Combining Symbolic Constraints and Large Language Models: An Application to the Abstraction and Reasoning Corpus

Lukas Kinder[1 [0009−0008−8275−1284]], Quentin Cappart[2,3,4 [0000−0002−8742−0774]], and Tobias Käfer[1 [0000−0003−0576−7457]]

[1] Institute AIFB, Karlsruhe Institute of Technology (KIT), Germany
lukas.kinder@kit.edu, tobias.kaefer@kit.edu
[2] Polytechnique Montréal, Montreal, Canada
quentin.cappart@polymtl.ca
[3] Quebec Artificial Intelligence Institute (MILA), Montreal, Canada
[4] UCLouvain, Louvain-la-Neuve, Belgium

Abstract. The Abstraction and Reasoning Corpus (ARC) is a challenging benchmark designed to test reasoning over image-to-image transformations. Although large language models (LLMs) have recently shown strong performance on ARC, they still struggle with symbolic concepts such as counting, symmetry, and structural consistency. The recently proposed ARChitect pipeline is currently the leading open-source approach; it combines a fine-tuned LLM with an augmentation-based search and candidate-selection scheme. In this work, we improve ARChitect by incorporating a set of lightweight symbolic constraints that restrict the search space to outputs that follow general structural properties of ARC tasks. Integrating these constraints into a backtracking search procedure yields consistent improvements in both accuracy and inference time. Our results demonstrate that combining symbolic constraints with LLM-guided search provides a simple and effective form of neuro-symbolic reasoning, improving performance across different models.

Keywords: Generative models · Constraints · Neuro-symbolic

1 Introduction

Despite recent progress, systems based on large language models (LLMs) still struggle with tasks that demand abstract reasoning and broad generalization, areas in which humans excel. This limitation is evident in the *Abstraction and Reasoning Corpus* (ARC) benchmark [5], which was developed to assess how close we are in achieving artificial general intelligence. The dataset consists of hand-crafted visual reasoning puzzles, each requiring the solver to infer a unique logical transformation between small input and output images. While humans can typically solve most of these problems, machine performance remains substantially lower. At the time of writing, both benchmarks remain unsolved. A

600,000 USD prize was offered for surpassing 85% accuracy under a restricted compute budget, but it remains unclaimed [5].

Since the release of the initial ARC benchmark in 2019, researchers have investigated a broad spectrum of solution strategies. Early efforts were largely driven by search procedures over domain-specific languages [11,20,15]. More recently, approaches leveraging pretrained large language models have become increasingly prominent. These models are typically used for transduction, i.e. directly predicting the test output from the provided examples [14,7]. They can also be used for induction, where the model generates a latent transition function, often expressed as a Python program, that is subsequently applied to the test input to obtain the final solution [16,14,8,13]. There have also been propositions for alternative architectures of neural networks to solve the ARC [6,12].

The ARChitect pipeline is a transductive LLM approach that leads the open-source competition, achieving high accuracy under relatively tight compute constraints [1]. It won the ARC Prize 2024 with an accuracy of 53.5%. ARChitect combines multi-stage fine-tuning, custom tokenization, test-time training and an augmentation scheme for finding candidate solutions [7]. Despite these promising results, performance remains far below human level, and LLMs often fail on tasks requiring precise symbolic reasoning, for example, counting, symmetry detection, or enforcing structural characteristics [3,9,18]. This motivates our work: we extend the ARChitect framework by integrating symbolic constraints during inference, thereby combining the generative flexibility of LLMs with lightweight symbolic reasoning.

Our idea builds on the assumption that if a specific characteristic is observed in the input–output transformations of the training examples, the same characteristic is likely to hold for the test transformation of that problem. We can then use this knowledge to constrain the LLM to generate only candidate solutions that satisfy these characteristics. Frequent examples include rules such as "*blue pixels remain unchanged*" or "*the output is horizontally mirrored*". We propose integrating these constraints at inference time via a backtracking search. During LLM-based decoding, the constraints act as filters, preventing the model from producing token sequences that either violate the inferred symbolic rules or lead to low-probability outputs. This approach not only increases the likelihood of generating correct solutions, but also reduces runtime by pruning unpromising partial generations early in the search. Moreover, checking symbolic constraints is substantially faster than LLM token generation, making constraint-based pruning an efficient mechanism for guiding inference. Figure 1 shows an example of an ARC-like problem in which the LLM is prevented from generating candidates that would violate the inferred characteristic that the output image must have the same color histogram as the input image. Based on this context, the contributions of this paper are as follows:

1. We present a collection of 12 symbolic constraints that can be applied to ARC to reduce the solution space and to detect violations while constructing a solution.

