined in the DSL. Full details of the solver and h_{ln} can be found in Lei, Lipovetzky, and Ehinger (2023).

Instantiation over Pointers GPAR supports partial instantiation over pointers, where a subset of parameters in a predicate or action schema are substituted by pointers and others are substituted by objects, such as the planning ac-
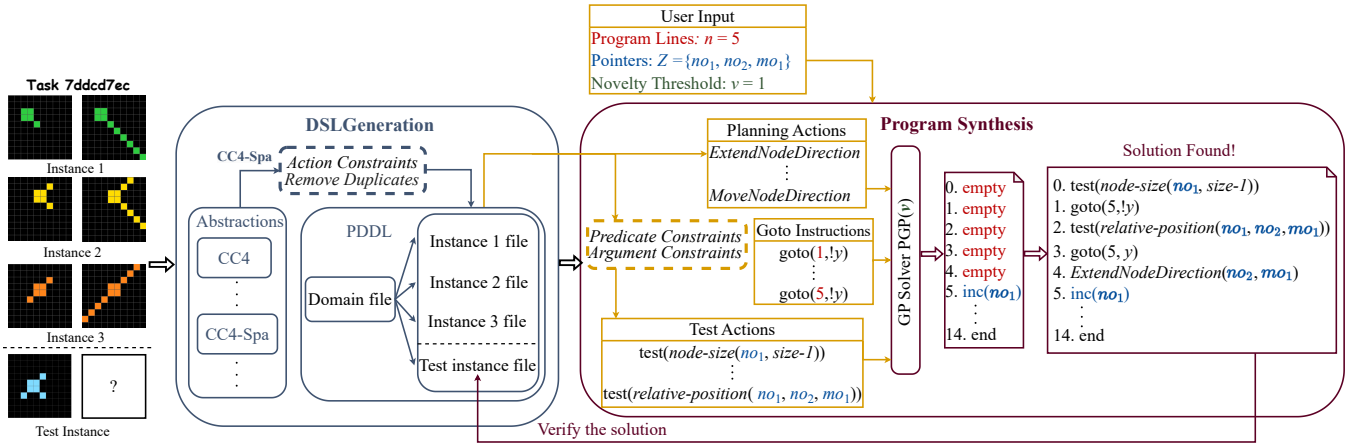
Figure 6: A pipeline sketch of GPAR. CC4 stands for the 4-connected abstraction; CC4-Spa stands for the abstraction that contains complicated movement and extension operations.

Pointer $\mapsto$ Object Type
$no_1 \mapsto$ NODE
$no_1 \mapsto$ NODE, $no_2 \mapsto$ NODE
$no_1 \mapsto$ NODE, $co_1 \mapsto$ COLOR
$no_1 \mapsto$ NODE, $no_2 \mapsto$ NODE, $co_1 \mapsto$ COLOR
$no_1 \mapsto$ NODE, $co_1 \mapsto$ COLOR, $co_2 \mapsto$ COLOR
$no_1 \mapsto$ NODE, $no_2 \mapsto$ NODE, $mo_1 \mapsto$ M-DIRECTION
$no_1 \mapsto$ NODE, $no_2 \mapsto$ NODE, $no_3 \mapsto$ NODE

Table 4: Pointer combinations in GPAR.

tion shown in Figure 3. This occurs when the number of pointers used to index an object type is less than the number of parameters specified by that object type. Partial instantiation allows `test` actions to fix a specific attribute for looping and branching, and naturally supports parameter bindings (Xu, Khalil, and Sanner 2023) without additional grammar extensions in our DSL.

System Overview

Figure 6 illustrates the pipeline sketch of GPAR, a two-stage system that employs GP to solve ARC tasks. The DSL generation stage encompasses a collection of abstractions with distinct node object, attribute and relation identifications to generate a domain file and associated instance files for each ARC task, where action constraints and duplication removal ensure that only helpful action schemes are included in the domain file, and unique abstractions are utilized. In the program synthesis stage, ground planning actions and `test` actions are generated by instantiating action schemes and predicates, described in the domain file, over objects declared in the instance file or pointers given by users, and `goto` instructions are generated based on the given program lines. The predicate and argument constraints increase the likelihood that generated `test` actions are useful and goal-oriented. PGP(v), the GP solver, leverages the user input, program lines n, pointers Z, and novelty threshold v that limits the number of occurrences of an action in a program, as parameters to implement the application section and looping section programming. The solution of PGP(v) is a planning program Π that can map the input image, the initial state, to the output image, the goal state, by executing Π on

the corresponding initial state in each training instance. Π is a verified solution if Π has been validated as a solution in the test instances.

Experiments

As a benchmark, we use the subset of 160 object-centric ARC tasks introduced by Xu, Khalil, and Sanner (2023). These tasks are further categorized into: 1) *recoloring* tasks which involve changing object colors; 2) *movement* tasks which involve changing object positions; 3) *augmentation* tasks which involve changing aspects of objects like size or pattern. Figure 1 shows example tasks from each class.

