From Reasoning to Generalization: Knowledge-Augmented LLMs for the ARC Benchmark

Anonymous ACL submission

Abstract

Despite extensive research on reasoning-oriented LLMs, core cognitive faculties of human intelligence, such as abstract reasoning and generalization, remain underexplored. To address this, we evaluate recent reasoning-oriented LLMs on the Abstraction and Reasoning Corpus (ARC) benchmark, which explicitly demands both faculties. We formulate ARC as a program synthesis task and propose nine candidate solvers. Experimental results show that repeated-sampling planning-aided code generation (RSPC) achieves the highest test accuracy and demonstrates consistent generalization across most LLMs. To further improve performance, we introduce Knowledge Augmentation for Abstract Reasoning (KAAR), which encodes *core knowledge* priors within an ontology that classifies priors into three hierarchical levels based on their dependencies. KAAR progressively expands LLM reasoning capacity by gradually augmenting priors at each level, and invokes RSPC to generate candidate solutions after each augmentation stage. Empirical results show that KAAR maintains strong generalization and consistently outperforms non-augmented RSPC across all evaluated LLMs, achieving around 5% absolute gains and up to 64.52% relative improvement.

1 Introduction

Abstract reasoning and generalization are fundamental to understanding and evaluating machine intelligence (Małkiński and Mańdziuk, 2023; Barrett et al., 2018; Moskvichev et al., 2023). Chollet (2019) introduced Abstraction and Reasoning Corpus (ARC) to assess these capabilities of AI systems. In each ARC task, the solver is required to infer generalized rules or procedures from a small set of training instances, typically fewer than five input-output image pairs, and apply them to generate output images for given input images in test instances (Figure 1 (a)). Each image in ARC is a pixel grid represented as a 2D matrix, where each value denotes a pixel color (Figure 1 (b)). ARC evaluates *broad generalization*, encompassing reasoning over individual input-output pairs and inferring generalized solutions via high-level abstraction, akin to inductive reasoning (Peirce, 1868).

ARC is grounded in *core knowledge* priors that serve as foundational cognitive faculties of human intelligence (Spelke and Kinzler, 2007). We note that all ARC solvers are assumed to possess these priors, enabling equitable comparisons between AI systems and human cognitive abilities (Chollet, 2019). An abstract and expressive representation of core knowledge priors is fundamental in ARC to enable human-like reasoning (Chollet, 2019). Specifically, these priors include: (1) *objectness* – aggregating elements into coherent, persistent objects; (2) *geometry and topology* – recognizing and manipulating shapes, symmetries, spatial transformations, and structural patterns (e.g., containment, repetition, projection); (3) *numbers and counting* – counting, sorting, comparing quantities, performing basic arithmetic, and identifying numerical patterns; and (4) *goal-directedness* – inferring purposeful transformations between initial and final states without explicit temporal cues.

Chollet (2019) suggested approaching ARC tasks as instances of program synthesis. Following this proposal, previous solvers, Abstract Reasoning with Graph Abstractions (ARGA) (Xu et al., 2023a) and Generalized Planning for Abstract Reasoning (GPAR) (Lei et al., 2024b), a state-of-the-art object-centric solver, have successfully solved subsets of ARC tasks by searching for program solutions within object-centric domain-specific languages (DSLs). GPAR and KAAR achieve notable generalization benefits from their lifted core knowledge priors representations in DSLs. However, the expansive search space limits their scalability. Recently, reasoning-oriented LLMs trained via reinforcement learning to integrate chain-of-thought

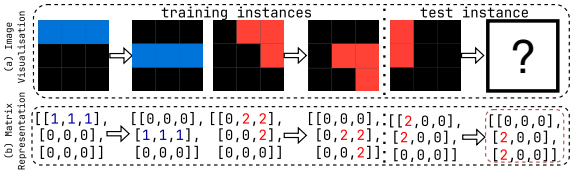


Figure 1: Example ARC problem (*25ff71a9*) with image visualizations (a), including two input-output pairs in the training instances and one input image in the test instance, along with their 2D matrix representations (b). The ground-truth test output is enclosed in a red box.

reasoning (Wei et al., 2022) further advance program synthesis performance. Common approaches using LLMs for code generation include repeated sampling, where multiple candidate programs are generated (Chen et al., 2021), followed by best-program selection strategies (Li et al., 2022; Chen et al., 2023; Zhang et al., 2023; Ni et al., 2023), and code refinement, where initial LLM-generated code is iteratively improved using error feedback from execution results (Zhong et al., 2024; Lei et al., 2024a) or LLM-generated explanations (Zhong et al., 2024; Chen et al., 2024; Lei et al., 2024a). We note that ARC presents greater challenges than existing program synthesis benchmarks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and LiveCode (Jain et al., 2025), due to its emphasis on generalization and abstract reasoning grounded in core knowledge priors, which remain underexplored. This gap motivates our evaluation of reasoning-oriented LLMs on the ARC benchmark and our proposed knowledge augmentation approach to improve their performance. See Appendix A for detailed related work.

We begin by systematically assessing how reasoning-oriented LLMs approach ARC tasks within the program synthesis framework, given solely 2D matrices as input. In total, nine ARC solvers are considered with accuracy on test instances reported as the primary metric. When evaluated on the ARC public evaluation set (400 problems), repeated-sampling planning-aided code generation (RSPC) demonstrates consistent generalization and achieves the highest test accuracy across most evaluated LLMs. We treat the most competitive ARC solver, RSPC, as the solver backbone. We further propose Knowledge Augmentation for Abstract Reasoning (KAAR) for solving ARC tasks. KAAR refines the core knowledge priors encoded in GPAR and formalizes them through a lightweight ontology that organizes priors into hierarchical levels based on their dependencies. It incrementally augments LLMs with priors at each level through in-context learning. After completing augmentation at each level, KAAR applies the ARC solver backbone (RSPC) to generate the solution. This progressive augmentation enables LLMs to gradually expand their reasoning capabilities and facilitates stage-wise reasoning, aligning with human cognitive development (Babakr et al., 2019). Empirical results show that KAAR consistently outperforms RSPC in test accuracy across all evaluated LLMs, while preserving strong generalization. We outline our contributions as follows:

- We evaluate the abstract reasoning and generalization capabilities of reasoning-oriented LLMs on ARC using nine program synthesis solvers.
- We introduce KAAR, a knowledge augmentation approach for solving ARC problems with LLMs, further improving performance.
- We conduct a comprehensive performance analysis of the evaluated LLMs and proposed ARC solvers, highlighting failure cases and remaining challenges in ARC.

2 Problem Formulation

We formulate each ARC task as a tuple $\mathcal{P} = \langle I_r, I_t \rangle$, where I_r and I_t are sets of training and test instances. Each instance is an input-output image pair (i^i, i^o), represented as 2D matrices. The goal is to leverage the LLM $\mathcal{M}$ to generate a solution s based on training instances I_r and test input images $\{i^i \mid (i^i, i^o) \in I_t\}$, where s maps each test input i^i to its output i^o, i.e., $s(i^i) = i^o$, for $(i^i, i^o) \in I_t$. We note that test input images are visible during the generation of solution s, whereas test output images become accessible only after s is produced for validation. We encode the solution s in different forms, as a text-based plan p, or as Python code c, optionally guided by p. We denote each ARC problem description, comprising I_r and $\{i^i \mid (i^i, i^o) \in I_t\}$, as Q.

3 ARC Solver Backbone

LLMs have shown promise in solving tasks that rely on ARC-relevant priors (Deng et al., 2024; Meng et al., 2024; Ahn et al., 2024; Zang et al., 2025), suggesting an implicit encoding of core knowledge priors for solving ARC tasks. Unlike previous work that treats LLMs as output image generators (Min, 2023; Xu et al., 2023b; Lee et al., 2024), we cast each ARC task as a program synthesis problem, which involves generating a solution

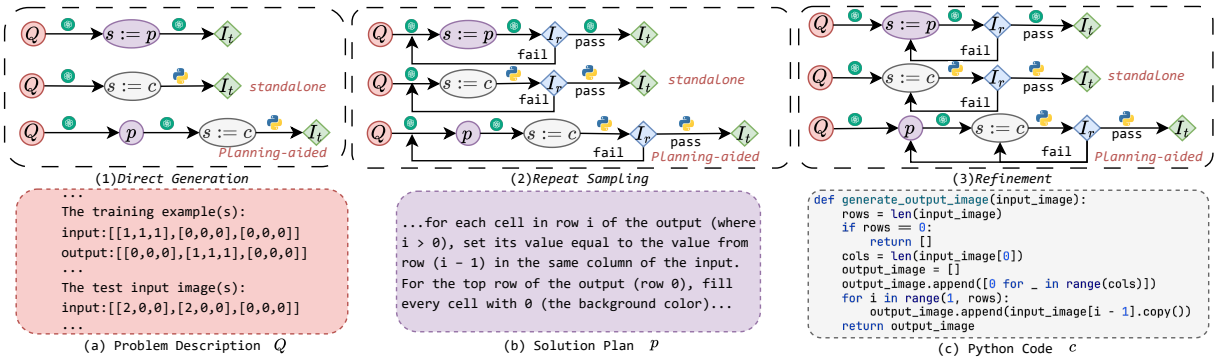


Figure 2: ARC solution generation approaches, (1) *direct generation*, (2) *repeated sampling*, and (3) *refinement*, with the GPT-o3-mini input and response fragments (a–c) for solving task *25ff71a9* (Figure 1). For each approach, when the solution s is code, $s := c$, a plan p is either generated from the problem description Q to guide code generation (*planning-aided*) or omitted (*standalone*). Otherwise, when $s := p$, the plan p serves as the final solution.

s from a problem description Q, where knowledge priors emerge from LLMs' perception of grid dynamics. We consider established LLM-based code generation approaches (Zhong et al., 2024; Lei et al., 2024a; Chen et al., 2024; Islam et al., 2024) as candidate ARC solution generation strategies, illustrated at the top of Figure 2. These include: (1) *direct generation*, where LLM produces the solution s in a single attempt, and then validates it on test instances I_t; (2) *repeated sampling*, where LLM samples solutions until one passes training instances I_r, and then evaluates it on I_t; and (3) *refinement*, where LLM iteratively refines an initial solution s based on failures on I_r until it succeeds, followed by evaluation on I_t. In addition, we extend the solution representation beyond code to include text-based solution plans. Given the problem description Q as input (Figure 2, block (a)), all strategies prompt the LLM to generate a solution s, represented either as a natural language plan p (block (b)), $s := p$, or as a Python code c (block (c)), $s := c$. For $s := p$, the solution is derived directly from Q. For $s := c$, we explore two modalities: the LLM either generates c directly from Q (*standalone*), or first generates a plan p for Q, which is then concatenated with Q to guide subsequent code development (*planning-aided*), a strategy widely adopted in recent work (Lei et al., 2024a; Jiang et al., 2023; Islam et al., 2024).

Repeated sampling and refinement iteratively produce new solutions based on the correctness of s on training instances I_r, and validate s on test instances I_t once it passes I_r or the iteration limit is reached. When $s := p$, its correctness is evaluated by querying the LLM to generate each output image i^o given its corresponding input i^i and the solution plan p, where $(i^i, i^o) \in I_r$ or $(i^i, i^o) \in I_t$. Alternatively, when $s := c$, its correctness is assessed by executing c on I_r or I_t. In repeated sampling, the LLM iteratively generates a new plan p and code c from the problem description Q without additional feedback. In contrast, refinement revises p and c by presenting the LLM with the previously incorrect p and c, concatenated with failed training instances. In total, nine ARC solvers are employed to evaluate the performance of reasoning-oriented LLMs on the ARC benchmark, and the most competitive solver is served as the ARC solver backbone.

4 Knowledge Augmentation

Prior Knowledge Encoding. The core knowledge priors specified in ARC can be formulated as a finite set of lifted, structured primitives (predicates and operators), enabling ARGA and GPAR to manually encode and instantiate them to ensure prior satisfaction. GPAR models predicates as abstraction-defined nodes enriched with attributes and inter-node relations, which are extracted using standard image processing algorithms. Building on this insight, we propose KAAR, a knowledge augmentation approach for solving ARC tasks using reasoning-oriented LLMs. Following the core knowledge priors formulated in GPAR, KAAR expands their scope, revises their implementations, and categorizes these priors into four knowledge dimensions in ARC. In detail, KAAR adopts fundamental abstraction methods from GPAR to enable objectness. Objects are typically defined as components based on adjacency rules and color consistency (e.g., 4-connected or 8-connected components), while also including the entire image as a component. KAAR further introduces four novel abstractions: (1) *middle-vertical*, which vertically

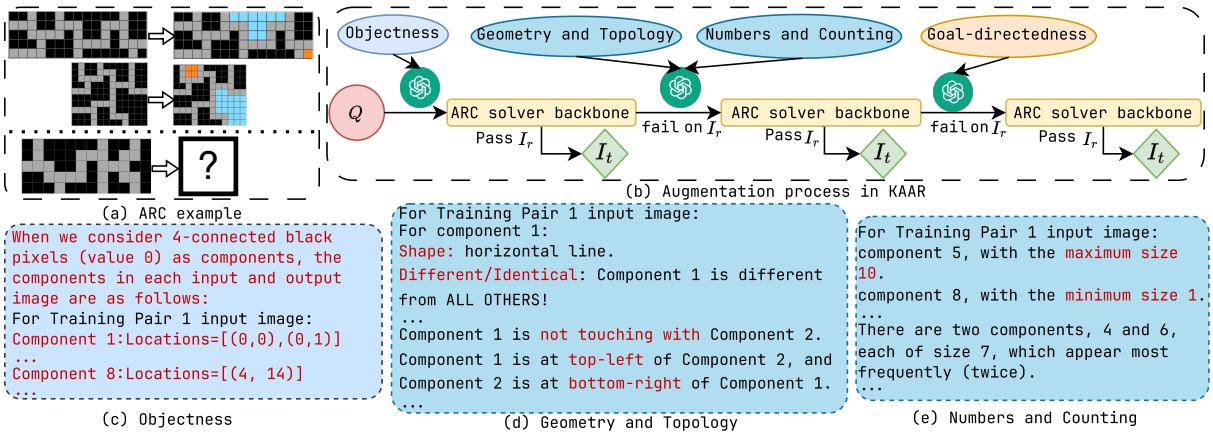


Figure 3: Augmentation process in KAAR (block (b)) and the corresponding knowledge augmentation fragments (blocks (c-e)) for ARC problem *62ab2642* (block (a)).

splits the image into two equal parts, and treats each as a distinct component; (2) *middle-horizontal*, which applies the same principle along the horizontal axis; (3) *multi-lines*, which segments the image using full-length rows or columns of uniform color, and treats each resulting part as a distinct component; and (4) *no abstraction*, which considers only raw 2D matrices. Under *no abstraction*, KAAR degrades to the ARC solver backbone without incorporating any priors. KAAR incorporates GPAR's component attributes (size, color, shape) and relations (spatial, congruent, inclusive), treating them as geometric and topological priors, respectively. It further extends with four novel attributes, including symmetry, bounding box, nearest boundary, and hole count, as well as a new touching relation. For numeric and counting priors, KAAR adopts GPAR's largest/smallest component sizes, and the most/least frequent component colors, while extending them with novel statistical analysis of hole counts, symmetry and color diversity, as well as the most/least frequent sizes and shapes.

GPAR approaches goal-directedness priors by searching for a sequence of program instructions (Lei et al., 2023) defined in a DSL. Each instruction supports conditionals, branching, looping, and action statements. Inspired by these condition and action concepts, KAAR novelly enables goal-directedness priors by augmenting LLM knowledge in two steps: 1) It prompts the LLM to identify the most relevant actions for solving the given ARC problem from ten predefined action categories (Figure 4 block (a)), partially derived from GPAR and extended based on the training set, such as color change, movement, and extension; 2) For each selected action, KAAR instructs the LLM with the associated schema to resolve implementation details. For example, for a color change action, KAAR first queries the LLM to identify the target components (Figure 4 blocks (b)), and then specify the source and target colors for modification based on the target components (Figure 4 blocks (c)). We note that KAAR also instructs the LLM to incorporate condition-aware reasoning when determining action implementation details, using knowledge derived from geometry, topology, numbers, and counting priors. This enables fine-grained control, for example, applying color changes only to black components conditioned on the maximum or minimum size: from black (value 0) to blue (value 8) if largest, or to orange (value 7) if smallest. Figure 4 shows fragments of the goal-directedness prior augmentation. See Appendix B for whole priors in KAAR.

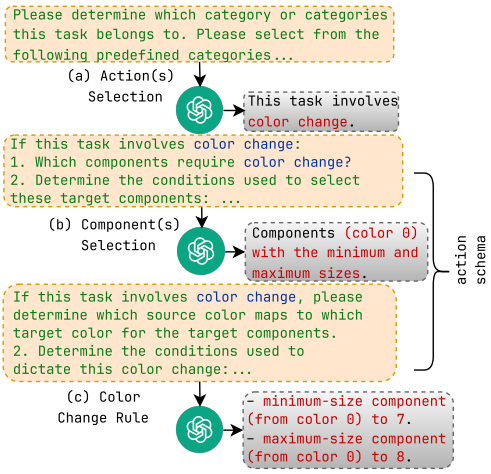


Figure 4: Goal-directedness priors augmentation with input and response fragments from GPT-o3-mini.