[5] https://arcprize.org/leaderboard

2. We demonstrate that these constraints improve accuracy and reduce runtime when solving ARC tasks, across LLMs of various sizes (a $1.5B$ version of *DeepSeek-R1*, a $3B$ version of *Llama-3.2*, and an $8B$ version of *Mistral-NeMo-Minitron*).
3. Beyond these specific results, this study demonstrates how soft symbolic constraints can be effectively incorporated into LLM-based generation.
4. Finally, we release our implementation to foster research in this area [6].

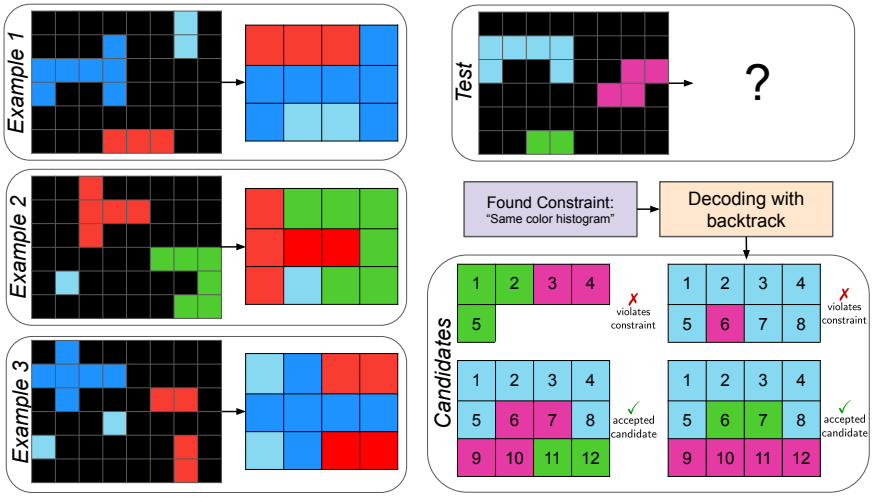

Fig. 1. An example in which a constraint is inferred stating that the input and output images share the same color histogram. This constraint can be used to prune candidates during the LLM's decoding with a backtracking search, where the output image is constructed row by row. For instance, the first candidate is rejected as soon as a third green pixel is generated, because the test input image contains only two green pixels. The numbers on the pixels indicate the order in which they are generated.

2 Overview of The ARChitect pipeline

The ARChitect pipeline [7] combines fine-tuning, test-time training, augmentation, depth-first search for candidate generation, and multi-augmentation scoring to achieve state-of-the-art performance among open-source ARC systems. Its implementation is highly optimized to minimize both runtime and memory consumption. An overview of the pipeline is shown in Figure 2. This section describes the core mechanisms of ARChitect.

[6] `https://github.com/LukasKinder/Product-of-Experts-ARC-Constraints`

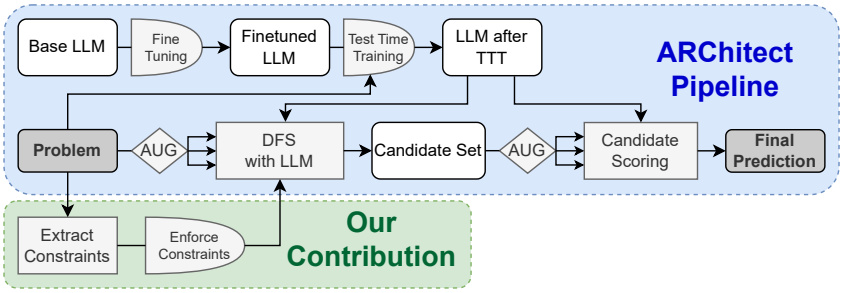

Fig. 2. The ARChitect pipeline highlighted in blue together with our extension, highlighted in green. The rhombus-shape labeled (AUG) indicates that a problem is augmented multiple times.

