Evolving and Refining Code: A Neuro-Evolutionary Program Synthesis for Compositional Reasoning

Anonymous submission

Abstract

We propose a neuro-evolutionary framework for program synthesis that combines Genetic Programming (GP) with a frozen Large Language Model (LLM) to tackle problems that require planning and compositional reasoning. Rather than generating programs from scratch, our method evolves candidate solutions using GP and then refines failed programs via LLM-based code edits. The LLM purely operates through a self-feedback prompting loop and serves as a semantic editor for improving GP-generated programs. The system operates in two stages. Stage 1: GP performs evolutionary search over a type-safe Domain-Specific Language (DSL), adopted from prior work, using the Distributed Evolutionary Algorithms in Python (DEAP) framework with strict typing to ensure semantic validity. Stage 2: If GP fails to solve the given task, the best candidate program so far is passed to LLM for repair. LLM receives the final best program, current input and output examples, program fitness score, and iteratively suggests edited programs until fitness improvements or the maximum retry loop is reached. Although recent methods rely heavily on fine-tuned LLMs, evolutionary approaches remain underexplored, especially in the Abstraction and Reasoning Corpus (ARC) benchmark. We show that frozen LLM can enhance genetic programming for code refinement. Experiments on the ARC reveal performance gains of the proposed approach over the GP-only baseline. Our goal is to show that evolutionary learning methods can be enhanced through LLM code editing, forming a neuro-evolutionary pipeline.

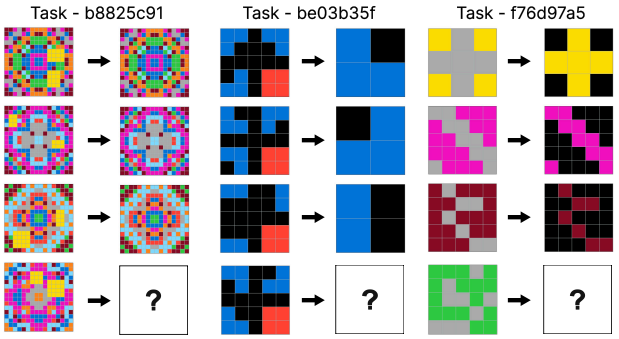


Figure 1: Sample tasks from the ARC benchmark. Each task provides a few input-output grid examples, where the goal is to infer the rule that maps input grids (left) to output grids (right) and apply the discovered rule to the test input (final row). Task `b8825c91` is to fill out the yellow patches based on the pattern of the remaining part. Task `be03b35f` is to guess the red patch based on the characteristics of other objects in the grid. Since all forms of N times 90-degree clockwise rotation of the left top object appear in the input and output grid, when $N = 3$, the rule of the task is to apply rotations on the left top object of the input grid. Task `f76d97a5` is to change the color gray to a non-gray color in the grid and the non-gray color to black.

Introduction

The task of program synthesis is to automatically generate a program that fulfills a given specification, often represented by input-output examples (O'Neill and Spector 2020). A robust program synthesizer must compose programs from reusable sub-components and generalize beyond training examples—combining learned subprograms in new ways to synthesize longer or conceptually novel programs that have the ability to reason over multi-step transformations (Shi et al. 2023). While neural based program synthesis approaches have made progress in this area these approaches have important limitations: (a) they are computationally expensive and hard to train, (b) a model has to be trained for each task (program) separately, (c) it is hard to interpret or verify the correctness of the learnt mapping (Parisotto et al. 2016).

Given these limitations, alternative approaches based on evolutionary search have received renewed interest in program synthesis. In particular, Genetic Programming (GP) has remained a robust and interpretable framework for learning-based program synthesis, due to its ability to evolve structured programs over Domain-Specific Languages (DSLs) using selection, crossover, and mutation (Sobania, Schweim, and Rothlauf 2021). Unlike neural models that require gradient-based training and massive data, GP directly manipulates symbolic program trees, enabling compositional reuse and open-ended exploration of the program space (Koza 1994b). These characteristics make GP especially appealing for synthesis tasks that require multi-step reasoning, hierarchical abstraction, and few-shot generalization, aligning closely with the capabilities expected of a strong synthesizer (Muslea 1997). How-

ever, GP suffers from a well-known limitation: it frequently stalls in local optima, especially in large, rugged program spaces, where small mutations often degrade fitness rather than improve it (Pantridge and Helmuth 2023).

In this work, we explore whether Genetic Programming alone is sufficient for solving compositional and multi-step reasoning tasks in the Abstraction and Reasoning Corpus (ARC) - a benchmark explicitly designed to test few-shot generalization, multi-step reasoning, and goal-directed planning abilities (Chollet 2019). Three example ARC tasks from the training set are shown in Figure 1.