Ontology Construction. In contrast to GPAR and previous works that leave priors unorganized (Xu et al., 2023a; Tan and Motani, 2024), KAAR

structures the full set of core knowledge priors into an ontology, where priors are organized into three hierarchical levels based on their dependencies. KAAR augments LLMs with priors at each level to enable incremental augmentation via in-context learning. This reduces context interference and supports stage-wise reasoning aligned with human cognitive development (Babakr et al., 2019). Figure 3, block (b), illustrates the augmentation process in KAAR alongside the augmented prior fragments used to solve the problem shown in block (a). KAAR begins augmentation with objectness priors, encoding images into components with detailed coordinates based on a specific abstraction method (block (c)). KAAR then prompts geometry and topology priors (block (d)), followed by numbers and counting priors (block (e)). These priors reside at the same ontological level, as they all build upon objectness. Finally, KAAR augments goal-directedness priors, as shown in Figure 4, where target components are derived from objectness analysis and conditions are inferred from geometric, topological, and numerical analyses. After augmenting each level of priors, KAAR invokes the ARC solver backbone to generate solutions. If any solution passes I_r, it is validated on I_t; otherwise, augmentation proceeds to the next level of priors.

Prior Restriction. While the ontology provides a hierarchical representation of priors, it may also introduce hallucinations, such as duplicate abstractions, irrelevant component attributes or relations, and inapplicable actions. To address this, KAAR integrates additional restrictions to filter out inapplicable priors. KAAR adopts the duplicate-checking strategy in GPAR, retaining only abstractions that yield distinct components by size, color, or shape, in at least one training instance. In KAAR, each abstraction is associated with a set of applicable priors. For instance, when the entire image is treated as a component, relation priors are excluded, and actions such as movement and color change are omitted, whereas symmetry and size attributes are retained and actions such as flipping and rotation are considered. In contrast, 4-connected and 8-connected abstractions include all component attributes and relations, and the full set of ten action priors. See Appendix C for detailed restrictions.

5 Experiments

Experimental Setup. We begin by comparing nine candidate solvers on the full ARC public evaluation set of 400 tasks. This offers broader insights than previous studies limited to the subsets of 400 training tasks (Lei et al., 2024b; Xu et al., 2023a; Wang et al., 2024), given the greater difficulty of the evaluation set (LeGris et al., 2024). We experiment with reasoning-oriented LLMs, including proprietary models, GPT-o3-mini (OpenAI, 2025), and Gemini-2.0-Flash-Thinking (Gemini-2.0) (DeepMind, 2024), and open-source models, DeepSeek-R1-Distill-Llama-70B (DeepSeek-R1-70B) (Guo et al., 2025) and QwQ-32B (Cloud, 2025). We compute accuracy on test instances I_t as the primary evaluation metric. It measures the proportion of problems where the first solution passes training instances I_r and successfully solves I_t; otherwise, if none pass I_r within 12 iterations, the last solution is evaluated on I_t, applied to both repeated sampling and refinement strategies. We also report accuracy on I_r and $I_r \& I_t$, measuring the percentage of problems whose solutions solve I_r and both I_r and I_t. See Appendix D for parameter settings.

Results on Program Synthesis. Table 1 reports the performance of nine ARC solvers across four reasoning-oriented LLMs. For direct generation methods, accuracy on I_r and $I_r \& I_t$ is omitted, as solutions are evaluated directly on I_t. GPT-o3-mini outperforms all other LLMs, achieving the highest accuracy on I_r (52.50%), I_t (32.50%), and $I_r \& I_t$ (31.75%) under repeated sampling with standalone code generation (*C*), highlighting its strong abstract reasoning capabilities. Notably, QwQ-32B, the smallest model, outperforms DeepSeek-R1-70B across all solvers and surpasses Gemini-2.0 under refinement. Among nine solvers, repeated sampling-based methods generally outperform those based on direct generation or refinement. This diverges from previous findings where refinement dominated conventional code generation tasks that lack abstract reasoning and generalization demands (Lei et al., 2024a; Zhong et al., 2024; Chen et al., 2024) (see Appendix I for detailed analysis). Within repeated sampling, planning-aided code generation (*PC*) yields the highest accuracy on I_t across most LLMs. It also demonstrates the strongest generalization with GPT-o3-mini and Gemini-2.0, as evidenced by the smallest accuracy gap between I_r and $I_r \& I_t$. A similar trend is observed on QwQ-32B and DeepSeek-R1-70B, where both *C* and *PC* generalize effectively across repeated sampling and refinement. Overall, repeated sampling with planning-aided code generation (RSPC) shows the best performance and thus

		Direct Generation			Repeated Sampling			Refinement		
		P	*C*	*PC*	*P*	*C*	*PC*	*P*	*C*	*PC*
GPT-o3-mini	I_r	-	-	-	35.50	**52.50**	35.50	31.00	47.25	32.00
	I_t	20.50	24.50	22.25	23.75	**32.50**	30.75	24.75	29.25	25.75
	$I_r\&I_t$	-	-	-	22.00	**31.75**	29.25	21.75	28.50	25.00
Gemini-2.0	I_r	-	-	-	36.50	**39.50**	21.50	15.50	25.50	15.50
	I_t	7.00	6.75	6.25	10.00	14.75	**16.75**	8.75	12.00	11.75
	$I_r\&I_t$	-	-	-	9.50	14.25	**16.50**	8.00	10.50	10.75
QwQ-32B	I_r	-	-	-	**19.25**	13.50	15.25	16.75	15.00	14.25
	I_t	9.50	7.25	5.75	11.25	13.50	**14.25**	11.00	**14.25**	14.00
	$I_r\&I_t$	-	-	-	9.25	12.75	**13.00**	8.75	**13.00**	11.75
DeepSeek-R1-70B	I_r	-	-	-	**8.75**	6.75	7.75	6.25	5.75	7.75
	I_t	4.25	4.75	4.50	4.25	7.25	**7.75**	4.75	5.75	**7.75**
	$I_r\&I_t$	-	-	-	3.50	6.50	**7.25**	4.25	5.25	7.00

Table 1: Performance of nine ARC solvers measured by accuracy on I_r, I_t, and $I_r\&I_t$ using four reasoning-oriented LLMs. For each LLM, the highest accuracy on I_r and $I_r\&I_t$ is in bold; the highest accuracy on I_t is in red. Accuracy is reported as a percentage. *P* denotes the solution plan; *C* and *PC* refer to standalone and planning-aided code generation, respectively.

		I_r			I_t			$I_r\&I_t$		
		Acc	Δ	γ	Acc	Δ	γ	Acc	Δ	γ
GPT-o3-mini	RSPC	35.50	-	-	30.75	-	-	29.25	-	-
	KAAR	**40.00**	4.50	12.68	**35.00**	4.25	13.82	**33.00**	3.75	12.82
Gemini-2.0	RSPC	21.50	-	-	16.75	-	-	16.50	-	-
	KAAR	25.75	4.25	19.77	21.75	5.00	29.85	20.50	4.00	24.24
QwQ-32B	RSPC	15.25	-	-	14.25	-	-	13.00	-	-
	KAAR	22.25	**7.00**	45.90	21.00	**6.75**	47.37	19.25	**6.25**	48.08
DeepSeek-R1-70B	RSPC	7.75	-	-	7.75	-	-	7.25	-	-
	KAAR	12.25	4.50	**58.06**	12.75	5.00	**64.52**	11.50	4.25	**58.62**

Table 2: Comparison of RSPC and KAAR in terms of accuracy (Acc) on I_r, I_t, and $I_r\&I_t$. Δ and γ denote the absolute and relative improvements over RSPC, respectively. All values are reported as percentages. For I_r, I_t, and $I_r\&I_t$, the highest Acc values are in red and the best Δ and γ results are in bold.

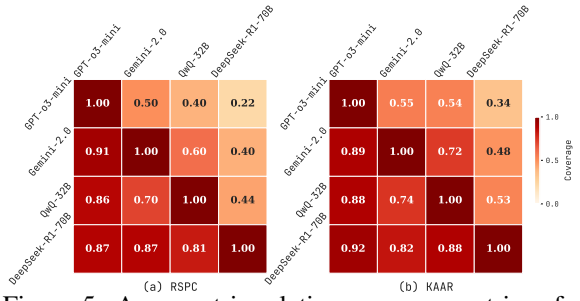


Figure 5: Asymmetric relative coverage matrices for RSPC (a) and KAAR (b), showing the proportion of problems whose test instances are solved by the row model that are also solved by the column model.

serves as the ARC solver backbone.

Results on KAAR. We further compare the performance of RSPC with its knowledge-augmented variant, KAAR. For each task, KAAR begins with simpler abstractions, i.e., no abstraction and whole image, and progresses to complicated 4-connected and 8-connected abstractions, consistent with GPAR. KAAR reports the accuracy on test instances I_t based on the first abstraction whose solution solves all training instances I_r; otherwise, it records the final solution from each abstraction and selects the one that passes the most I_r to evaluate on I_t (details in Appendix E). KAAR allows the solver backbone (RSPC) up to 4 iterations per invocation, totaling 12 iterations, consistent with the non-augmented setting. As shown in Table 2, KAAR consistently outperforms non-augmented RSPC, yielding around 5% absolute accuracy gains on I_r, I_t, and $I_r\&I_t$ across all LLMs. This highlights the effectiveness and model-agnostic nature of the augmented priors. KAAR achieves the highest accuracy using GPT-o3-mini, i.e., 40% on I_r, 35% on I_t, and 33% on $I_r\&I_t$. It shows the best absolute improvements (Δ), around 6.5%, using QwQ-32B and the most pronounced relative gains (γ), over 58%, using DeepSeek-R1-70B on I_r, I_t, and $I_r\&I_t$. Moreover, KAAR maintains generalization comparable to RSPC across all LLMs, indicating that the augmented priors are sufficiently abstract and expressive to serve as basis functions for reasoning, in line with ARC assumptions. See Appendix J for a discussion of generalization.

Performance Analysis. We compare relative problem coverage across evaluated LLMs under RSPC and KAAR based on successful solutions on test instances. As shown in Figure 5, each cell (i, j) is computed as $\frac{|A_i \cap A_j|}{|A_i|}$, where A_i and A_j are the sets of problems solved by the row and column LLMs, respectively. Values near 1 indicate that the column LLM covers most problems solved by the row LLM. Under RSPC (Figure 5 (a)), GPT-o3-mini exhibits broad coverage, with column values consistently above 0.85. Gemini-2.0 and QwQ-32B also show substantial alignment, with mutual coverage exceeding 0.6. Figure 5 (b) illustrates that KAAR generally improves or maintains inter-

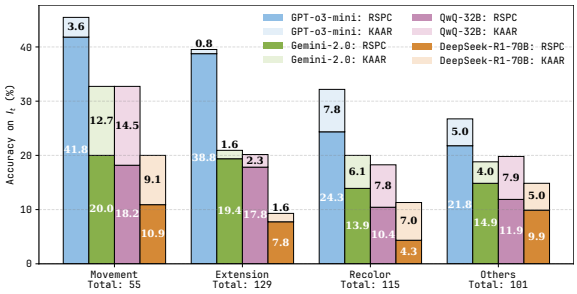


Figure 6: Accuracy on test instances I_t for RSPC and KAAR across the *movement*, *extension*, *recolor*, and *others* categories using four LLMs. Each stacked bar shows RSPC accuracy (darker segment) and the additional improvement from KAAR (lighter segment).

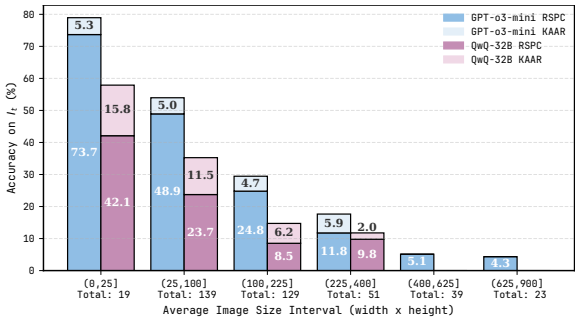


Figure 7: Accuracy on test instances I_t for RSPC and KAAR across average image size intervals, evaluated using GPT-o3-mini and QwQ-32B. See Figure 11 in Appendix for results on other LLMs.

model overlap compared to RSPC. Notably, the coverage between GPT-o3-mini and DeepSeek-R1-70B increases from 0.22 under RSPC to 0.34 under KAAR. These results underscore the effectiveness of KAAR in improving cross-model generalization, enabling all evaluated LLMs to solve additional shared problems and allowing smaller models such as QwQ-32B and DeepSeek-R1-70B to better align with stronger LLMs on ARC. See Appendix K for problem coverage across evaluated solvers.

Performance by Category. Following prior work (Xu et al., 2023a; Lei et al., 2024b), we categorize the 400 problems in the ARC public evaluation set into four classes based on their primary transformations: (1) *movement* (55 problems), (2) *extension* (129 problems), (3) *recolor* (115 problems), and (4) *others* (101 problems). The *others* category comprises infrequent transformations. See Appendix F for examples of each category. Figure 6 illustrates the accuracy on test instances I_t for RSPC and KAAR across four categories with evaluated LLMs. Each stacked bar represents RSPC accuracy and the additional improvement contributed by KAAR. KAAR consistently outperforms RSPC with the largest accuracy gain in *movement* (14.5% with QwQ-32B). It shows limited improvements in *extension*, where several problems involve pixel-level extension, which minimizes the reliance on component-level recognition. Moreover, *extension* requires accurate spatial inference across multiple components. This poses greater difficulty than *movement*, which requires mainly direction identification. We note that multi-component spatial inference remains a persistent challenge for LLMs (Yamada et al., 2024; Cohn and Hernandez-Orallo, 2023; Bang et al., 2023), highlighting a difficulty that is intrinsic to ARC and reinforcing its role as a challenging benchmark for abstract reasoning and generalization.

Impact of Size. A notable feature of ARC is the variation in image size both within and across problems. We categorize tasks by averaging the image size per problem, computed over both training and test image pairs. We report the accuracy on I_t for RSPC and KAAR across average image size intervals using GPT-o3-mini and QwQ-32B, the strongest proprietary and open-source models. As shown in Figure 7, both LLMs experience performance degradation as image size increases. When the average image size exceeds 400 (20×20), GPT-o3-mini solves only three problems, while QwQ-32B solves none. Benefiting from object-centric representations, KAAR consistently outperforms RSPC on problems with average image sizes below 400. By abstracting each image into components, KAAR reduces interference from irrelevant pixels, directs attention to salient components, and facilitates component-level transformation analysis. However, larger images tend to yield oversized and numerous components after abstraction, where oversized components impede transformation execution and numerous components complicate the identification of target components.

Scaling with Iterations. Figure 8 presents the variance in accuracy on $I_r \& I_t$ for RSPC and KAAR as iteration count increases using GPT-o3-mini and QwQ-32B. For each task under KAAR, we record iterations from the abstraction that solves both I_r and I_t. For KAAR, performance improvements across each 4-iteration block are driven by the solver backbone invocation after augmenting an additional level of priors. RSPC shows rapid improvement in the first 4 iterations and plateaus around iteration 8. KAAR consistently outperforms RSPC, with the performance gap progressively increasing after additional priors are aug-

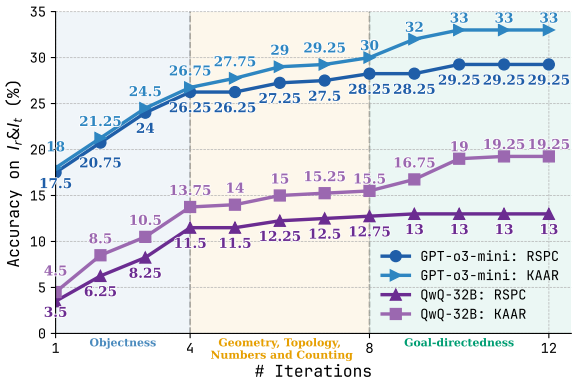


Figure 8: Accuracy variance on I_r&I_t over iterations for RSPC and KAAR using GPT-o3-mini and QwQ-32B. See Appendix Figure 12 for results on other LLMs.

mented and peaking after the integration of goal-directedness. KAAR benefits from newly introduced priors that extend the priors in GPAR and organizes these priors into a dependency-aware ontology to enable progressive augmentation and stage-wise reasoning, thereby improving solution accuracy. Compared with variants that remove the newly introduced priors or apply augmentation at once without stage-wise reasoning, KAAR yields superior accuracy, as detailed in Appendix G.

6 Discussion

Rationale for ARC. While related benchmarks exist, ARC remains a highly authentic measure of generalization. In contrast to RAVEN (Raven, 2000) and PGM (Barrett et al., 2018), which primarily evaluate closed-form rule fitting within predefined rule spaces, ARC assumes that tasks are solvable through core knowledge priors, while intentionally leaving the problems underspecified to prevent the explicit encoding of complete solution rules (Chollet, 2019). Each ARC task is unique, defining a distinct domain that requires solvers to infer solutions from a few examples and generalize to test cases, thereby advancing models beyond rule fitting toward truly generalization. Recent training approaches that employ data augmentation to generate additional examples for each task (Akyürek et al., 2024; Franzen et al., 2025) achieve notable results on ARC. However, they deviate substantially from ARC's objective of solving each task from limited examples without reliance on externally generated data (Chollet, 2019).

Cost Analysis. As the solver backbone, RSPC uses fewer tokens than its refinement-based counterpart while achieving comparable or higher accuracy. In KAAR, knowledge augmentation increases token consumption, while the additional tokens remain relatively constant since all priors, except goal-directedness, are generated via image processing algorithms. On GPT-o3-mini, augmentation tokens constitute around 60% of solver backbone token usage, while on QwQ-32B, this overhead decreases to about 20%, as the solver backbone consumes more tokens. See Appendix H for a detailed discussion.