Fine-tuning. ARChitect is initially and exclusively fine-tuned on the RE-ARC dataset [10], which provides procedurally generated instances of all ARC-1 training tasks. The authors also modify the tokenizer so that each pixel corresponds to exactly one token.

Test-time training (TTT). A key component of the pipeline is test-time training, originally introduced by Sun et al. [19] and later applied to ARC by Akyürek et al. [2]. During evaluation, the LLM is briefly fine-tuned on the input–output examples of the problem being solved. This significantly boosts performance by adapting the model to the underlying pattern of the specific task.

Data augmentation (AUG). The ARC tasks are invariant under rotations, reflections, and color permutations. Formally, if a problem p is augmented to p_{aug} using only these transformations, we have:

$$\text{Solution}(p) = \text{ReverseAugmentation}\big(\text{Solution}(p_{aug})\big). \tag{1}$$

Augmentations are used to increase the number of examples during TTT and are also applied later during search and when scoring candidates.

Depth-first search (DFS). Each problem is augmented 16 times and for each augmentation the LLM generates all solutions whose cumulative token probability exceeds a predefined threshold Θ. Concretely, the decoder prunes any partial sequence whose product of token probabilities falls below Θ, yielding a depth-first search tree over possible outputs.

Candidate selection. Each unique candidate-solution generated during search is evaluated under 16 augmentations. A candidate's score is computed as its average token probability across these augmentations. Because ARC allows two guesses per task, ARChitect submits the two best-scoring candidates. A task is marked correct if either guess matches the ground truth.

3 Backtracking Search for LLM Decoding

Table 1. Summary of symbolic constraints observed in the 400 instances of the ARC-1 training set. **Holds** indicates the number of times the constraint is satisfied in all input-output examples of a problem. **Exceptions** indicates the number of times the constraint did hold for the examples but did not for the test-transition of a problem. **Evaluation** indicates if the constraint can be evaluated *online* (during generation of the output) or only *offline* (after the complete output image was generated). While we provide only a brief description of each constraint in the **Description** column, the full implementation of the constraints as symbolic checkers is available in our Git repository. The official ARC-identifiers in the **Examples** column link to visualizations of the problem.

Evaluation	#	Description	Holds	Exceptions	Examples
offline	**1**	Output image differs from input	355	0	332efdb3, ac2e8ecf
	2	Counts of the background color decreases	220	2	332efdb3, 3f23242b
	3	All pixels have at least one neighbor that is not background	134	4	3f23242b, 776ffc46
	4	Output is horizontally mirrored	24	0	332efdb3, 9ddd00f0
online	**5**	Only one color changes	141	0	332efdb3, 3f23242b
	6	Color-histogram the same (pixels are just repositioned)	23	0	5ffb2104, ac2e8ecf
	7	Color-histogram preserved except for one color	101	0	759f3fd3, 14754a24
	8	Count of a specific color remains unchanged	114	1	3f23242b, aa300dc3
	9	Outputs have a fixed set of colors	119	0	332efdb3, 3f23242b
	10	Counts of a specific color changes by fixed amount	27	0	b7fb29bc, 19bb5feb
	11	A specific color does not change	126	0	3f23242b, aa300dc3
	12	Output is vertically mirrored	21	1	e21a174a, 9ddd00f0

Our research hypothesis is that incorporating a backtracking search into the ARChitect pipeline can improve performance (see the bottom part of Figure 2). The main idea is to terminate a partial generation as soon as it violates an inferred constraint. To this end, we introduce twelve symbolic constraints, listed in Table 1.

We assume that a constraint holds for a given problem if all of its input–output examples satisfy it. Most constraints concern color histograms or spatial symmetries. Some apply to nearly all ARC tasks, whereas others hold only for a small subset. For instance, constraints 5 and 11 apply only to problems in which the input and output grids have the same size. In this setting, a "*change in color*" is defined by a position-wise comparison between the input and output images. Figure 3 illustrates how the backtracking search is carried out on a toy example.