Parameters

In GPAR, PGP(v) takes n, v, and Z as parameters. The number of program lines n ranges from 3 to 10 where the valid Π configuration for $n = 3$ is $v = 1$ since each instruction included in Π with $n = 3$ can only appear once, such as a `test` action, a `goto` instruction and a planning action. For $n = 4$, reasonable configurations include $v = 1$ and $v = 2$ since a planning action can appear twice. For $n > 4$, the value of v ranges from 1 to 3. All the possible combinations of Z are presented in Table 4, where only object types NODE, COLOR, and M-DIRECTION are referenced since they are typical specifications of parameters in the design action schemes. The complexity of the search space is proportional to the values of n and v. The upper-bound values of n and v ensure the search space is large enough to cover most solutions while still being tractable.

The combination of feasible parameters and a valid DSL is supplied as the input for PGP(v). For each ARC task, possible combinations are executed in order of increasing complexity, starting from lower values of n and v, fewer pointers, and simpler abstractions (e.g., 4-connected are considered before 8-connected abstractions) with a time limit of 1800s for each. We treat the first encountered Π as the solution to generate the test output images for validation. Our approach keeps the search space tractable and ensures we find the simplest solution.

The Kaggle Challenge's first-place model and ARGA

Model	Task Type	Training Accuracy		Testing Accuracy	
	movement	18/31	(58.06%)	17/31	(54.84%)
ARGA	recolor	25/62	(40.32%)	23/62	(37.10%)
	augmentation	20/67	(29.85%)	17/67	(25.37%)
	all	63/160	(39.38%)	57/160	(35.62%)
	movement	**21**/31	(**67.74%**)	15/31	(48.39%)
Kaggle	recolor	23/62	(37.10%)	28/62	(45.16%)
First Place	augmentation	**35**/67	(**52.24%**)	21/67	(31.34%)
	all	79/160	(49.38%)	64/160	(40.00%)
	movement	20/31	(64.52%)	**19**/31	(**61.30%**)
GPAR	recolor	**41**/62	(**66.13%**)	**39**/62	(**62.90%**)
	augmentation	25/67	(37.31%)	**23**/67	(**34.33%**)
	all	**86**/160	(**53.75%**)	**81**/160	(**50.63%**)

Table 5: Performance of ARGA, Kaggle First Place and GPAR over 160 object-centric ARC tasks. Training accuracy is the number of tasks where the solution solves all the training instances. Testing accuracy is the number of tasks where the solution also generates the correct output images for all test instances. Best results are in bold.

are used as state-of-the-art baselines. All experiments were conducted on a cloud computer with clock speeds of 2.00 GHz Xeon processors. For the Kaggle first-place model and ARGA, the models were executed with a time limit of 1800s per task, and the highest-scored candidate generated by the model is selected as the final solution.
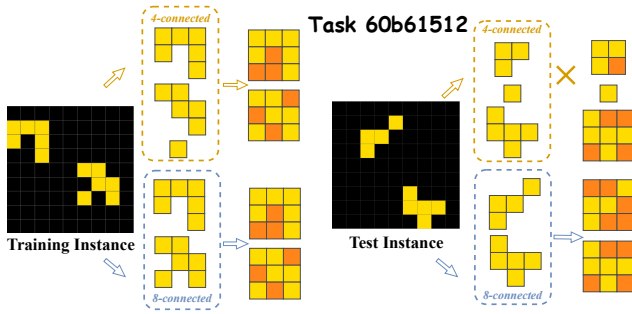

Figure 7: An example task where GPAR generated solution succeeds in the training instance but fails in the test instance.

Synthesis of Solutions

Table 5 shows the training and testing accuracy of GPAR, ARGA and Kaggle's winner. We score a model as "correct" in training if it is able to find a solution that solves all training instances for a given task, and we score that model "correct" in testing if its solution also gives the ground truth correct outputs on the test instances. In training, GPAR outperforms the other models in the *recolor* class, and outperforms all other approaches over the test instances. GPAR is the only planner that solves more than half of the tasks, 53.75% in training and 50.63% in testing overall. GPAR also shows the great generalization ability (the smallest gap between the training and testing accuracy).

GPAR has a distinct advantage in the *recolor* class, where solutions are compactly implemented by imperative programs with conditions mainly relying on predicates describing attributes, such as size, shape and color. For the *movement* class, the description of spatial relations remains challenging when dynamic attributes between nodes are needed, such as the center, corner, and area. Meanwhile, some tasks require movement actions defined with large numeric parameters, which is currently not supported well in our DSL. The *augmentation* class involves shape transformations, including rescaling, completion, and analogical replication, which are difficult to implement in imperative programs based on DSLs. All existing planners struggle with this category, with both training and testing accuracy below 50%.