Solution Analysis. To assess adherence to core knowledge priors, we manually reviewed the solution plans and code generated by RSPC and KAAR that successfully solve I_t with GPT-o3-mini. RSPC tends to solve problems without object-centric reasoning. For instance, in Figure 1, it shifts each row downward by one and pads the top with zeros, rather than reasoning over objectness to move each 4-connected component down by one step. We note that object recognition in ARC involves grouping pixels into task-specific components based on clustering rules, differing from feature extraction approaches (Zhao et al., 2019) in conventional computer vision tasks. It remains challenging for current LLMs (Xu et al., 2023b; Li et al., 2024a). In contrast, KAAR enables objectness through explicitly defined abstractions, implemented via standard image processing algorithms, thus ensuring both accuracy and generalization. KAAR-generated solutions are consistent with the augmented priors, as shown in Figure 4.

7 Conclusion

We explored the generalization and abstract reasoning capabilities of reasoning-oriented LLMs on the ARC benchmark using nine candidate solvers. Experimental results show that repeated-sampling planning-aided code generation (RSPC) achieves the highest test accuracy and exhibits consistent generalization across most evaluated LLMs. To further improve performance, we propose KAAR, which progressively augments LLMs with core knowledge priors structured into hierarchical levels based on their dependencies, and applies RSPC after augmenting each level of priors to enable stage-wise reasoning. KAAR improves the LLM performance on ARC while maintaining strong generalization compared to RSPC. However, ARC remains challenging even for the most capable reasoning-oriented LLMs, given its emphasis on abstract reasoning and generalization, highlighting current limitations and motivating future research.

Limitations

KAAR improves the performance of reasoning-oriented LLMs on ARC tasks by progressively prompting with core knowledge priors. Although this inevitably increases token usage, the trade-off can be justified, as the exploration of LLM generalization remains in its early stages. KAAR integrates diverse abstraction methods to enable objectness and iteratively applies abstractions in order of increasing complexity. In contrast, humans typically infer appropriate abstractions directly from training instances, rather than leveraging exhaustive search. To address this, we prompt different LLMs with raw 2D matrices of each ARC problem to select one or three relevant abstractions, but the results are unsatisfactory. Accurate abstraction inference often depends on validation through viable solutions, thereby shifting the challenge back to solution generation. Given the low solution-generation accuracy reported in Table 1, direct abstraction inference remains a challenging task for evaluated LLMs. Additionally, KAAR augments core knowledge priors through in-context learning but lacks mechanisms to enforce LLM adherence to these priors during reasoning. However, the close correspondence between KAAR-generated solutions and augmented core knowledge priors (Figure 4) suggests a potential underlying alignment. Future work could explore fine-tuning or reinforcement learning to better align model behavior with the desired reasoning patterns.

References

Sam Acquaviva, Yewen Pu, Marta Kryven, Theodoros Sechopoulos, Catherine Wong, Gabrielle Ecanow, Maxwell Nye, Michael Tessler, and Josh Tenenbaum. 2022. Communicating natural programs to humans and machines. In *Proceedings of the 36th Advances in Neural Information Processing Systems*, NeurIPS, pages 3731–3743.

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, EACL, pages 225–237.

Ekin Akyürek, Mehul Damani, Adam Zweiger, Linlu Qiu, Han Guo, Jyothish Pari, Yoon Kim, and Jacob Andreas. 2024. The surprising effectiveness of test-time training for few-shot learning. *arXiv preprint arXiv:2411.07279*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Zana H Babakr, Pakstan Mohamedamin, and Karwan Kakamad. 2019. Piaget's cognitive developmental theory: Critical review. *Education Quarterly Reviews*, 2(3):517–524.

Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. 2023. A multitask, multilingual, multimodal evaluation of ChatGPT on reasoning, hallucination, and interactivity. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, IJCNLP-AACL, pages 675–718.

David Barrett, Felix Hill, Adam Santoro, Ari Morcos, and Timothy Lillicrap. 2018. Measuring abstract reasoning in neural networks. In *Proceedings of the 37th International conference on machine learning*, ICML, pages 511–520.

Kiril Bikov, Mikel Bober-Irizar, and Soumya Banerjee. 2024. Reflection system for the abstraction and reasoning corpus. In *Proceedings of the 2nd AI4Research Workshop: Towards a Knowledge-grounded Scientific Research Lifecycle*.

Giacomo Camposampiero, Loic Houmard, Benjamin Estermann, Joël Mathys, and Roger Wattenhofer. 2023. Abstract visual reasoning enabled by language. *arXiv preprint arXiv:2306.04091*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. Codet: Code generation with generated tests. In *Proceedings of the 11th International Conference on Learning Representations*, ICLR, pages 1–19.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *Proceedings of the 12th International Conference on Learning Representations*, ICLR.

François Chollet. 2019. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*.

Alibaba Cloud. 2025. Alibaba cloud unveils qwq-32b: A compact reasoning model with cutting-edge performance. *Alibaba Cloud*. Accessed: 2025-03-22.

Anthony G Cohn and Jose Hernandez-Orallo. 2023. Dialectical language model evaluation: An initial appraisal of the commonsense spatial reasoning abilities of llms. *arXiv preprint arXiv:2304.11164*.

Google DeepMind. 2024. Gemini 2.0 flash thinking. *Google DeepMind*. Accessed: 2025-03-22.

Hourui Deng, Hongjie Zhang, Jie Ou, and Chaosheng Feng. 2024. Can llm be a good path planner based on prompt engineering? mitigating the hallucination for path planning. *arXiv preprint arXiv:2408.13184*.

Daniel Franzen, Jan Disselhoff, and David Hartmann. 2025. Product of experts with llms: Boosting performance on arc is a matter of perspective. *arXiv preprint arXiv:2505.07859*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Michael Hodel. 2024. Addressing the abstraction and reasoning corpus via procedural example generation. *arXiv preprint arXiv:2404.07353*.

Keya Hu, Ali Cy, Linlu Qiu, Xiaoman Delores Ding, Runqian Wang, Yeyin Eva Zhu, Jacob Andreas, and Kaiming He. 2025. Arc is a vision problem! *arXiv preprint arXiv:2511.14761*.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, ACL, pages 4912–4944.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *Proceedings of the 13th International Conference on Learning Representations*, ICLR.

Xue Jiang, Yihong Dong, Lecheng Wang, Fang Zheng, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2023. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–28.

Aysja Johnson, Wai Keen Vong, Brenden M Lake, and Todd M Gureckis. 2021. Fast and flexible: Human program induction in abstract reasoning tasks. *arXiv preprint arXiv:2103.05823*.

Seungpil Lee, Woochang Sim, Donghyeon Shin, Wongyu Seo, Jiwon Park, Seokki Lee, Sanha Hwang, Sejin Kim, and Sundong Kim. 2024. Reasoning abilities of large language models: In-depth analysis on the abstraction and reasoning corpus. *ACM Transactions on Intelligent Systems and Technology*.

Solim LeGris, Wai Keen Vong, Brenden M Lake, and Todd M Gureckis. 2024. H-arc: A robust estimate of human performance on the abstraction and reasoning corpus benchmark. *arXiv preprint arXiv:2409.01374*.

Chao Lei, Yanchuan Chang, Nir Lipovetzky, and Krista A Ehinger. 2024a. Planning-driven programming: A large language model programming workflow. *arXiv preprint arXiv:2411.14503*.

Chao Lei, Nir Lipovetzky, and Krista A Ehinger. 2023. Novelty and lifted helpful actions in generalized planning. In *Proceedings of the International Symposium on Combinatorial Search*, SoCS, pages 148–152.

Chao Lei, Nir Lipovetzky, and Krista A Ehinger. 2024b. Generalized planning for the abstraction and reasoning corpus. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence*, AAAI, pages 20168–20175.

Wenhao Li, Yudong Xu, Scott Sanner, and Elias Boutros Khalil. 2024a. Tackling the abstraction and reasoning corpus with vision transformers: the importance of 2d representation, positions, and objects. *arXiv preprint arXiv:2410.06405*.

Xingxuan Li, Ruochen Zhao, Yew Ken Chia, Bosheng Ding, Shafiq Joty, Soujanya Poria, and Lidong Bing. 2024b. Chain-of-knowledge: Grounding large language models via dynamic knowledge adapting over heterogeneous sources. In *Proceedings of the 12th International Conference on Learning Representations*, ICLR.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378:1092–1097.

Mikołaj Małkiński and Jacek Mańdziuk. 2023. A review of emerging research directions in abstract visual reasoning. *Information Fusion*, 91:713–736.

Silin Meng, Yiwei Wang, Cheng-Fu Yang, Nanyun Peng, and Kai-Wei Chang. 2024. LLM-a*: Large language model enhanced incremental heuristic search on path planning. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 1087–1102.

Grégoire Mialon, Roberto Dessi, Maria Lomeli, Christoforos Nalmpantis, Ramakanth Pasunuru, Roberta Raileanu, Baptiste Roziere, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. 2023. Augmented language models: a survey. *Transactions on Machine Learning Research*.

Tan John Chong Min. 2023. An approach to solving the abstraction and reasoning corpus (arc) challenge. *arXiv preprint arXiv:2306.03553*.

Arseny Moskvichev, Victor Vikram Odouard, and Melanie Mitchell. 2023. The conceptarc benchmark: Evaluating understanding and generalization in the arc domain. *arXiv preprint arXiv:2305.07141*.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, ICML, pages 26106–26128.

OpenAI. 2025. Openai o3-mini. *OpenAI*. Accessed: 2025-03-22.

Charles S Peirce. 1868. Questions concerning certain faculties claimed for man. *The Journal of Speculative Philosophy*, 2(2):103–114.

Shuofei Qiao, Honghao Gui, Chengfei Lv, Qianghuai Jia, Huajun Chen, and Ningyu Zhang. 2024. Making language models better tool learners with execution feedback. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, NNACL, pages 3550–3568.

John Raven. 2000. The raven's progressive matrices: change and stability over culture and time. *Cognitive psychology*, 41(1):1–48.

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

John Chong Min Tan and Mehul Motani. 2024. Llms as a system of multiple expert agents: An approach to solve the abstraction and reasoning corpus (arc) challenge. In *Proceedings of the 2024 IEEE Conference on Artificial Intelligence*, CAI, pages 782–787.

Zhao Tian and Junjie Chen. 2023. Test-case-driven programming understanding in large language models for better code generation. *arXiv preprint arXiv:2309.16120*.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2023. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, ACL, pages 10014–10037.

Tu Vu, Mohit Iyyer, Xuezhi Wang, Noah Constant, Jerry Wei, Jason Wei, Chris Tar, Yun-Hsuan Sung, Denny Zhou, Quoc Le, and Thang Luong. 2024. Fresh-LLMs: Refreshing large language models with search engine augmentation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13697–13720.

Ruocheng Wang, Eric Zelikman, Gabriel Poesia, Yewen Pu, Nick Haber, and Noah Goodman. 2024. Hypothesis search: Inductive reasoning with language models. In *Proceedings of the 12 th International Conference on Learning Representations*, ICLR.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th Advances in Neural Information Processing Systems*, NeurIPS, pages 24824–24837.

J S Wind. 2020. 1st place solution + code and official documentation. https://www.kaggle.com/competitions/abstraction-and-reasoning-challenge/discussion/154597. Accessed: 2025-03-22.

Yudong Xu, Elias B Khalil, and Scott Sanner. 2023a. Graphs, constraints, and search for the abstraction and reasoning corpus. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence*, AAAI, pages 4115–4122.

Yudong Xu, Wenhao Li, Pashootan Vaezipoor, Scott Sanner, and Elias B Khalil. 2023b. Llms and the abstraction and reasoning corpus: Successes, failures, and the importance of object-based representations. *arXiv preprint arXiv:2305.18354*.

Yutaro Yamada, Yihan Bao, Andrew Kyle Lampinen, Jungo Kasai, and Ilker Yildirim. 2024. Evaluating spatial understanding of large language models. *Transactions on Machine Learning Research*.

Yuhang Zang, Wei Li, Jun Han, Kaiyang Zhou, and Chen Change Loy. 2025. Contextual object detection with multimodal large language models. *International Journal of Computer Vision*, 133(2):825–843.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. 2023. Coder reviewer reranking for code generation. In *Proceedings of the 40th International Conference on Machine Learning*, ICML, pages 41832–41846.

Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. 2019. Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems*, 30(11):3212–3232.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870.

Yuqi Zhu, Shuofei Qiao, Yixin Ou, Shumin Deng, Shiwei Lyu, Yue Shen, Lei Liang, Jinjie Gu, Huajun Chen, and Ningyu Zhang. 2025. KnowAgent: Knowledge-augmented planning for LLM-based agents. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3709–3732.

Appendix

A Related Work

Knowledge-Augmented LLMs. Augmenting LLMs with external knowledge can improve reasoning capabilities and mitigate hallucination in different tasks (Mialon et al., 2023). Previous studies achieve this by incorporating domain-specific knowledge, designed by human experts (Zhu et al., 2025), retrieved via search engines (Vu et al., 2024), or extracted from Wikipedia documents (Li et al., 2024b). Trivedi et al. (2023) demonstrated that interleaving knowledge augmentation within reasoning steps further reduces model hallucination, resulting in more accurate multi-step reasoning. Additionally, augmenting LLMs with execution feedback improves performance on both question answering (Qiao et al., 2024) and program synthesis tasks (Lei et al., 2024b; Zhong et al., 2024; Chen et al., 2024). KAAR follows the interleaved knowledge augmentation approach and further introduces an ordered knowledge augmentation strategy based on knowledge hierarchical dependencies to mitigate context interference and facilitate stage-wise reasoning. Its superior performance over both the non-augmented variant (RSPC) and the non–stage-wise reasoning counterpart (Appendix G) underscores its effectiveness.

Search in DSL. An abstract, expressive, and compositional representation of core knowledge priors is essential for solving ARC tasks (Chollet, 2019). Recent studies have manually encoded core knowledge priors into domain-specific languages (DSLs) with lifted representations (Xu et al., 2023a; Lei et al., 2024b; Wind, 2020). Various program synthesis methods have been proposed to search for valid program solutions within their DSLs, including DAG-based search (Wind, 2020), graph-based constraint-guided search used in ARGA (Xu et al., 2023a), and generalized planning employed in GPAR (Lei et al., 2024b).

KAAR benefits from the hand-crafted DSL in GPAR, where core knowledge priors are encoded with high precision and interpretability. Unlike GPAR, which performs an exhaustive search over the entire DSL, resulting in significant inefficiencies, KAAR leverages LLMs that pre-trained on extensive code-related corpora to synthesize programs. Moreover, the underlying code generation strategy in KAAR, RSPC, ensures high-quality and accurate program solutions. KAAR further formalizes the core knowledge priors into a dependency-aware ontology, which preserves knowledge coverage and enables progressive knowledge augmentation for LLMs to support stage-wise reasoning, thereby reducing interference from irrelevant and noisy context.

LLMs for ARC. Recent studies have explored using LLMs as ARC solvers to directly generate test output matrices and have provided LLMs with different problem descriptions to improve output accuracy. Camposampiero et al. (2023) employed LLMs to generate output grids from textual task descriptions, derived from a vision module which is designed to capture human-like visual priors. Min (2023) prompted LLMs with the raw 2D matrices of each task, along with transformation and abstraction examples. Xu et al. (2023b) demonstrated that object representations produced by predefined abstractions can improve LLM performance on ARC tasks. Diverging from direct image generation, Tan and Motani (2024) evaluated LLM performance on the ARC benchmark by generating Python program solutions. Additionally, Wang et al. (2024) approached ARC as an inductive reasoning problem and introduced hypothesis search, where program solutions are generated by selecting LLM-generated hypotheses encoded as functions.

When leveraging LLMs as ARC solvers, existing studies tend to emphasize accuracy on partial training set problems and overlook the core principle of ARC, where solutions should be constructed using core knowledge priors (Chollet, 2019). However, LLMs still lack these priors, such as objectness, as evidenced by RSPC-generated solutions. KAAR addresses this gap by progressively augmenting LLMs with structured core knowledge priors. KAAR achieves strong performance, 32.5% test accuracy on the full evaluation set of 400 problems using GPT-o3-mini. It demonstrates substantial generalization, and produces solutions that adhere to core knowledge priors

Training-Based Methods. The few training examples available per ARC task restrict the effectiveness of fine-tuning for achieving robust generalization and accurate reasoning in billion-parameter LLMs. To address this limitation, Bikov et al. (2024) fine-tuned LLMs on augmented ARC tasks constructed through standard image augmentation operations, including rotation, flipping, and permutation. Akyürek et al. (2024) applied test-time training to further improve LLM performance on ARC. Models are first fine-tuned on large-scale synthetic ARC data (RE-ARC) (Hodel, 2024) and

subsequently optimized for each ARC task using original input–output pairs, augmented examples, and leave-one-out examples. Extending this framework, Franzen et al. (2025) incorporated augmented data across test-time training, inference, and selection, following initial fine-tuning on RE-ARC, further advancing LLM performance on the ARC benchmark. Hu et al. (2025) introduced a similar two-stage ViT training paradigm, with initial fine-tuning on RE-ARC followed by test-time training with canvas-derived augmented examples for each ARC tasks.