We manually derived these constraints by analyzing common mistakes made by LLMs in the ARC training set. Initially, we considered a broader set of constraints, which we later narrowed down to 12 by selecting only those that (A) apply to a sufficient number of problems and (B) have few exceptions in the ARC-1-training set. The original, broader set of constraints is still available in our Git repository.

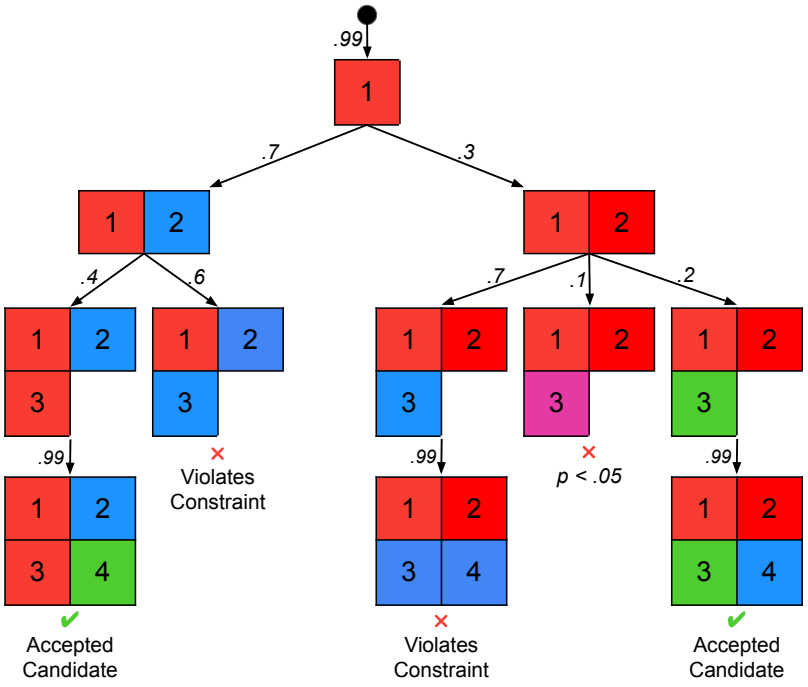

Fig. 3. Illustration of our backtracking search with the constraint "*There should be only one blue pixel*" on a 2×2 grid. Only branches are explored that **(1)** have a cumulative probability (p) above the threshold of $\Theta = 0.05$, and **(2)** do not violate the constraint. The numbers next to the arrows indicate the probability of generating a given pixel.

Online and offline checking. A constraint can be evaluated online if a violation can be detected before the entire output image has been generated. Online evaluation enables pruning during search and thereby reduces unnecessary computation. For example, the first constraint ("*output image is different from input*") can only be evaluated offline, since a violation can be detected only after the full output has been produced. Likewise, any constraint that depends on the background color requires access to the complete output grid, since we define the background color as the most frequent color in the image. Horizontal symmetry also cannot be ruled out early, because later rows may restore symmetry.

Algorithm 1 Backtracking with symbolic constraints

```
1: function BacktrackingLLM(prefix)
2:     S ← ∅
3:     for each token t where p(llmGenerate(prefix + t)) > Θ do
4:         prefix′ ← prefix + t
5:         if checkConstraints(prefix′) is true then
6:             if t = ⟨eos⟩ then
7:                 S ← S ∪ {prefix′}
8:             else
9:                 S ← S ∪ BacktrackingLLM(prefix′)
10:    return S
11: end function
```

In contrast, vertical symmetry can be checked online: each generated row must be a palindrome, and a single non-palindromic row is sufficient to conclusively violate the constraint.

Dealing with exceptions. Because constraints are inferred from the examples, *exceptions* may occur: cases where a constraint does not hold for the test transition, despite holding in all examples. In such situations, enforcing the constraint would make the true solution unreachable. We mitigate this issue by disabling a constraint in the rare case in which *all* generated candidates violate it, which suggests that the constraint is not valid. An example of such an exception is shown in the Appendix (Figure 4).