Like previous models, GPAR shows some gap between training and testing, which means that a solution that solves the training set does not generalize to produce the correct results on test instances. Figure 7 shows an example where GPAR fails to generalize because both 4-connected and 8-connected abstractions can solve the training instance; however, only the 8-connected abstraction gives the correct solution for the test instance. The correct solution to this task is ambiguous given the training instances.

Of the tasks that GPAR solved in testing, over 50% require only a novelty threshold of 1 ($v = 1$) and just three program lines ($n = 3$). The low novelty threshold implies that most of the tasks can be solved without repeated actions, and the low number of program lines indicates that only a few conditions and/or actions are necessary to produce a solution (44/81 tasks require only one condition). This shows the efficiency of the DSL and the PGP(v) solver used by GPAR, which also contributes to its high generalization performance.

Conclusion

We leverage an existing solver for generalized planning to synthesize programs with pointers that represent expressive solutions with branches and loops for ARC tasks. We show how the *de facto* language for planning can be used to model object-aware abstractions, resulting in the state-of-the-art performance on the ARC, with greater generalization results. Identifying the most useful abstractions is still an open problem. In the future, new heuristics can be defined to guide the search of programs through relaxations from the DSL representation, and connections with alternative planning computational models can be explored to improve visual reasoning performance.

Acknowledgements

References

Acquaviva, S.; Pu, Y.; Kryven, M.; Sechopoulos, T.; Wong, C.; Ecanow, G.; Nye, M.; Tessler, M.; and Tenenbaum, J. 2022. Communicating Natural Programs to Humans and Machines. In *Proceedings of the 36th Advances in Neural Information Processing Systems*, NeurIPS, 3731–3743.

Barrett, D.; Hill, F.; Santoro, A.; Morcos, A.; and Lillicrap, T. 2018. Measuring Abstract Reasoning in Neural Networks. In *Proceedings of the 37th International conference on machine learning*, ICML, 511–520.

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

Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; and Nebel, B. 2009. Semantic Attachments for Domain-Independent Planning Systems. In *Proceedings of the 19th International conference on machine learning*, ICAPS, 114–121.

Frances, G.; Ramírez Jávega, M.; Lipovetzky, N.; and Geffner, H. 2017. Purely Declarative Action Descriptions are Overrated: Classical Planning with Simulators. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI, 4294–301.

Genesereth, M. R.; and Nilsson, N. J. 1987. CHAPTER 6 - Nonmonotonic Reasoning. In *Logical Foundations of Artificial Intelligence*, 115–159. Morgan Kaufmann.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. An Introduction to the Planning Domain Definition Language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2): 1–187.

Hu, Y.; and De Giacomo, G. 2011. Generalized Planning: Synthesizing Plans that Work for Multiple Environments. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, IJCAI, 918–923.

Jiménez, S.; Segovia-Aguas, J.; and Jonsson, A. 2019. A Review of Generalized Planning. *The Knowledge Engineering Review*, 34: e5.

Johnson, A.; Vong, W. K.; Lake, B. M.; and Gureckis, T. M. 2021. Fast and Flexible: Human Program Induction in Abstract Reasoning Tasks. *arXiv preprint arXiv:2103.05823*.

Lei, C.; Lipovetzky, N.; and Ehinger, K. A. 2023. Novelty and Lifted Helpful Actions in Generalized Planning. In *Proceedings of the 16th International Symposium on Combinatorial Search*, SoCS, 148–152.

Levesque, H. J. 1986. Knowledge Representation and Reasoning. *Annual Review of Computer Science*, 1(1): 255–287.

Małkiński, M.; and Mańdziuk, J. 2023. A Review of Emerging Research Directions in Abstract Visual Reasoning. *Information Fusion*, 91: 713–736.

Moskvichev, A.; Odouard, V. V.; and Mitchell, M. 2023. The ConceptARC Benchmark: Evaluating Understanding and Generalization in the ARC Domain. *arXiv preprint arXiv:2305.07141*.

Newell, A.; and Simon, H. A. 1963. GPS, A Program that Simulates Human Thought. In *Computers and Thought*, 279–293. McGraw-Hill.

Segovia-Aguas, J.; Celorrio, S. J.; Sebastiá, L.; and Jonsson, A. 2022. Scaling-up Generalized Planning as Heuristic Search with Landmarks. In *Proceedings of the 15th International Symposium on Combinatorial Search*, SoCS, 171–179.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2019. Computing Programs for Generalized Planning Using a Classical Planner. *Artificial Intelligence*, 272: 52–85.

Spelke, E. S.; and Kinzler, K. D. 2006. Core Knowledge. *Developmental Science*, 10(1): 89–96.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning Generalized Plans Using Abstract Counting. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, AAAI, 991–997.

top quarks. 2020. ARC-solution. https://github.com/top-quarks/ARC-solution. Accessed: 2023-06-01.

Xu, Y.; Khalil, E. B.; and Sanner, S. 2023. Graphs, Constraints, and Search for the Abstraction and Reasoning Corpus. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence*, AAAI, 4115–4122.