Training approaches have achieved state-of-the-art performance on ARC. However, they diverge from ARC's original goal of evaluating human-like intelligence, which involves performing abstract reasoning and generalization from limited examples without reliance on externally generated data (Chollet, 2019). Human solvers typically infer solutions solely from the provided examples (Johnson et al., 2021; Acquaviva et al., 2022). Given this, training with additionally augmented data is inconsistent with ARC's intent. Instead, learning core knowledge priors and performing inference grounded in them offers a promising alternative. KAAR adopts this idea by augmenting LLMs through in-context learning with core knowledge priors, thereby preserving generalization while enhancing performance on ARC tasks.

B Core Knowledge Priors in KAAR

KAAR incorporates abstractions to enable objectness priors; leverages component attributes, relations, and statistical analysis of component attributes to encode geometry, topology, numbers, and counting priors; and employs predefined actions to support goal-directedness priors. Table 7 presents all abstractions used in KAAR, organized by their prioritization. KAAR incorporates fundamental abstractions, such as 4-connected and 8-connected components, from GPAR, and extends them with additional abstractions unique to KAAR, highlighted in red. Table 8 introduces geometry, topology, numbers, and counting priors, and ten predefined transformations used in KAAR. For each action, KAAR augments the LLM with its corresponding schema to resolve implementation details. The actions and their schemas are detailed in Table 9. Most actions can be specified within three steps, keeping them tractable for LLMs.

		KAAR	KAAR-P	Δ
Gemini-2.0	I_r	25.75	22.25	-3.50
	I_t	21.75	18.75	-3.00
	$I_r\&I_t$	20.50	17.75	-2.75
QwQ-32B	I_r	22.25	18.75	-3.50
	I_t	21.00	17.25	-3.75
	$I_r\&I_t$	19.25	16.00	-3.25
DeepSeek-R1-70B	I_r	12.25	11.25	-1.00
	I_t	12.75	11.00	-1.75
	$I_r\&I_t$	11.50	11.00	-0.50

Table 3: Accuracy on I_r, I_t, and $I_r\&I_t$ for KAAR and KAAR-P across three LLMs. KAAR-P denotes KAAR without the newly introduced priors, retaining only the priors from GPAR. Δ denotes the performance drop relative to KAAR. All values are reported as percentages.

		KAAR	KAAR-S	Δ
Gemini-2.0	I_r	25.75	23.00	-2.75
	I_t	21.75	19.00	-2.75
	$I_r\&I_t$	20.50	18.00	-2.50
QwQ-32B	I_r	22.25	18.50	-3.75
	I_t	21.00	17.75	-3.25
	$I_r\&I_t$	19.25	16.25	-3.00
DeepSeek-R1-70B	I_r	12.25	9.00	-3.25
	I_t	12.75	9.00	-3.75
	$I_r\&I_t$	11.50	8.50	-3.00

Table 4: Accuracy on I_r, I_t, and $I_r\&I_t$ for KAAR and KAAR-S across three LLMs. KAAR-S invokes the solver backbone (RSPC) only after all knowledge priors are augmented. Δ denotes the performance drop relative to KAAR. All values are reported as percentages.

C Restrictions in KAAR

For certain abstractions, some priors are either inapplicable or exclusive. The specific priors assigned to some abstractions are detailed in Table 10. For the *whole image* abstraction, few priors are applicable as the abstraction contains only a single component. In contrast, the *4/8-connected-multi-color-non-background* abstractions retain most priors. The highlighted priors that capture per-component color diversity are used exclusively for *4/8-connected-multi-color-non-background* abstractions, while priors tailored to a single-color component, such as *components with same color*, *components with most frequent color*, and *components with least frequent color*, are excluded. For the *middle-vertical* and *middle-horizontal* abstractions, where the image is evenly divided into two

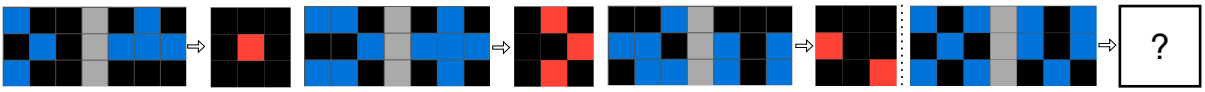


Figure 9: ARC problem *0520fde7*

components, flipping and movement actions are enabled to facilitate reasoning over overlapping components. For instance, in the problem shown in Figure 9, the solution involves splitting the image along a middle-vertical grid line and moving one component to overlap the other. In the resulting component, a pixel is colored red if the overlapping pixels in both components are blue; otherwise, it is colored black.

D Parameter Settings

KAAR operates on all LLMs through API access with the full conversational history. For proprietary models, GPT-o3-mini and Gemini-2.0 Flash-Thinking (Gemini-2.0), we use the 2025-01-31 snapshot and the exp-1219 version, respectively, with default parameter settings. For open-source models, DeepSeek-R1-Distill-Llama-70B (DeepSeek-R1-70B) and QwQ-32B, we set temperature to 0.6, top-p to 0.95, and top-k to 40 to reduce repetitive outputs and filter rare tokens while preserving generation diversity. We conduct experiments on a virtual machine with 4 NVIDIA A100 80GB GPUs. Our code is included in the software submission and will be made publicly available in the camera-ready version.

E KAAR

Algorithm 1 presents the pseudocode of KAAR. For each abstraction, KAAR incrementally augments the LLM with core knowledge priors, structured into three dependency-aware levels: beginning with objectness (Line 5), followed by geometry and topology (Lines 10 and 12), numbers and counting (Line 14), and concluding with goal-directedness priors(Line 18). We note that KAAR encodes geometry and topology priors through component attributes (Line 9) and relations (Line 11). The full set of priors is detailed in Tables 7, 8, and 9. After augmenting each level of priors, KAAR invokes the solver backbone (RSPC) at Lines 6, 15, and 19 to generate code solutions guided by text-based plans, allowing up to 4 iterations (Lines 25–37). In each iteration, the solver backbone first validates the generated code on the training instances I_r; if successful, it then evaluates the solution on the test instances I_t. The solver backbone returns `solve` if the generated solution successfully solves I_t after passing I_r; `pass` if only I_r is solved; or continues to the next iteration if the solution fails on I_r. If the solver backbone fails to solve I_r within the allotted 4 iterations at Lines 6 and 15, KAAR augments the next level of priors. KAAR proceeds to the next abstraction when the solver backbone fails to solve I_r at Line 19, after the 4-iteration limit. KAAR terminates abstraction iteration upon receiving either `pass` or `solve` from the solver backbone and reports accuracy on I_r, I_t, and $I_r \& I_t$ accordingly. If no abstraction fully solves I_r, KAAR records the final code solution for each abstraction (Line 22), selects the one that passes the most training instances (Line 23), and evaluates it on I_t to determine additional accuracy gains (Line 24).

KAAR generates priors offline using image processing algorithms at Lines 4, 9, 11 and 13. In contrast, KAAR enables goal-directedness priors at Line 18 by prompting the LLM to select the most suitable actions and identify their implementation details, as described in Table 9. KAAR iterates over abstractions from simpler to more complex, following the order specified in Table 7. We note that the highest-priority abstraction is *no abstraction*, where KAAR degrades to the solver backbone (RSPC) as no priors are applied.

F Example Tasks by Category in the ARC Evaluation Set

ARC comprises 1000 unique tasks, with 400 allocated to the training set and 600 to the evaluation set. The evaluation set is further divided into a public subset (400 tasks) and a private subset (200 tasks). Figure 10 illustrates example ARC tasks for the *movement*, *extension*, *recolor*, and *others* categories in the public evaluation set. In the *movement* example, components are shifted to the image boundary in directions determined by their colors. The *extension* example is more complex, requiring LLMs to find the shortest path between two red pixels while avoiding obstacles, which presents challenges for current reasoning-oriented models. Ad-

ditionally, reliance on pixel-level recognition weakens the effectiveness of KAAR, which is designed to facilitate component identification. The *recolor* example involves changing non-black components to black and updating black components based on original non-black colors. The *others* example requires generating a blue diagonal line whose length depends on the number of 4-connected components that are green and have a size greater than one in the input image. The combination of numerical reasoning and structural pattern generation makes this task difficult to classify within the other three categories.

G Ablation Study

KAAR adopts the core-knowledge priors in GPAR and extends them with additional priors required for solving ARC tasks. These priors are organized within a dependency-aware ontology that facilitates progressive knowledge augmentation and stage-wise reasoning. Table 3 shows that excluding the newly introduced priors while preserving priors from GPAR and progressive knowledge augmentation with stage-wise reasoning, yielding the variant KAAR-P, results in a substantial performance degradation relative to KAAR. We note that, for the goal-directedness priors, KAAR-P prompts LLMs to identify the most relevant actions from candidates derived from GPAR without schema and condition-awareness instructions. Table 4 reports the accuracy decrease resulting from removing progressive augmentation and stage-wise reasoning in KAAR, denoted as KAAR-S. Unlike KAAR, which invokes the solver backbone (RSPC) after augmenting each level of priors to enable stage-wise reasoning, KAAR-S uses RSPC to solve the problem within 12 iterations after augmenting all priors at once.

We evaluate KAAR-P and KAAR-S using the same reasoning-oriented LLMs as in Tables 1 and 2, excluding GPT-o3-mini due to its computational cost. KAAR-P and KAAR-S show decreased accuracy across all metrics, I_r, I_t, and $I_r \& I_t$, for all evaluated LLMs. These results underscore the effectiveness of newly introduced priors in KAAR and progressive augmentation with stage-wise reasoning. Presenting all knowledge priors simultaneously introduces superfluous information, which may obscure viable solutions and impair the LLM reasoning accuracy. We note that KAAR constructs the ontology of core knowledge priors based on their dependencies, thereby establishing a fixed augmentation order.

H Cost Analysis

Table 5 reports the average token cost per problem and test accuracy I_t for planning-aided code generation (PC) under three generation strategies: direct generation, repeated sampling, and refinement. Direct generation incurs the lowest token cost but also yields the lowest test accuracy. Compared with refinement PC, repeated sampling PC (PSPC) achieves comparable or higher test accuracy while requiring fewer tokens, demonstrating superior efficiency and effectiveness.

Table 6 reports the average token cost, including both prompts and LLM responses, for knowledge augmentation and the solver backbone (RSPC), when using KAAR as the ARC solver. For each ARC task, we consider the abstraction whose solution solves I_t; if none succeed, the one that passes I_r; otherwise, the abstraction with the lowest token usage is selected. Except for goal-directedness priors, all core knowledge priors in KAAR are generated offline using image processing algorithms, resulting in comparable augmentation costs across all evaluated models. In contrast, token usage by the solver backbone varies substantially due to differences in the LLMs' abstract reasoning and generalization capabilities. GPT-o3-mini solves most tasks efficiently, with the lowest token consumption by the solver backbone, where tokens used for knowledge augmentation account for approximately 62% of the solver backbone's token usage. However, the solver backbone consumes more tokens with QwQ-32B, as QwQ-32B consistently generates longer reasoning traces. In this case, tokens used for knowledge augmentation constitute only 19% of the solver backbone's token usage. Figure 15 illustrates the average token cost for augmenting priors at each level in KAAR.

I Performance Analysis

In Table 1, the incorporation of planning facilitates accurate code generation under the repeated sampling setting. For example, in Figure 13, repeated sampling with planning-aided code generation produces a correct solution using GPT-o3-mini by replicating the input image horizontally or vertically based on the presence of a uniform row or column, as specified in the plan and implemented accordingly in code. In contrast, without planning

Model	Direct Generation PC		Repeated Sampling PC		Refinement PC	
	Tokens	I_t	Tokens	I_t	Tokens	I_t
GPT-o3-mini	11K	22.25%	106K	30.75%	121K	25.75%
Gemini-2.0	13K	6.25%	110K	16.75%	136K	11.75%
QwQ-32B	39K	5.75%	427K	14.25%	489K	14.00%
DeepSeek-R1-70B	23K	4.50%	252K	7.75%	276K	7.75%

Table 5: Average token cost per problem vs. accuracy on test instance I_t for planning-aided code generation (PC) under three generation strategies: direct generation, repeated sampling, and refinement. K is 10^3.

	Knowledge Augmentation	Solver Backbone (RSPC)
GPT-o3-mini	66K	106K
Gemini-2.0	58K	110K
QwQ-32B	79K	427K
DeepSeek-R1-70B	66K	252K

Table 6: Average token cost for knowledge augmentation and solver backbone (RSPC) in KAAR across four evaluated LLMs. K is 10^3.

assistance, standalone code generation produces incomplete logic, considering only whether the first column is uniform to determine the replication direction, which leads to failure on the test instance.

For the ARC benchmark, repeated sampling–based methods achieve higher accuracy on I_r, I_t, and $I_r\&I_t$ compared to refinement-based approaches when using GPT-o3-mini, Gemini-2.0, and DeepSeek-R1-70B as shown in Table 1. Figure 14 presents an ARC problem where repeated sampling with planning-aided code generation yields a correct solution, whereas its refinement variant fails to correct the initial erroneous code, and the flawed logic persists across subsequent refinements when using GPT-o3-mini. Recent studies have shown that refinement struggles when the initial code significantly deviates from the intended functionality (Tian and Chen, 2023). Moreover, the lack of explicit correction instructions in feedback messages often leads to refinements diverging further from the intended solution (Lei et al., 2024a).

J Generalization

Table 1 highlights performance variations across reasoning-oriented LLMs and ARC solvers with respect to both accuracy and generalization. Notably, the ARC solver, repeated sampling with standalone code generation, exhibits a substantial accuracy gap between I_r and $I_r\&I_t$, indicating limited generalization capability when using GPT-o3-mini and Gemini-2.0. In contrast, repeated sampling with planning-aided code generation demonstrates markedly improved generalization by preventing solutions from directly replicating the output matrices of training instances, as illustrated in Figure 16. This output copying, observed under repeated sampling with standalone code generation, accounts for approximately 24% and 95% of 83 and 101 overfitting problems with GPT-o3-mini and Gemini-2.0, respectively. When planning is incorporated, output copying is reduced to around 8% and 35% of 25 and 20 overfitting problems with GPT-o3-mini and Gemini-2.0, respectively.

Planning-aided methods, RSPC and KAAR, demonstrate smaller accuracy gaps in Table 2. However, the generalization challenge persists. One reason is that solutions include low-level logic for the training pairs, thus failing to generalize. Another reason is the usage of incorrect abstractions. KAAR selects the first abstraction that solves I_r, to report accuracy on I_t, while this abstraction may overfit to I_r. Figures 17 and 18 illustrate two ARC problems, *695367ec* and *b1fc8b8e*, where both RSPC and KAAR successfully solve the training instances I_r but fail on the test instances I_t when using GPT-o3-mini. For problem *695367ec*, the correct solution involves generating a fixed 15×15 output image by repeatedly copying the input image, changing its color to black, and adding internal horizontal and vertical lines colored with the original input image's color. However, the RSPC-generated code applies a distinct rule to each input image size without considering generalization. For problem *b1fc8b8e*, the solution requires accurate object recognition despite component contact, and correctly placing each component into one of the four corners. However, RSPC fails to recognize objectness, and its solution deviates from human intuition, being overfitted to I_r. For problems *695367ec* and *b1fc8b8e*, KAAR exhibits the same limitations, although it adopts abstractions to en-

able objectness. KAAR begins with the simplest abstraction, *no abstraction*, where KAAR degrades to RSPC. As a result, it generates the same solution as RSPC and terminates without attempting other abstractions, since the solution already solves I_r and is then evaluated on I_t, resulting in overfitting.

K Solution Coverage

We also report the relative problem coverage across nine ARC solvers based on successful test instance solutions using GPT-o3-mini (Figure 19), Gemini-2.0 (Figure 20), QwQ-32B (Figure 21), and DeepSeek-R1-70B (Figure 22). Each cell (i, j) is computed using the same method described in Figure 5. All solvers exhibit the strongest overall coverage, with pairwise overlap consistently exceeding 0.55 when using GPT-o3-mini. Among them, repeated sampling with standalone (*C*) and planning-aided code generation (*PC*) show the highest coverage, with column values above 0.8 for GPT-o3-mini. This trend persists across Gemini-2.0, QwQ-32B, and DeepSeek-R1-70B. Except for GPT-o3-mini, repeated sampling with planning-aided code generation exhibits better alignment than its standalone code generation counterpart, generally yielding higher coverage values. However, under the direct generation setting, planning-aided code generation shows the opposite behavior, demonstrating lower alignment than standalone code generation across all evaluated LLMs. Among the four evaluated LLMs, all solvers present the lowest average off-diagonal (i.e., $i \neq j$) coverage of 0.603 with DeepSeek-R1-70B, suggesting potential output instability and variability attributable to solver choice.