Full-fledged backtracking search. We integrate the symbolic constraints directly into the DFS procedure of ARChitect, yielding a backtracking search. Whenever a newly generated token causes the partial output to violate any active constraint, we prune the corresponding branch and do not explore it further. The adapted pseudo-code is provided in Algorithm 1. It implements a standard backtracking search: we expand only those branches whose cumulative sequence probability exceeds a threshold Θ (Line 3) and whose partial outputs satisfy all active constraints (Line 5). A branch terminates successfully when an end-of-sequence token (eos) is generated, at which point the resulting solution is added to the pool S.

4 Experimental Results

The goal of the experiments is to assess whether our improvements to ARChitect are beneficial regardless of the underlying LLM. To this end, we conducted experiments with three base models: (1) a 1.5B-parameter version of R1 (deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B), (2) a 3B-parameter version of Llama-3 (meta-llama/Llama-3.2-3B), and (3) an 8B-parameter version of NeMo (nvidia/Mistral-NeMo-Minitron-8B-Base). All models were finetuned using the

RE-ARC dataset [10] following the procedure described in the ARChitect paper [7]. While the algorithm is theoretically deterministic, slight fluctuations in runtime and accuracy were caused in practice by our GPUs, which is why we report the average over 5 runs (standard deviations were always below 0.5%). We focus on the ARC-1 dataset.

Table 2. Results for different models on the public ARC-1 evaluation dataset (400 tasks for testing), comparing performance *with* and *without* using symbolic constraints.

Model (Θ)	**Accuracy (%)**		**Time (hh:mm)**	
	without Constraints	with Constraints	without Constraints	with Constraints
R1 (0.2)	45.62	46.75 (+1.13)	14 : 21	14 : 01 (-2.4%)
R1 (0.09)	48.93	50.57 (+1.65)	23 : 56	22 : 24 (-6.4%)
R1 (0.05)	49.80	50.55 (+0.75)	36 : 21	32 : 58 (-9.3%)
Llama (0.2)	58.98	59.05 (+0.07)	16 : 30	16 : 18 (-1.2%)
Llama (0.09)	61.00	62.05 (+1.05)	26 : 07	25 : 42 (-1.6%)
Llama (0.05)	61.88	62.67 (+0.80)	38 : 59	37 : 07 (-4.8%)
NeMo (0.2)	69.98	69.98 (+0.00)	24 : 17	23 : 31 (-3.1%)
NeMo (0.09)	70.73	71.25 (+0.52)	34 : 17	33 : 43 (-1.7%)
NeMo (0.05)	71.58	71.62 (+0.05)	49 : 30	47 : 17 (-4.5%)

Table 3. The performance of R1 with $\Theta = 0.9$ for subsets of the ARC-1 public testing data in which specific constraints hold. The #**Task** column indicates the size of the subset.

Constraint Type	#**Tasks**	**Accuracy (%)**		**Time (hh:mm)**	
		without Constraints	with Constraints	without Constraints	with Constraints
# **1:** Output is different to input.	353	49.93	51.59 (+1.66)	20 : 05	18 : 47 (-6.5%)
# **2:** Background color decreases.	204	53.43	55.39 (+1.96)	9 : 58	9 : 31 (-4.5%)
# **3:** No lonely pixels in output.	95	53.16	55.55 (+2.40)	4 : 37	4 : 16 (-7.7%)
# **4:** Output is horizontally mirrored.	14	57.14	57.73 (+0.59)	0 : 37	0 : 31 (-15.8%)
# **5:** Only one color changes.	137	45.50	44.98 (-0.52)	10 : 18	9 : 25 (-8.5%)
# **6:** Color histogram stays the same.	26	25.64	32.69 (+7.05)	1 : 37	1 : 30 (-6.5%)
# **7:** Color histogram stays the same except one.	90	45.69	52.22 (+6.53)	5 : 31	5 : 04 (-8.2%)
# **8:** Count of a specific color does not change.	154	40.18	43.73 (+3.55)	10 : 25	9 : 31 (-8.6%)
# **9:** Output has specific set of colors.	124	45.83	47.58 (+1.75)	7 : 21	6 : 54 (-6.0%)
# **10:** Count of a color changes by a set amount.	33	33.33	34.85 (+1.52)	1 : 40	1 : 34 (-5.6%)
# **11:** A specific color does not change.	163	40.80	41.62 (+0.82)	11 : 42	10 : 55 (-6.7%)
# **12:** Output is vertically mirrored.	12	58.33	58.33 (+0.00)	0 : 25	0 : 23 (-8.3%)