Algorithm 1: KAAR

```
Input  : LLM M; ARC problem P = (I_r, I_t); description Q = (I_r, {i^i | (i^i, i^o) ∈ I_t});
         abstraction list A; max iterations t = 4
Function KnowledgeAugmentation (M, Q, P, A, t):
    solutionList ← [];
    foreach abstraction abs in A do
        objectnessPriors ← GenerateObjectnessPriors(Q, abs);
        AugmentKnowledge(M, objectnessPriors);
        result, code, passedCount ← SolverBackbone (M, P, Q, t);
        if result ≠ failure then
            return result
        attributePriors ← GenerateAttributePriors(Q, abs);
        AugmentKnowledge(M, attributePriors);
        relationPriors ← GenerateRelationPriors(Q, abs);
        AugmentKnowledge(M, relationPriors);
        numberPriors ← GenerateNumbersCountingPriors(Q, abs);
        AugmentKnowledge(M, numberPriors);
        result, code, passedCount ← SolverBackbone (M, P, Q, t);
        if result ≠ failure then
            return result
        AugmentGoalPriors ← (M, Q, abs);
        result, code, passedCount ← SolverBackbone (M, P, Q, t);
        if result ≠ failure then
            return result
        solutionList.append((code, passedCount));
    bestCode ← SelectMostPassed(solutionList);
    return EvaluateOnTest(bestCode, I_t);
Function SolverBackbone (M, P, Q, t):
    i ← 0;
    while i < t do
        plan ← M.generatePlan(Q);
        code ← M.generateCode(Q, plan);
        passedCount ← EvaluateOnTrain(code, I_r);
        if passedCount == |I_r| then
            if EvaluateOnTest(code, I_t) then
                return solve, code, passedCount;
            else
                return pass, code, passedCount;
        i ← i + 1;
    return failure, code, passedCount;
```

Abstractions	Definitions
No Abstraction	-
Whole Image	We consider the whole image as a component.
Middle-Vertical	We vertically split the image into two equal parts, treating each as a distinct component.
Middle-Horizontal	We horizontally split the image into two equal parts, treating each as a distinct component.
Multi-Lines	We use rows or columns with a uniform color to divide the input image into multiple components.
4/8-Connected	We consider the 4/8-adjacent pixels of the same color as a component.
4/8-Connected-Non-Background	We consider the 4/8-adjacent pixels of the same color as a component, excluding components with the background color.
4/8-Connected-Non-Background-Edge	We consider the 4/8-adjacent pixels of the same color as a component, containing components with the background color when they are not attached to the edges of the image.
4/8-Connected-Multi-Color-Non-Background	We consider 4/8-adjacent pixels as a component, which may contain different colors, while excluding pixels with the background color.
4/8-Connected-Bounding-Box	We consider 4/8-adjacent pixels of the same color, and treat all pixels within their bounding box as a component, which may include different colors.
4/8-Connected-With-Black	We consider the 4/8-adjacent pixels of black color, represented by the value 0, as a component, excluding components with other colors.
Same-Color	We consider pixels of the same color as a component, excluding components with the background color.

Table 7: Abstractions in KAAR. The background color is black if black exists; otherwise, it is the most frequent color in the image. We present abstractions according to their prioritization in KAAR, where the order is given by the table from top to bottom, with 8-connected abstractions appearing after the 4-connected abstractions at the end of the sequence while preserving the order of their corresponding 4-connected counterparts. Abstractions highlighted in red are exclusive to KAAR.

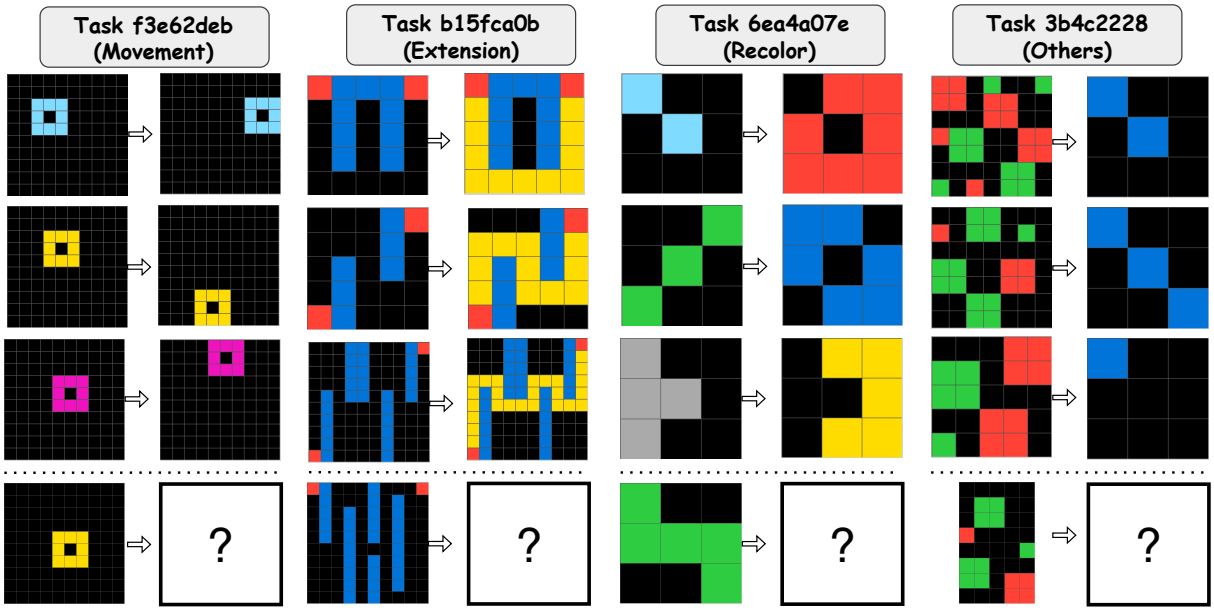


Figure 10: Example ARC tasks for *movement*, *extension*, *recolor*, and *others* categories.

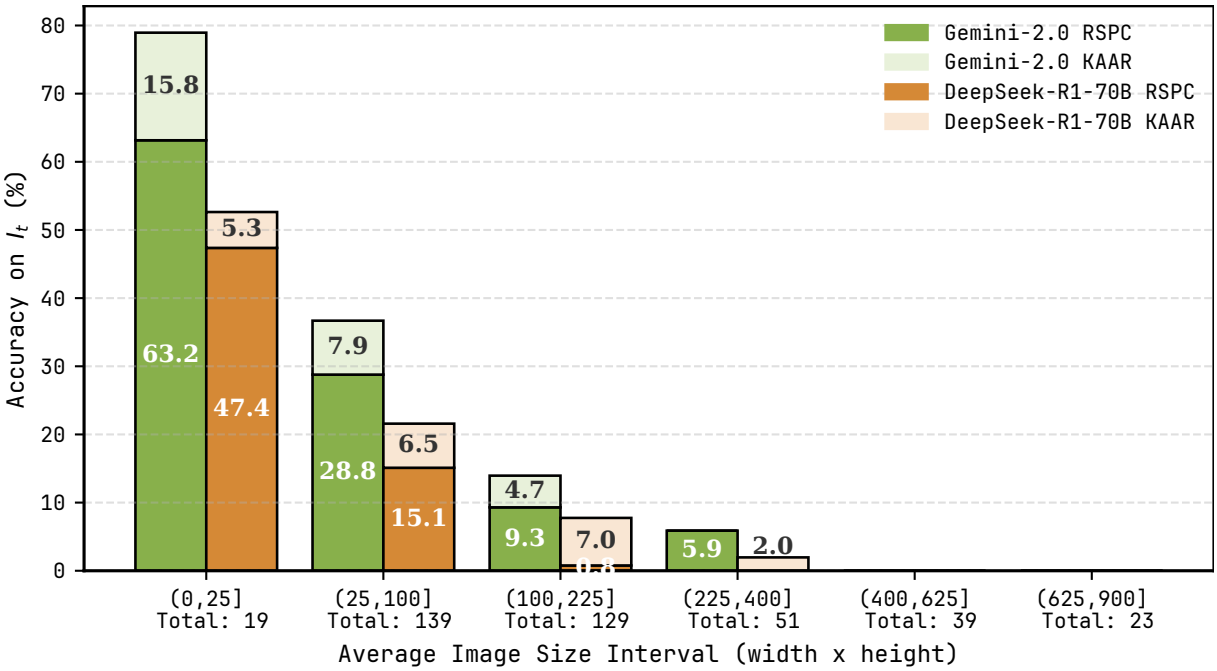


Figure 11: Accuracy on test instances I_t for RSPC and KAAR across average image size intervals, evaluated with Gemini-2.0 and DeepSeek-R1-70B.

Classifications	Priors
Geometry and Topology	Size (Width and Height);
	Color;
	Shape (One Pixel; Horizontal Line; Vertical Line; Diagonal Line; Square; Rectangle; Cross; Irregular Shape);
	Symmetry (Horizontal Symmetry; Vertical Symmetry; Diagonal Symmetry; Anti-Diagonal Symmetry; Central Symmetry);
	Bounding Box;
	Hole Count;
	Nearest Boundary;
	Different/Identical with Other Components;
	Touching;
	Inclusive;
	Spatial (Horizontally Aligned to the Right; Horizontally Aligned to the Left; Vertically Aligned Below; Vertically Aligned Above; Top-Left; Top-Right; Bottom-Left; Bottom-Right; Same Position).
Numbers and Counting	Component Size Counting; Components with Same Size; Components with Most Frequent Size; Components with Least Frequent Size; Components with Maximum Size; Components with Minimum Size;
	Component Color Counting; Components with Same Color; Components with Same Number of Colors; Components with Most Frequent Color; Components with Least Frequent Color; Component with Most Distinct Colors; Component with Fewest Distinct Colors;
	Component Shape Counting; Components with Same Shape; Components with Most Frequent Shape; Components with Least Frequent Shape;
	Component Hole Number Counting; Components with Same Number of Holes; Components with Maximum Number of Holes; Components with Minimum Number of Holes;
	Component Symmetry Counting.
Goal-directedness	Color Change (modifying component value);
	Movement (shifting component's position);
	Extension (expanding component's area);
	Completing (filling in missing parts of a component);
	Resizing (altering component size);
	Selecting (isolating a component);
	Copying (duplicating a component);
	Flipping (mirroring a component);
	Rotation (rotating a component);
	Cropping (cutting part of a component).

Table 8: KAAR priors classified into geometry and topology, numbers and counting, and goal-directedness with priors exclusive to KAAR highlighted in red. For goal-directedness, we incorporate ten predefined actions, with their corresponding action schemas detailed in Table 9.

Actions	Schemas (Implementation Details)					
Color Change	Targets	Source and Target Colors				
Movement	Targets	Direction	Start and End Locations	Pattern	Order	Overlapping
Extension	Targets	Direction	Start and End Locations	Pattern	Order	Intersection
Completing	Targets	Pattern				
Resizing	Targets	Source and Target Sizes				
Selecting	Targets					
Copying	Targets	Locations	Overlapping			
Flipping	Targets	Flipping Axis	Overlapping			
Rotation	Targets	Degrees				
Cropping	Targets	Subsets				

Table 9: Actions in KAAR and their schemas (implementation details). Each action schema is presented according to its prompting order in KAAR (left to right). Some actions include a pattern schema that instructs the LLM to identify underlying logic rules, such as repeating every two steps in movement or extension, or completing based on three-color repetition. Targets denote the target components.

Abstractions	Geometry and Topology	Numbers and Counting	Goal-directedness
whole image	Symmetry, Size	-	Flipping; Rotation; Extension; Completing, Cropping
middle-vertical	Size	-	Flipping; Movement
middle-horizontal	Size	-	Flipping; Movement
multi-lines	Size; Color; Shape; Symmetry; Bounding Box; Hole Count	All	All
4/8-connected-multi-color-non-background	All	... Component Color Counting; Components with Same Number of Colors; Component with Most Distinct Colors; Component with Fewest Distinct Colors ...	All

Table 10: Abstractions with their assigned knowledge priors. "–" denotes no priors, while "All" indicates all priors in the corresponding category, as defined in Table 8. For the *4/8-connected-multi-color-non-background* abstractions, we present color-counting priors, highlighted in red, specific to multi-colored components, while all other non-color-counting priors follow those in Table 8.

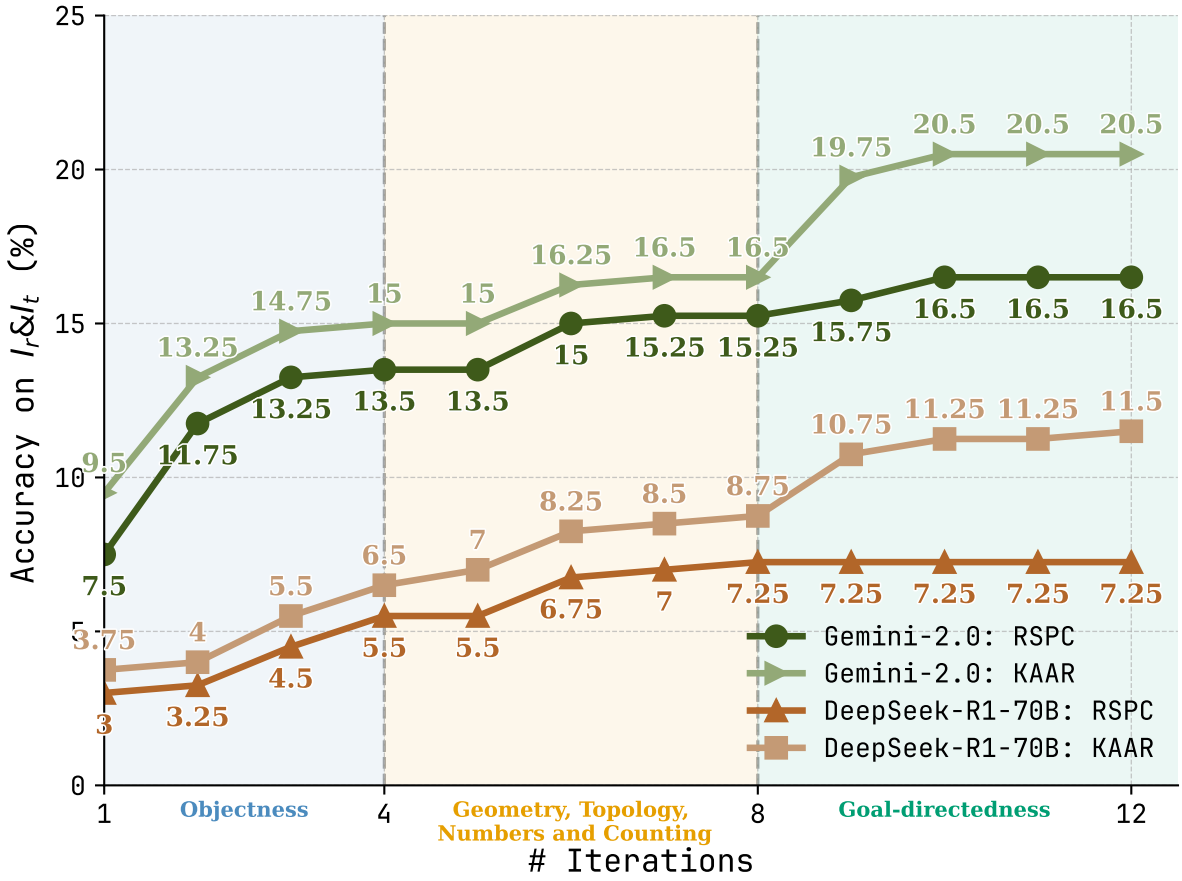


Figure 12: Variance in accuracy on I_r&I_t with increasing iterations for RSPC and KAAR using Gemini-2.0 and DeepSeek-R1-70B.

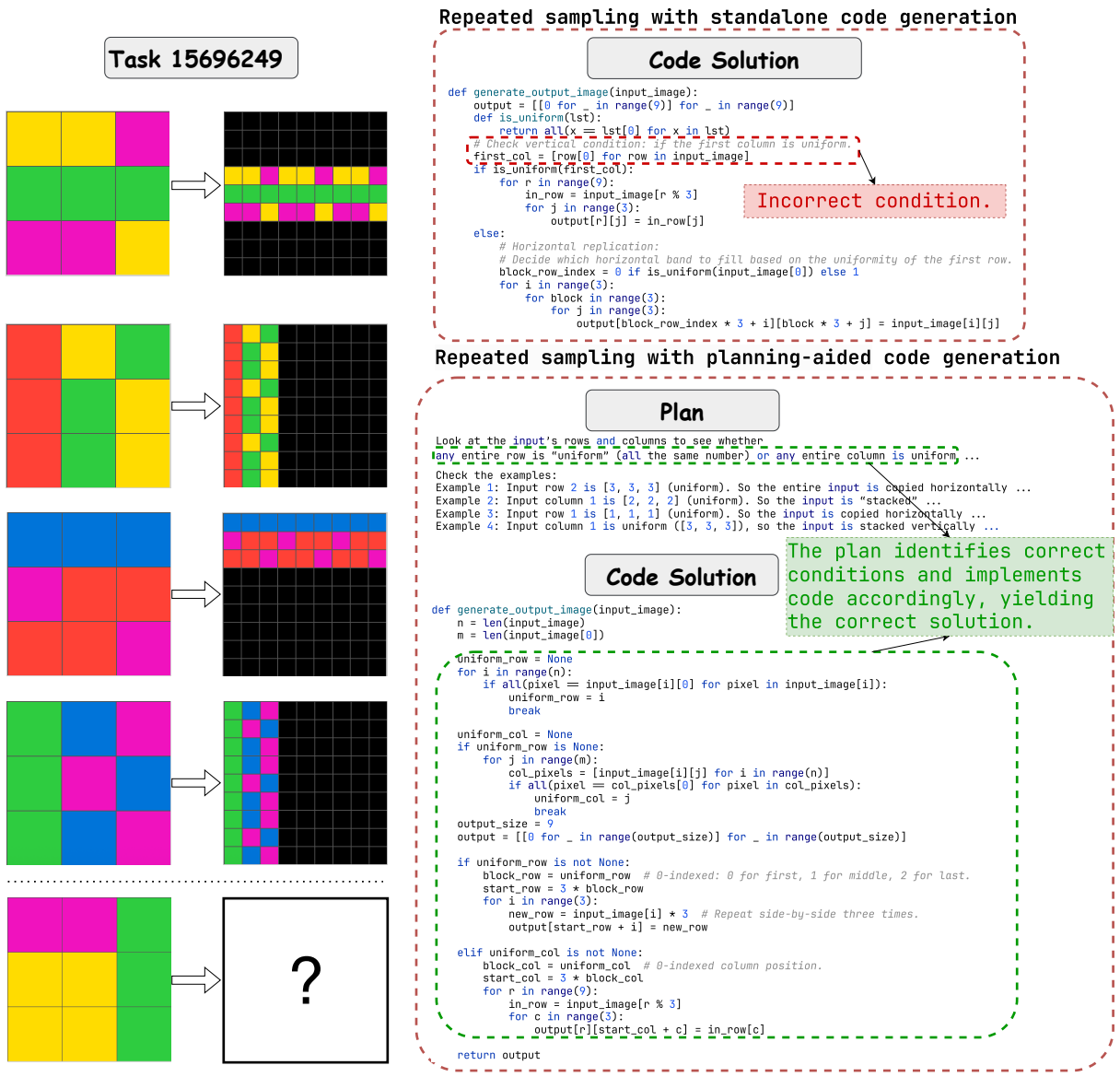
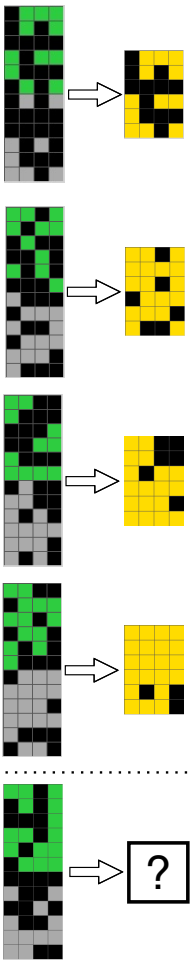


Figure 13: ARC problem *15696249*, where repeated sampling with standalone code generation produces an incorrect solution, whereas repeated sampling with planning-aided code generation yields the correct solution using GPT-o3-mini.

Refinement with planning-aided code generation

Task d19f7514




```python
def generate_output_image(input_image):
    # Step 1: Discard the lower 6 rows to work only with the top 6 rows.
    cropped = [row[:] for row in input_image[:6]]

    # Step 2: Change every nonzero pixel to 4 (convert 3's or any nonzero values to 4).
    for r in range(len(cropped)):
        for c in range(len(cropped[r])):
            if cropped[r][c] != 0:
                cropped[r][c] = 4

    # Step 3: Vertical propagation.
    # Process from the second row (index 1) downwards
    for r in range(1, len(cropped)):
        for c in range(len(cropped[r])):
            # If this pixel is 0 and the pixel immediately above is 4, fill it with 4.
            if cropped[r][c] == 0 and cropped[r-1][c] == 4:
                cropped[r][c] = 4

    return cropped
```

Initial incorrect code

```python
def generate_output_image(input_image):

    cropped = input_image[:6]
    rows = len(cropped)
    cols = len(cropped[0])

    # Step 2. Build the base image: every nonzero becomes 4.
    base = [[4 if pixel != 0 else 0 for pixel in row] for row in cropped]

    # The output image starts with the first (top) row of the base.
    output = [base[0][:]]


    def is_active(row):
        return any(pixel != 0 for pixel in row)

    # Process each subsequent row.
    for r in range(1, rows):
        # Start with the base row.
        new_row = base[r][:]

        if is_active(cropped[r]):
            for c in range(cols):
                # Only consider blank cells that were not drawn originally.
                if new_row[c] == 0 and cropped[r][c] == 0:
                    # Propagate only if the cell directly above is 4.
                    if output[r-1][c] == 4:
                        # For a right border cell, propagate automatically.
                        if c == cols - 1:
                            new_row[c] = 4

                        else:
                            left_drawn = (c - 1 >= 0 and cropped[r][c - 1] != 0)
                            right_drawn = (c + 1 < cols and cropped[r][c + 1] != 0)
                            if left_drawn or right_drawn:
                                new_row[c] = 4
        output.append(new_row)
    return output
```

Incorrect code after refinements

Incorrect logic persists after refinements.

Repeated sampling with planning-aided code generation

Code Solution

```python
def generate_output_image(input_image):
    # Split the input image into two halves
    num_rows = len(input_image)
    mid = num_rows // 2
    top_half = []
    bottom_half = []

    # Replace nonzero pixels with 4 in both halves
    for i in range(mid):
        top_half.append([4 if pixel != 0 else 0 for pixel in input_image[i]])
    for i in range(mid, num_rows):
        bottom_half.append([4 if pixel != 0 else 0 for pixel in input_image[i]])

    # Combine the corresponding rows of the top and bottom halves using pixelwise OR (nonzero becomes 4)
    output_image = []
    for i in range(mid):
        combined_row = []
        for j in range(len(top_half[i])):
            # If either pixel in the corresponding column is nonzero (i.e., equals 4), output 4; otherwise 0.
            if top_half[i][j] or bottom_half[i][j]:
                combined_row.append(4)
            else:
                combined_row.append(0)
        output_image.append(combined_row)

    return output_image
```

Correct logic.

Figure 14: ARC problem *d19f7514*, where repeated sampling with planning-aided code generation produces a correct solution, whereas its refinement variant fails to refine the initial erroneous code, and the incorrect logic persists across subsequent refinements when using GPT-o3-mini.

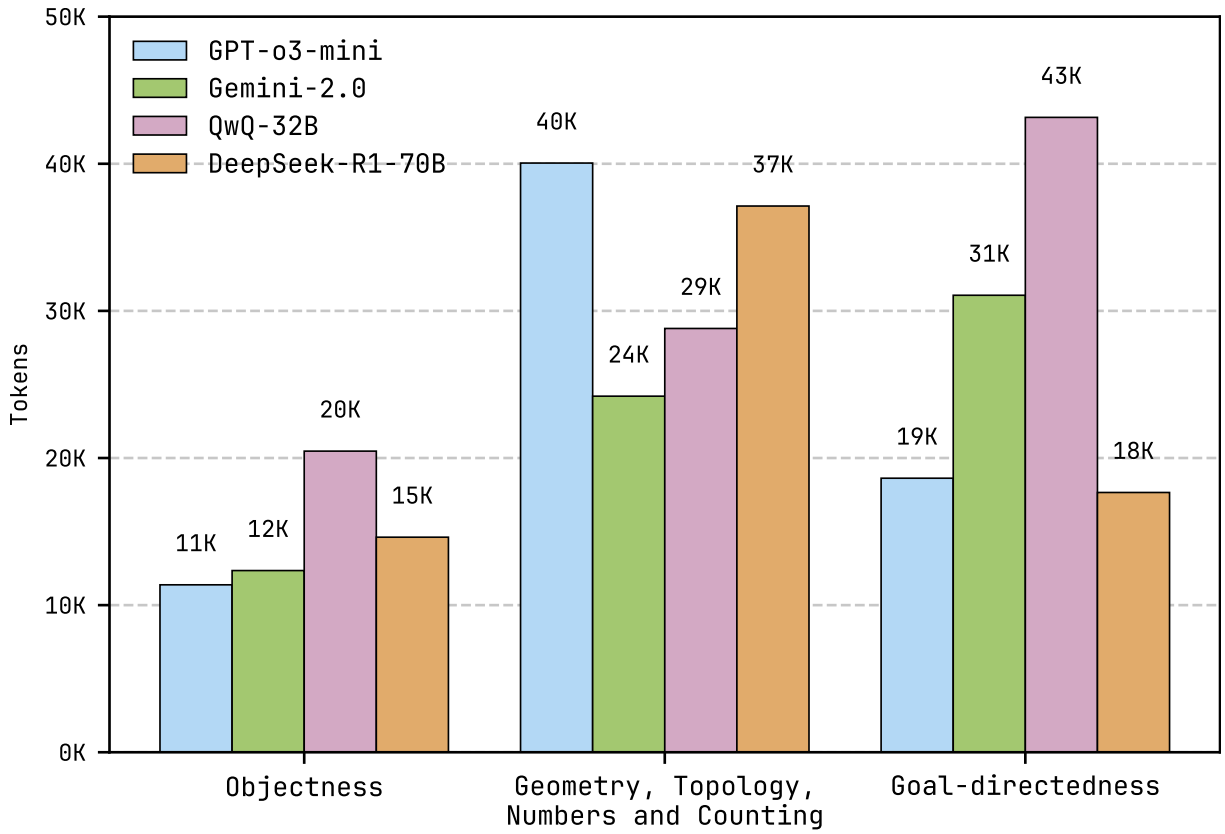
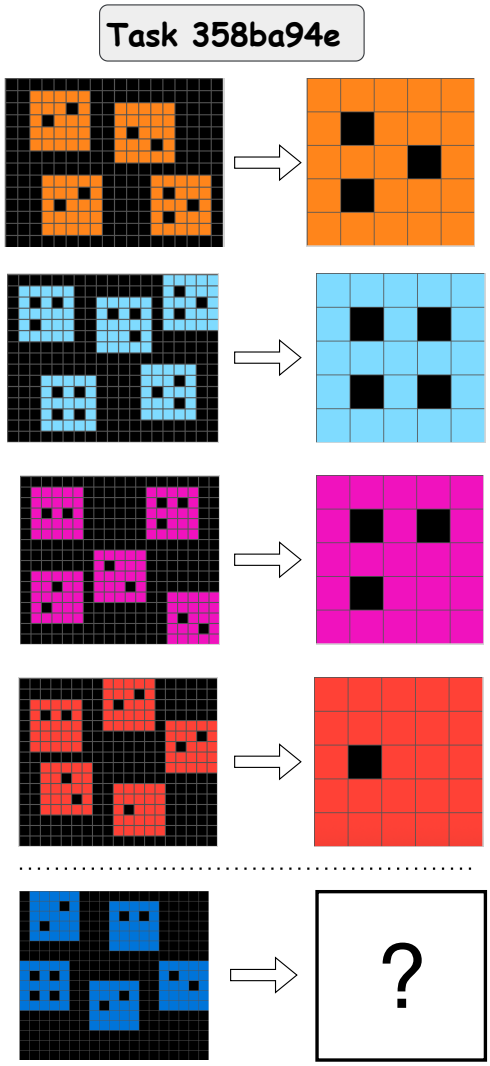


Figure 15: Average token cost for augmenting priors at each level across four LLMs. K is 10^3.



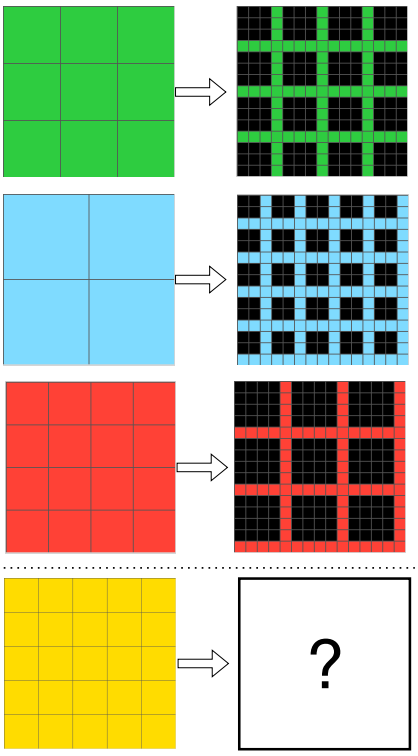
Code Solution

```python
def generate_output_image(input_image):
    freq = {}
    for row in input_image:
        for pix in row:
            if pix != 0:
                freq[pix] = freq.get(pix, 0) + 1
    # If there is no non-zero pixel, return a 5×5 zero matrix.
    if not freq:
        return [[0, 0, 0, 0, 0] for _ in range(5)]
    # Determine the "digit" (nonzero value) that occurs most often.
    digit = max(freq, key=freq.get)

    # Return the fixed normalized 5×5 pattern corresponding to the detected digit.
    if digit == 7:
        return [
            [7, 7, 7, 7, 7],
            [7, 0, 7, 7, 7],
            [7, 7, 7, 0, 7],
            [7, 0, 7, 7, 7],
            [7, 7, 7, 7, 7]
        ]
    elif digit == 8:
        return [
            [8, 8, 8, 8, 8],
            [8, 0, 8, 0, 8],
            [8, 8, 8, 8, 8],
            [8, 0, 8, 0, 8],
            [8, 8, 8, 8, 8]
        ]
    elif digit == 6:
        return [
            [6, 6, 6, 6, 6],
            [6, 0, 6, 0, 6],
            [6, 6, 6, 6, 6],
            [6, 0, 6, 6, 6],
            [6, 6, 6, 6, 6]
        ]
    elif digit == 2:
        return [
            [2, 2, 2, 2, 2],
            [2, 2, 2, 2, 2],
            [2, 0, 2, 2, 2],
            [2, 2, 2, 2, 2],
            [2, 2, 2, 2, 2]
        ]
    elif digit == 1:
        # For the test image the drawn digit is 1.
        # We choose a pattern that follows a similar style to the others.
        return [
            [1, 1, 1, 1, 1],
            [1, 0, 1, 0, 1],
            [1, 1, 1, 1, 1],
            [1, 0, 1, 1, 1],
            [1, 1, 1, 1, 1]
        ]
    else:
        # For any other digit, simply fill a 5x5 matrix with that digit.
        return [[digit]*5 for _ in range(5)]
```

Copy the output matrices.

Figure 16: ARC problem *358ba94e*, where repeated sampling with standalone code generation produces an incorrect solution by replicating the output matrices of training instances when using GPT-o3-mini.

Task 695367ec



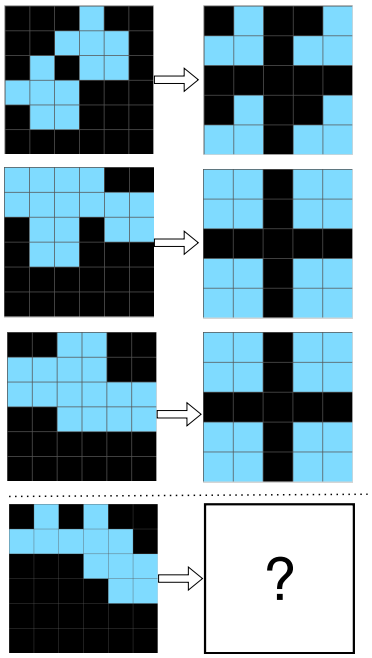
Code Solution

```python
def generate_output_image(input_image):
    # Determine the color value from the input image.
    v = input_image[0][0]

    # Get the input image dimension (assumed square).
    n = len(input_image)

    # Set the output image dimensions (always 15x15 as per the transformation rule).
    out_size = 15

    # Define grid-line indices for rows and columns based on the input dimension.
    # These mappings are inferred from the examples:
    # For 2x2 and 5x5 inputs, grid-lines occur at indices: 2, 5, 8, 11, 14.
    # For a 3x3 input, grid-lines occur at indices: 3, 7, 11.
    # For a 4x4 input, grid-lines occur at indices: 4, 9, 14.
    if n in (2, 5):
        grid_indices = {2, 5, 8, 11, 14}
    elif n == 3:
        grid_indices = {3, 7, 11}
    elif n == 4:
        grid_indices = {4, 9, 14}
    else:
        # Default: evenly space n grid-lines over a 15-element dimension.
        # This is a fallback if the input size is not one of the above.
        block_size = out_size // (n + 1)
        grid_indices = {(i + 1) * block_size - 1 for i in range(n)}

    # Create the 15x15 output image based on the grid drawing rule.
    output = []
    for r in range(out_size):
        if r in grid_indices:
            # This is a separator (grid-line) row; paint the entire row with v.
            output.append([v] * out_size)
        else:
            # For a pattern row, only the pixels at grid-line column positions are painted.
            row = []
            for c in range(out_size):
                if c in grid_indices:
                    row.append(v)
                else:
                    row.append(0)
            output.append(row)

    return output
```

Generate a rule for each input size.

Figure 17: ARC problem *695367ec*, where RSPC and KAAR generate the same code solution that passes the training instances but fails on the test instance using GPT-o3-mini.

Task b1fc8b8e



Code Solution

```python
def generate_output_image(input_image):
    # Determine the border pattern based on the top row of the 6x6 input.
    # Count how many 8's appear in the first row.
    count_eights = sum(1 for pixel in input_image[0] if pixel == 8)

    # If two or more 8's are present, use the "full-active" pattern for active rows.
    if count_eights >= 2:
        active_pattern = [8, 8, 0, 8, 8]
        top_active = active_pattern
        second_active = active_pattern
    else:
        # Use the "softer-border" pattern:
        top_active = [0, 8, 0, 0, 8]
        second_active = [8, 8, 0, 8, 8]

    # The blank row (middle row) is always all zeros.
    blank = [0, 0, 0, 0, 0]

    # Construct the final 5x5 output image.
    # The output consists of:
    #   - The first active row (top_active)
    #   - The second active row (second_active)
    #   - A middle blank row (blank)
    #   - The vertical mirror of the active rows (top_active then second_active).
    output_image = [
        top_active,
        second_active,
        blank,
        top_active,
        second_active
    ]
    return output_image
```

No objective-centric reasoning.

Rules are only applied to training instances.

Figure 18: ARC problem *b1fc8b8e*, where RSPC and KAAR generate the same code solution that passes the training instances but fails on the test instance using GPT-o3-mini.