Main results. Table 2 reports accuracy and runtime for all models, with and without symbolic constraints, across different probability thresholds Θ for cutting the search. As expected, the choice of base model has the largest effect on

performance. Across all settings, adding constraints consistently improves accuracy by about 1% and reduces runtime by roughly 4% on average. The time required to infer and check constraints during decoding is negligible, never exceeding one second per task.

Analysis: impact of each individual constraint. We additionally performed a study in which we evaluated R1 (0.09) model only on the subset of tasks for which a given constraint holds (Table 3). While the accuracy gains were not consistent across all subsets, we observed a consistent reduction in inference time. Constraint 6 "*Only one color changes*" resulted in a small decrease in accuracy duo to an exception in the ARC-1 test-set [7]. Interestingly, the benefit of symbolic constraints was especially pronounced for the constraints involving counting ("*6: Color histogram stays the same*","*7:Color histogram stays the same except one*" and "*8: Count of a specific color changes by a specific amount*"), where early pruning substantially reduces the explored search space. This result suggests that identifying relevant constraints is a key consideration in our approach.

5 Conclusion

The Abstraction and Reasoning Corpus is a challenging benchmark designed to test reasoning over image-to-image transformations, and it remains unsolved to date. We extend the ARChitect pipeline by embedding a set of symbolic constraints into a backtracking search procedure. Across three underlying LLMs, these constraints consistently improve accuracy and reduce runtime. Because faster inference can be traded for deeper search, the time savings can directly translate into higher accuracy. This result illustrates that simple symbolic constraints can effectively complement LLM-based reasoning, particularly for tasks in which partial solutions can be validated incrementally.

While the benefit of using the constraints is not very large, they are lightweight, model-agnostic, and easy to evaluate, making them suitable for integration into other ARC solvers. More generally, our results illustrate that simple symbolic checks can effectively complement LLM-based reasoning, especially in tasks where partial solutions can be validated incrementally.

For this initial proof of concept, we did not incorporate a constraint programming (CP) solver into the pipeline; instead, we enforced constraint satisfaction using ad hoc checkers. Inspired by recent works combining CP with LLMs for related tasks [17], we plan to leverage the full potential of CP solvers, such as propagation to prune the search space more aggressively and advanced search strategies, to further improve efficiency. In addition, we intend to explore active constraint acquisition techniques [4] to automate the discovery of symbolic constraints.

[7] Note that Table 1 is for the ARC-1 training-set.