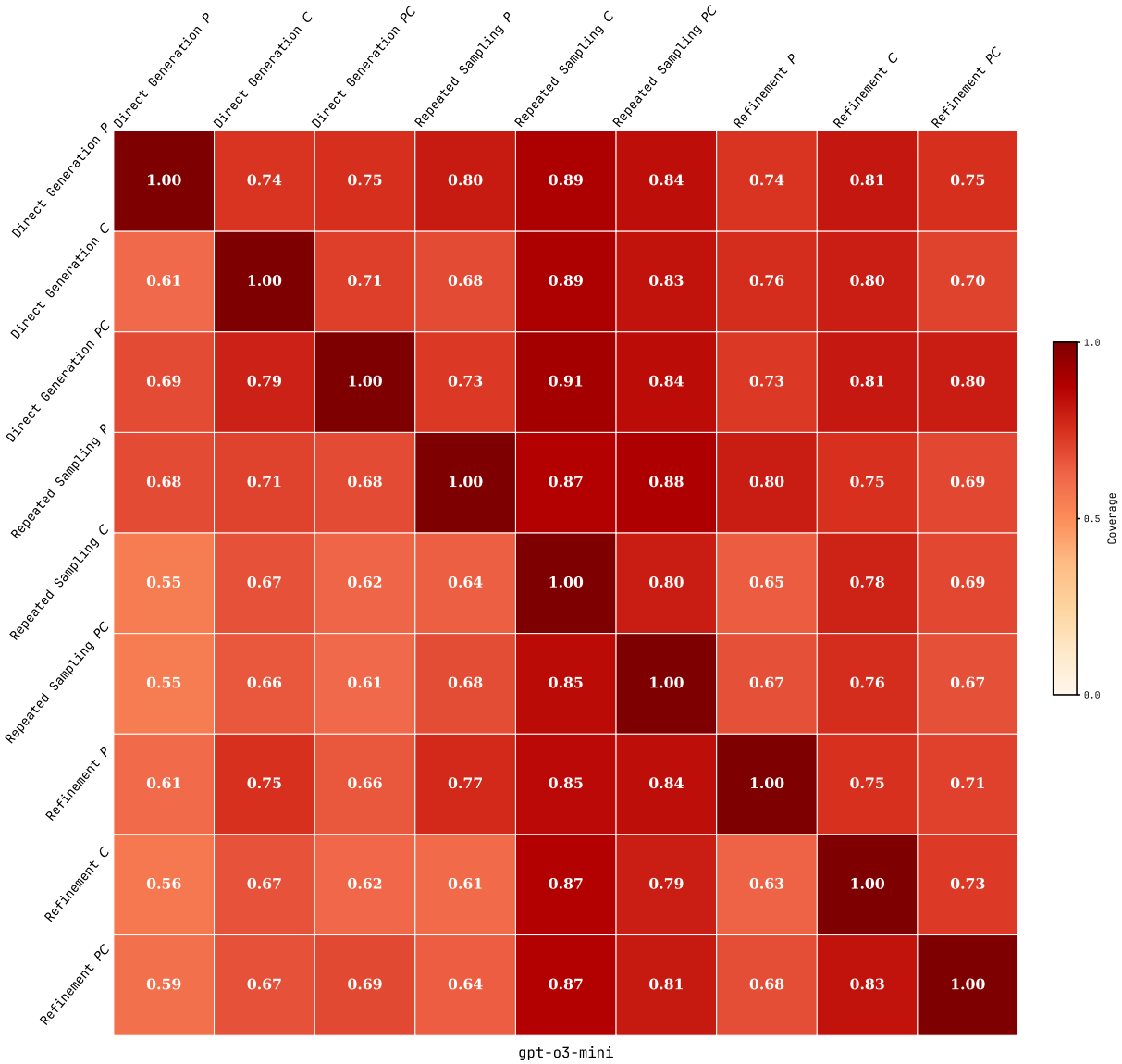


Figure 19: Asymmetric relative coverage matrix of nine ARC solvers using GPT-o3-mini, showing the proportion of problems whose test instances are solved by the row solver that are also solved by the column solver. *P* denotes the solution plan; *C* and *PC* refer to standalone and planning-aided code generation, respectively.

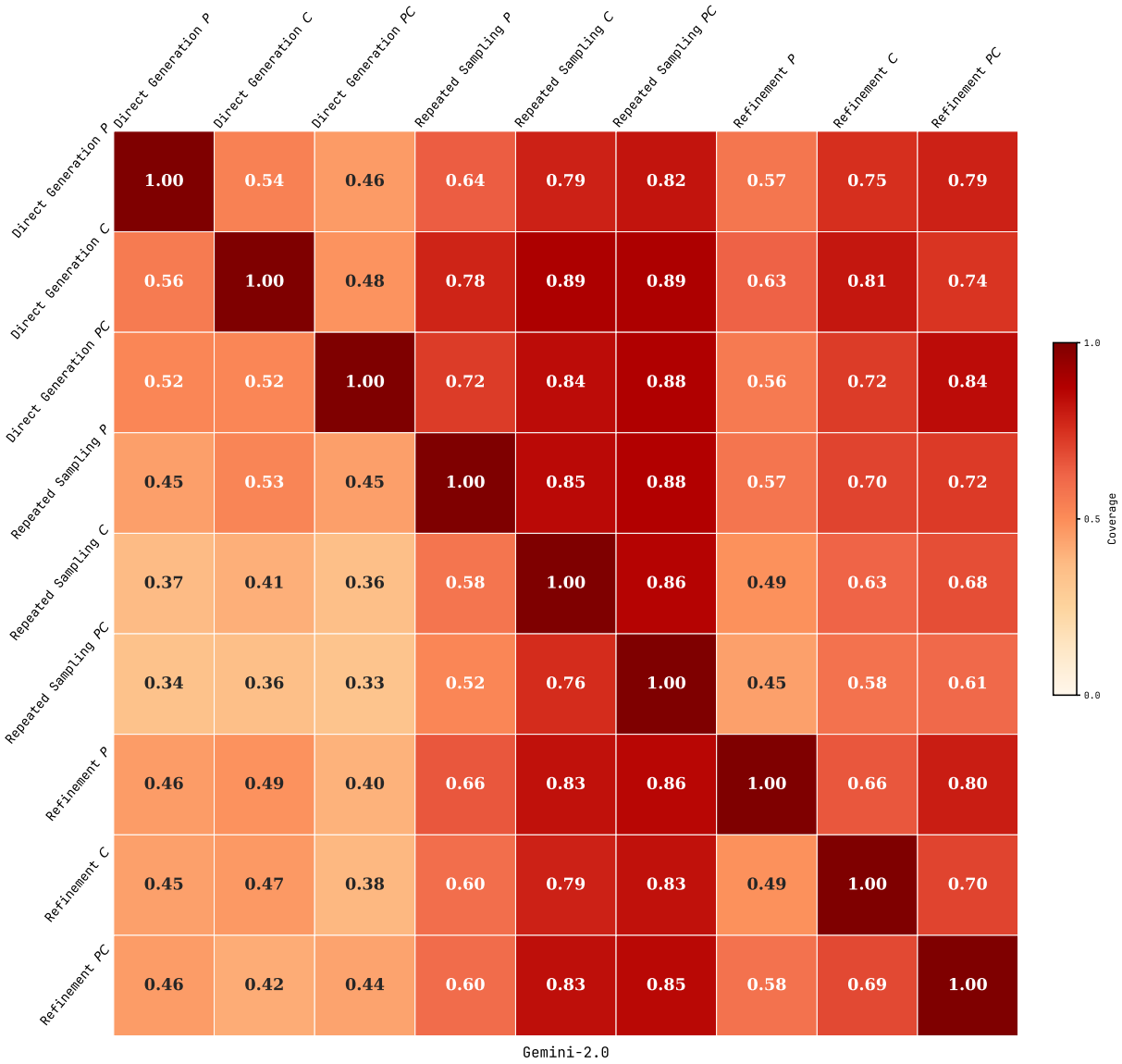


Figure 20: Asymmetric relative coverage matrix of nine ARC solvers using Gemini-2.0, showing the proportion of problems whose test instances are solved by the row solver that are also solved by the column solver. *P* denotes the solution plan; *C* and *PC* refer to standalone and planning-aided code generation, respectively.

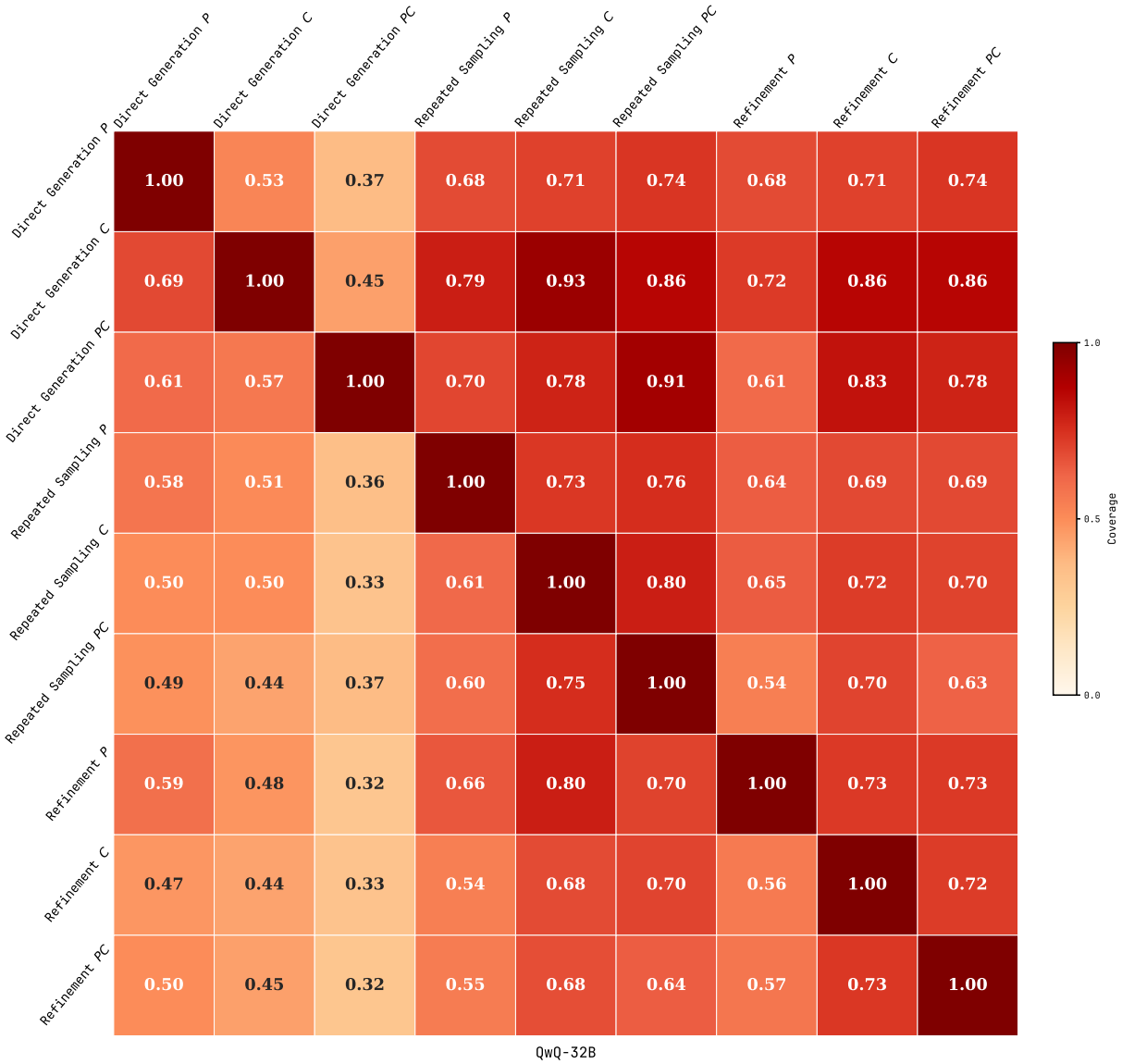


Figure 21: Asymmetric relative coverage matrix of nine ARC solvers using QwQ-32B, showing the proportion of problems whose test instances are solved by the row solver that are also solved by the column solver. *P* denotes the solution plan; *C* and *PC* refer to standalone and planning-aided code generation, respectively.

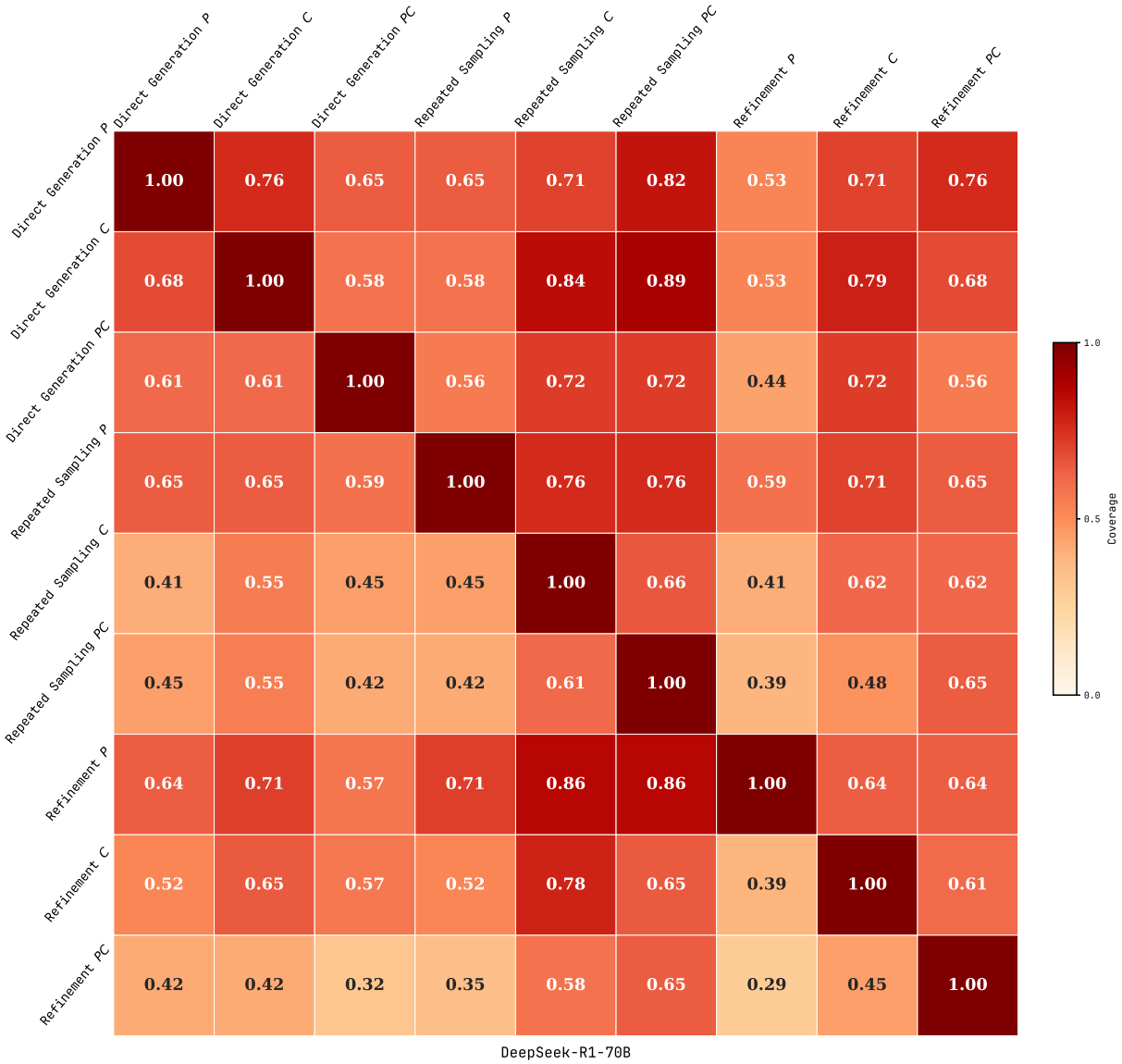


Figure 22: Asymmetric relative coverage matrix of nine ARC solvers using DeepSeek-R1-70B, showing the proportion of problems whose test instances are solved by the row solver that are also solved by the column solver. *P* denotes the solution plan; *C* and *PC* refer to standalone and planning-aided code generation, respectively.

L Prompts for LLMs

We include all prompts used by KAAR and nine ARC solvers described in Section 3. We adopt a bash-like notation for input arguments within the prompts, such as *${test_inputs}* denotes the test input 2D matrices. A brief description of the prompts used for each solver is provided below.

- **Direct generation with solution plan**: Prompt 1 describes how to generate the solution plan, and Prompt 2 uses the generated plan to produce the output images.
- **Direct generation with standalone code**: Prompt 3 describes how to generate the code to produce the output images.
- **Direct generation with planning-aided code**: It first generates a solution plan using Prompt 1, then uses Prompt 4 to produce code based on the generated plan.
- **Repeated sampling with solution plan**: It can be regarded as an iterative version of direct generation with solution plan, and thus also uses Prompts 1 and 2.
- **Repeated sampling with standalone code**: It can be regarded as an iterative version of direct generation with standalone code, and thus also uses Prompt 3.
- **Repeated sampling with planning-aided code**: It can be regarded as an iterative version of direct generation with planning-aided code, and thus also uses Prompts 1 and 4.
- **Refinement with solution plan**: Prompt 5 describes the process of refining the generated solution plan with the validation samples. It uses Prompts 1 and 2 to generate the initial plan and the result image.
- **Refinement with the standalone code**: Prompt 6 describes the process of refining the generated code with the validation samples. It uses Prompt 3 to produce the initial code solution.
- **Refinement with the planning-aided code**: Prompt 7 describes the process of refining the generated plan and code with the validation samples. It use Prompts 1 and 4 to generate the initial plan and produce the initial code guided by the plan, respectively.
- **KAAR**: Prompt 8 describes the augmentation of objectness priors. Prompts 9 and 10 introduce the augmentation of geometry and topology priors, encoded as component attributes and relations, respectively. Prompt 11 outlines the augmentation of numbers and counting priors. Prompts 12 and 13 describe action selection and target component identification in the process of augmenting goal-directedness priors. For prompts implementing each action's implementation details, please refer to our code.

Prompt 1: Direct generation with solution plan - solution plan generation.

```
================================ System ================================
You are an expert in analyzing grid-based image processing tasks. Your objective is to
derive a text transformation plan (not Python code) from each given input-output image
pair (both represented as 2D matrices), and then apply this plan to generate output image
(s), represented as a 2D matrix, based on the given test input image(s) (2D matrix).
Ensure that the derived plan generalizes across different cases while preserving
consistency with the observed transformations.

================================= User =================================
The input data consists of a few pairs of input and output images, where the left image
in each pair represents the input, and the right image represents the corresponding
output. Each image can be represented as a 2D matrix: ${matrix}

Please note that each number in the matrix corresponds to a pixel, and its value
represents the color.
```

```
Derive a text transformation plan (not Python code) that maps each given input image (2D
matrix) to its corresponding output image (2D matrix). Ensure that the plan generalizes
across different cases and the test input image(s) (2D matrix) while maintaining
consistency with the observed transformations.

The test input image(s): ${test_inputs}
```

Prompt 2: Direct generation with solution plan - output image(s) generation from the plan.

```
================================ System ================================
You are an expert in analyzing grid-based image processing tasks. Your objective is to
generate output image(s), represented as a 2D matrix, based on the given input images (2D
 matrix) and a derived text transformation plan.

================================= User =================================
Please generate the output image(s) as a 2D matrix (not Python code) based on the given
input image(s) (2D matrix) and the text transformation plan. Output only the test output
image(s) in 2D matrix format (not Python code). For each test input image, start with [
Start Output Image] and end with [End Output Image].

For example, if there is one test input image, the output image should be:
[Start Output Image]
[[0,0,0], [0,0,0], [0,0,0]]
[End Output Image]

If there are multiple (2) test input images, the the output images should be outputted as
:
[Start Output Image]
[[0,0,0], [0,0,0], [0,0,0]]
[End Output Image]
[Start Output Image]
[[1,1,1], [1,1,1], [1,1,1]]
[End Output Image]

The test input image(s): ${test_inputs}
```

Prompt 3: Direct generation with standalone code.

```
================================ System ================================
You are an expert in analyzing grid-based image processing tasks. Your goal is to
generate Python code that produces output image(s), represented as a 2D matrix, based on
the given input image(s) (2D matrix).

================================= User =================================
The input data consists of a few pairs of input and output images, where the left image
in each pair represents the input and the right image represents the corresponding output
.
Each image can be represented as a 2D matrix: ${matrix}
The test input image(s): ${test_inputs}

Please note that each number in the matrix corresponds to a pixel, and its value
represents the color.

Generate a Python script to map each input image (2D matrix) to the corresponding output
image (2D matrix).
Ensure that the Python script generalizes across different cases and test input image(s)
while maintaining consistency with the observed input-output image pairs.
Please output the Python program, starting with [Start Program] and ending with [End
Program].
Include an assert statement with the function signature to verify that the generated
output matches the expected result, starting with [Assert Statement].
Use placeholders like input_image and output_image for the variables representing the
input and output images.

For example:
[Start Program]
def generate_output_image(input_image):
    rows = len(input_image)
    cols = len(input_image[0])
```

```
    def dfs(r, c):
        """Depth-first search to mark all 4-connected '1's to '2's."""
        if r < 0 or r >= rows or c < 0 or c >= cols or input_image[r][c] != 1:
            return
        # Change the current component from 1 to 2
        input_image[r][c] = 2
        # Explore neighbors (up, down, left, right)
        dfs(r - 1, c) # Up
        dfs(r + 1, c) # Down
        dfs(r, c - 1) # Left
        dfs(r, c + 1) # Right

    # Traverse the image to find all components with '1'
    for r in range(rows):
        for c in range(cols):
            if input_image[r][c] == 1:
                dfs(r, c)
    return input_image
[End Program]
[Assert Statement]
assert generate_output_image(input_image) == output_image

Please note, the assert statement should strictly follow the provided format, and the
output image should be represented in list format!
Please note, the script should not include an if __name__ == "__main__": block.
```

Prompt 4: Direct generation with planning-aided code - code generation based on the generated plan.

```
================================ System ================================
You are an expert in analyzing grid-based image processing tasks. Your goal is to
generate Python code that produces output image(s) represented as a 2D matrix, based on
the given input image(s) (2D matrix). This code should be generated using a text
transformation plan inferred from a set of input-output image pairs (both represented as
2D matrices).

================================= User =================================
Generate a Python script based on your text transformation plan to map the input image (2
D matrix) to the output image (2D matrix). Please output the Python program, starting
with [Start Program] and ending with [End Program]. Include an assert statement with the
function signature to verify that the generated output matches the expected result,
starting with [Assert Statement]. Use placeholders like input_image and output_image for
the variables representing the input and output images.

For example:
[Start Program]
def generate_output_image(input_image):
    rows = len(input_image)
    cols = len(input_image[0])

    def dfs(r, c):
        """Depth-first search to mark all 4-connected '1's to '2's."""
        if r < 0 or r >= rows or c < 0 or c >= cols or input_image[r][c] != 1:
            return
        # Change the current component from 1 to 2
        input_image[r][c] = 2
        # Explore neighbors (up, down, left, right)
        dfs(r - 1, c) # Up
        dfs(r + 1, c) # Down
        dfs(r, c - 1) # Left
        dfs(r, c + 1) # Right

    # Traverse the image to find all components with '1'
    for r in range(rows):
        for c in range(cols):
            if input_image[r][c] == 1:
                dfs(r, c)
    return input_image

[End Program]
```

```
[Assert Statement]
assert generate_output_image(input_image) == output_image

Please note, the assert statement should strictly follow the provided format, and the
output image should be represented in list format!
Please note, the script should not include an if __name__ == "__main__": block.
```

Prompt 5: Refinement with solution plan - plan refinement.

```
================================ System ================================
As an expert in analyzing grid-based image processing tasks, your objective is to refine
your solution plan based on the provided feedback.

================================= User =================================
The problem description:
[start problem description]
The input data consists of a few pairs of input and output images, where the left image
in each pair represents the input, and the right image represents the corresponding
output. Each image can be represented as a 2D matrix: ${matrix}
Please note that each number in the matrix corresponds to a pixel, and its value
represents the color.
[end problem description]

The INCORRECT text transformation plan fails to solve some example training input and
output pairs in the above problem!

[start incorrect transformation plan]
${plan}
[end incorrect transformation plan]

The incorrect output(s) generated by the incorrect plan:
[start incorrect output]
${incorrect_output}
[end incorrect output]

The generated correct output(s):
[start correct output]
${correct_output}
[end correct output]

Please analyze the incorrect reasoning step-by-step, and then generate the revised
correct transformation plan (text only), starting with [Start Revised Transformation Plan
] and ending with [End Revised Transformation Plan]. Ensure that the revised
transformation plan generalizes across different cases and the test input image(s), while
 maintaining consistency with the observed transformations.
```

Prompt 6: Refinement with standalone code - code refinement.

```
================================ System ================================
As an expert in analyzing grid-based image processing tasks, your objective is to refine
your program based on the provided feedback.

================================= User =================================
The problem description:
[start problem description]
The input data consists of a few pairs of input and output images, where the left image
in each pair represents the input, and the right image represents the corresponding
output. Each image can be represented as a 2D matrix: ${matrix}
Please note that each number in the matrix corresponds to a pixel, and its value
represents the color.
[end problem description]

The generated incorrect program fails to solve some example training input and output
pairs in the above problem!

[start incorrect program]
${code}
[end incorrect program]
```

```
The incorrect output(s) generated by the incorrect program:
[start incorrect output]
${incorrect_output}
[end incorrect output]

The generated correct output(s):
[start correct output]
${correct_output}
[end correct output]

Please analyze the incorrect reasoning step-by-step, and then generate the revised
program (Python program only), starting with [Start Revised Program] and ending with [End
 Revised Program]. Ensure that the revised program generalizes across different cases and
 the test input image(s), while maintaining consistency with the observed input and
output image pairs.

Please include an assert statement with the function signature to verify that the
generated output matches the expected result, starting with [Assert Statement]. Use
placeholders like input_image and output_image for the variables representing the input
and output images.

For example:
[Start Revised Program]
def generate_output_image(input_image):
    rows = len(input_image)
    cols = len(input_image[0])

    def dfs(r, c):
        """Depth-first search to mark all 4-connected '1's to '2's."""
        if r < 0 or r >= rows or c < 0 or c >= cols or input_image[r][c] != 1:
            return
        # Change the current component from 1 to 2
        input_image[r][c] = 2
        # Explore neighbors (up, down, left, right)
        dfs(r - 1, c) # Up
        dfs(r + 1, c) # Down
        dfs(r, c - 1) # Left
        dfs(r, c + 1) # Right

    # Traverse the image to find all components with '1'
    for r in range(rows):
        for c in range(cols):
            if input_image[r][c] == 1:
                dfs(r, c)
    return input_image
[End Revised Program]
[Assert Statement]
assert generate_output_image(input_image) == output_image

Please note, the assert statement should strictly follow the provided format, and the
output image should be represented in list format!
Please note, the script should not include an if __name__ == "__main__": block.
```

Prompt 7: Refinement with planning-aided code - refinement on both generated plan and code.

```
================================ System ================================
As an expert in analyzing grid-based image processing tasks, your objective is to refine
your transformation plan and program based on the provided feedback.

================================= User =================================
The problem description:
[start problem description]
The input data consists of a few pairs of input and output images, where the left image
in each pair represents the input, and the right image represents the corresponding
output. Each image can be represented as a 2D matrix: ${matrix}
Please note that each number in the matrix corresponds to a pixel, and its value
represents the color.
[end problem description]
```

```
The generated incorrect transformation plan and program fail to solve some example
training input and output pairs in the above problem!

[start incorrect transformation plan]
${plan}
[end incorrect transformation plan]

[start incorrect program]
${code}
[end incorrect program]

The incorrect output(s) generated by the incorrect transformation plan and program:
[start incorrect output]
${incorrect_output}
[end incorrect output]

The generated correct output(s):
[start correct output]
${correct_output}
[end correct output]

Please analyze the incorrect reasoning step-by-step, and then generate the revised
transformation plan (text only) and program (Python program only).

For the revised transformation plan, start with [Start Revised Transformation Plan] and
end with [End Revised Transformation Plan]. Ensure that the revised transformation plan
generalizes across different cases and the test input image(s), while maintaining
consistency with the observed transformations.

For the revised Python program, start with [Start Revised Program] and end with [End
Revised Program]. Ensure that the revised program generalizes across different cases and
the test input image(s), while maintaining consistency with the observed input and output
 image pairs.

For the revised Python program, please include an assert statement with the function
signature to verify that the generated output matches the expected result, starting with
[Assert Statement]. Use placeholders like input_image and output_image for the variables
representing the input and output images.

For example:
[Start Revised Program]

def generate_output_image(input_image):
    rows = len(input_image)
    cols = len(input_image[0])

    def dfs(r, c):
        """Depth-first search to mark all 4-connected '1's to '2's."""
        if r < 0 or r >= rows or c < 0 or c >= cols or input_image[r][c] != 1:
            return
        # Change the current component from 1 to 2
        input_image[r][c] = 2
        # Explore neighbors (up, down, left, right)
        dfs(r - 1, c) # Up
        dfs(r + 1, c) # Down
        dfs(r, c - 1) # Left
        dfs(r, c + 1) # Right

    # Traverse the image to find all components with '1'
    for r in range(rows):
        for c in range(cols):
            if input_image[r][c] == 1:
                dfs(r, c)
    return input_image
[End Revised Program]
[Assert Statement]
assert generate_output_image(input_image) == output_image

Please note, the assert statement should strictly follow the provided format, and the
output image should be represented in list format!
```

```
Please note, the script should not include an if __name__ == "__main__": block.
```

Prompt 8: Objectness priors augmentation

```
================================ System ================================
You are an expert in grid-based image analysis.

================================= User =================================
The training instances consist of several pairs of input and output images, where the
left image in each pair represents the input and the right image represents the
corresponding output.
Please note that the test instance(s) only contains input image(s).
Each image is represented as a 2D matrix:
${matrix}

Please note that each number in the matrix corresponds to a pixel and its value
represents the color.

We treat the color represented by the number {background_color} as the background color.
${abstraction_rule}
The components in each input and output image pair are as follows:
${component_description}
```

Prompt 9: Geometry and topology priors augmentation - component attributes

```
================================ System ================================
You are an expert in geometry and topology analysis. Below is a summary of component
attributes, including:
Size (Width and Height); Color; Shape; Symmetry; Bounding Box; Hole Count; Nearest
Boundary.
================================= User =================================
${geometry_and_topology_priors_attributes}$
```

Prompt 10: Geometry and topology priors augmentation - component relations

```
================================ System ================================
You are an expert in geometry and topology analysis, Below is a summary of component
relations, including:
 Different/Identical with other components; Inclusive; Touching or or not touching with
 other component; Spatial Relations,

================================= User =================================
${geometry_and_topology_priors_relations}$
```

Prompt 11: Numbers and counting priors augmentation

```
================================ System ================================
You are an expert in numbers and counting analysis. Below is a summary of component
statistics, including:
Symmetry numerical summary; Size numerical summary; Color numerical summary; Shape
numerical summary; Hole counting summary.
================================= User =================================
${numbers_and_couting_priors}$
```

Prompt 12: Goal-directedness priors augmentation - action selection

```
================================ System ================================
You are an expert in analyzing and categorizing grid-based image tasks.
================================= User =================================

Please determine which category or categories this task belongs to. Please select from
the following:
1. color change: color change involves modifying the value of a component, and the
component size and position always does not change.
2. movement: movement involves shifting the position of a component to a new location
within the image, and the component size always does not change.
3. extension: extending involves expanding the boundaries of a component to increase its
size or reach within the image, and the component size always changes.
```

```
4. completing: completing an image involves filling in missing or incomplete parts of a
component to achieve a coherent and fully formed image.
5. resizing: resizing involves altering the dimensions of a component by expanding or
shrinking its size within the image.
6. selecting: selecting involves identifying and isolating a specific component within
the image as the output component, and the component size and color always does not
change.
7. copying: copying involves duplicating a component and either placing the duplicate in
a new location or replacing the existing component within the image.
8. flipping: flipping involves mirroring a component along a specified axis to reverse
its orientation within the image.
9. rotation: rotation involves turning a component around a fixed point or center by a
specified angle within the image.
10. cropping: cropping involves cutting out a specific portion of a component.

Please select the best suitable one or multiple categories from the provided list that
best describe the task.

Format your response by starting with [start category] and ending with [end category],
numbering each category selected.

For example, if the task belongs only to "color change", your response should be:
[start category]
1. color chang
[end category]

If the task belongs to both "selecting" and "extension", your response should be:
[start category]
1. selecting
2. extension
[end category]
```

Prompt 13: Goal-directedness priors augmentation - target component idetification

```
================================ System ================================
You are an expert in analyzing grid-based image tasks, specifically in ${action}
components.
================================= User =================================
If this task involves ${action}:
1. Begin by identifying WHICH COMPONENTS are to be ${action} in all input images (
training and test pairs).
 - Refer to these components as TARGET components (e.g., component 1 in the first input
 image, component 2 and component 3 in the second input image, etc.).
 - List ALL target components in each training and test input image.
 - For EACH target component, provide:
    - Attribute Analysis result
    - Relation analysis result
    - Numerical analysis result

2. Determine the CONDITIONS used to select these TARGET components for ${action} from
each training and test input image.
 - These conditions must be based on common priorities across all targeted components and
  must differ from the unselected components.
 - For example: the size of all target components might be equal to 3 while the size of
 the unselected components is not 3.
 2.1. Analyze whether these conditions are EMPTY or not.
 2.2. Evaluate if these conditions are derived from attribute analysis, including:
  2.2.1. Color
  2.2.2. Size
  2.2.3. Shape
  2.2.4. Width
  2.2.5. Height
  2.2.6. The number of holes
  2.2.7. Bounding box
  2.2.8. Symmetry
  2.2.9. Nearest boundary
 2.3. Evaluate if these conditions are derived from relation analysis, including:
  2.3.1. Relative position with other components
  2.3.2. Touching with other components
  2.3.3. Whether they differ from or are identical with other components
```

```
 2.3.4. Enclosure of other components
2.4. Evaluate if these conditions are derived from numerical analysis, including:
 2.4.1. Symmetry numerical analysis
 2.4.2. Size numerical analysis
 2.4.3. Color numerical analysis
 2.4.4. Shape numerical analysis
 2.4.5. Hole counting analysis

You must evaluate each condition ONE by ONE and determine the best conditions.
Note:
- The conditions MUST work for ALL training and test input and output image pairs.
- Conditions CANNOT come from the output images!
- A condition can be EMPTY.
- If a condition is based on numerical features (e.g., size (width and height), or the
number of holes), you may use the operators =, <, >, >=, or <=.
- For cropping or selecting tasks, consider using a bounding box to extract each
component.
```