References

1. ARC Prize 2024 Leaderboard. https://www.kaggle.com/competitions/arc-prize-2024/leaderboard (2024)
2. Akyürek, E., Damani, M., Zweiger, A., Qiu, L., Guo, H., Pari, J., Kim, Y., Andreas, J.: The surprising effectiveness of test-time training for few-shot learning. arXiv preprint arXiv:2411.07279 (2024)
3. Ball, T., Chen, S., Herley, C.: Can we count on LLMs? the fixed-effect fallacy and claims of GPT-4 capabilities. arXiv preprint arXiv:2409.07638 (2024)
4. Bessiere, C., Koriche, F., Lazaar, N., O'Sullivan, B.: Constraint acquisition. Artificial Intelligence **244**, 315–342 (2017)
5. Chollet, F., Knoop, M., Kamradt, G., Landers, B.: ARC prize 2024: Technical report. arXiv preprint arXiv:2412.04604 (2024)
6. Ferré, S.: Tackling the abstraction and reasoning corpus (ARC) with object-centric models and the MDL principle. In: International Symposium on Intelligent Data Analysis. pp. 3–15. Springer (2024)
7. Franzen, D., Disselhoff, J., Hartmann, D.: The LLM ARChitect: Solving ARC-AGI is a matter of perspective (2024)
8. Greenblatt, R.: Getting 50%(sota) on ARC-AGI with GPT-4o. Redwood Research Blog, June (2024)
9. Hasani, H., Izadi, A., Askari, F., Bagherian, M., Mohammadian, S., Izadi, M., Baghshah, M.S.: Understanding counting mechanisms in large language and vision-language models. arXiv preprint arXiv:2511.17699 (2025)
10. Hodel, M.: Addressing the abstraction and reasoning corpus via procedural example generation. arXiv preprint arXiv:2404.07353 (2024)
11. Hodel, M.: ARC-DSL: A Domain-Specific Language for Solving ARC Tasks. https://github.com/michaelhodel/arc-dsl (2024)
12. Jolicoeur-Martineau, A.: Less is more: Recursive reasoning with tiny networks. arXiv preprint arXiv:2510.04871 (2025)
13. Kumar, C.: Towards artificial general intelligence: Enhancing LLMs capability for abstraction and reasoning (2024)
14. Li, W.D., Hu, K., Larsen, C., Wu, Y., Alford, S., Woo, C., Dunn, S.M., Tang, H., Naim, M., Nguyen, D., et al.: Combining induction and transduction for abstract reasoning. arXiv preprint arXiv:2411.02272 (2024)
15. Ouellette, S.: Towards efficient neurally-guided program induction for ARC-AGI. arXiv preprint arXiv:2411.17708 (2024)
16. Pourcel, J., Colas, C., Oudeyer, P.Y.: Self-improving language models for evolutionary program synthesis: A case study on ARC-AGI. arXiv preprint arXiv:2507.14172 (2025)
17. Régin, F., De Maria, E., Bonlarron, A.: Combining constraint programming reasoning with large language model predictions. In: 30th International Conference on Principles and Practice of Constraint Programming (CP 2024). pp. 25–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2024)
18. Seely, J., Imajuku, Y., Zhao, T., Cetin, E., Jones, L.: Sudoku-bench: Evaluating creative reasoning with sudoku variants. arXiv preprint arXiv:2505.16135 (2025)
19. Sun, Y., Wang, X., Liu, Z., Miller, J., Efros, A., Hardt, M.: Test-time training with self-supervision for generalization under distribution shifts. In: International conference on machine learning. pp. 9229–9248. PMLR (2020)
20. Wind, J.S.: DSL solutions to the ARC challenge. https://github.com/top-quarks/ARC-solution/blob/master/ARC-solution_documentation.pdf (2020)

6 Appendix

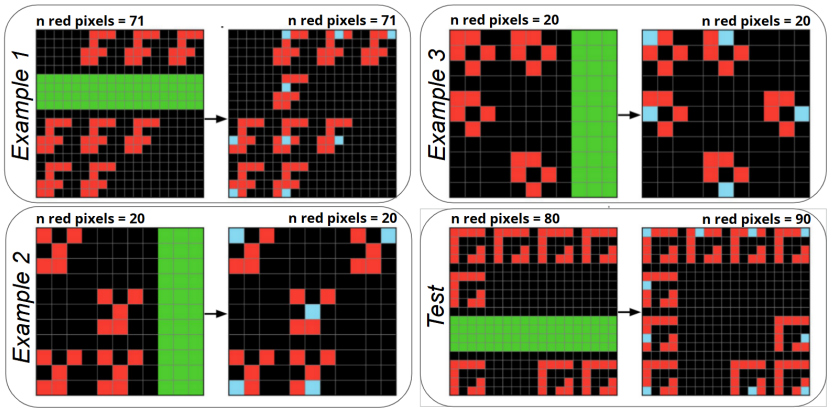

Fig. 4. An example of a problem that is an exception to the rule *"Counts of a specific color remains unchanged."* In all three examples, the number of red pixels remains constant: 71 in both the input and output of the first example, and 20 in both the input and output of the second and third examples. However, this fails in the test case: the number of red pixels increases from 80 in the input to 90 in the correct output.