The ARC tasks require abstract transformations over grid-based inputs, often involving compositional rules and reasoning over multiple steps to arrive at the correct output. To investigate this, we adopted a type-safe GP framework to evolve programs over a Domain-Specific Language (DSL), and used it as a base synthesizer for solving ARC tasks. However, consistent with long-standing observations in the GP literature (Koza 1994b; Goldberg 1989), we found that GP often stalls in local optima, particularly when small mutations fail to correct semantically meaningful yet structurally subtle errors in programs.

To overcome this limitation, and drawing on the emerging paradigm where LLMs are used to refine and debug program outputs (Tang et al. 2024; Wei, Xia, and Zhang 2023; Xu and ... 2023), we adopt a hybrid feedback mechanism: when the Genetic Programming baseline gets stuck, a frozen LLM is prompted with the best-evolved program, input-output pairs, and current fitness to perform code repair. We introduce a neuro-evolutionary ARC solver that incorporates LLMs' code repair into the GP pipeline. The LLM acts as a semantic editor, proposing code refinements of the program made by the GP. We observe that this LLM-guided editing process enables GP to escape local optima, producing correct programs that were otherwise unreachable through evolution alone.

Figure 2 outlines the two stages in the pipeline in which Stage 1 runs GP program synthesis with population P and Stage 2 passes the best individual program (if the GP failed to solve the given task) to the LLM for at most R repair rounds. The typed DSL enforces safety and keeps all solutions symbolic. The pseudo-code of this framework is shown in Algorithm 1.

Our contribution. We propose a neuro-evolutionary ARC program synthesizer that combines GP with an LLM-based code repair stage. When GP reaches near-solutions, the LLM edits the code to fix semantic errors to maximize the fitness. This hybrid flow enables more effective and open-ended exploration.

Related Work

Genetic Programming

Genetic Programming (GP) is an evolutionary algorithm for searching combinatorial spaces by evolving typed program trees $T \in \mathcal{T}$ to maximize a fitness function $f : \mathcal{T} \to \mathbb{R}$ (Koza 1994a). Each individual is represented as $(p, f(p))$, and a population of N individuals is denoted $\mathcal{P} = \{(p_1, f(p_1)), \ldots, (p_N, f(p_N))\}$. Over G generations, GP applies three core operators:

- **Selection.** $\mathrm{SEL} : \mathcal{T}^N \times \mathbb{R}^N \to \Delta(\mathcal{T}^N)$.
- **Crossover.** $\mathrm{XO} : \mathcal{T}^\nu \to \Delta(\mathcal{T})$.
- **Mutation.** $\mathrm{MUT} : \mathcal{T} \to \Delta(\mathcal{T})$.

However, traditional GP does not enforce that functions are applied to compatible data types, which becomes problematic when programs involve multiple data types and type-specific operations. This lack of constraint can significantly expand the search space and reduce generalization performance. Strongly Typed Genetic Programming (STGP) addresses this by introducing type constraints, ensuring that all functions operate only on appropriately typed inputs. By doing so, STGP reduces invalid program generation and guides the search toward semantically valid solutions (Montana 1995).

The ARC Benchmark and Approaches

The **Abstraction and Reasoning Corpus** (ARC) (Chollet 2019) is designed to evaluate fluid intelligence through few-shot grid transformations, where systems must infer latent rules from just a few input-output examples. To aid interpretability and concept-level evaluation, **ConceptARC** (Moskvichev, Odouard, and Mitchell 2023) extends ARC by categorizing tasks into high-level conceptual classes such as symmetry, object movement, and counting. This categorization enables more targeted analysis of model capabilities across distinct reasoning types. Despite recent progress, ARC remains a challenging benchmark for both neural and symbolic systems due to its emphasis on compositional generalization and minimal supervision.

Evolutionary Learning on ARC. Some approaches have explored evolutionary synthesis for ARC tasks. DSL with GE (Fischer et al. 2020) used grammatical evolution over DSLs, achieving top 4 % in the Kaggle competition, but solving only 30 tasks. LCSs (Coombe, Howard, and Browne 2024) applied Learning Classifier Systems with limited success (4.8 %), while Claude + EA (Berman 2024; Chollet et al. 2024) and SOAR (Pourcel, Colas, and Oudeyer 2025) introduced hybrid LLM-evolution strategies. SOAR achieves 52 % accuracy, but requires costly model queries and fine-tuning.

Domain Specific Language (DSL). A domain-specific language (DSL) is a programming or specification language designed to express solutions within a particular problem domain through concise, high-level abstractions (Van Deursen, Klint, and Visser 2000). For ARC-style reasoning tasks, where solutions involve complex visual and structural transformations over grid inputs, DSLs provide a symbolic and interpretable alternative to raw code generation. One widely adopted DSL in this ARC domain is the DSL introduced by Hodel (Hodel 2024), which defines a set of primitives for object manipulation, color filtering, grid operations, and compositional reasoning (Hodel 2024). Our work builds directly on this DSL, leveraging its typed expression to enable safe and structured search via STGP.

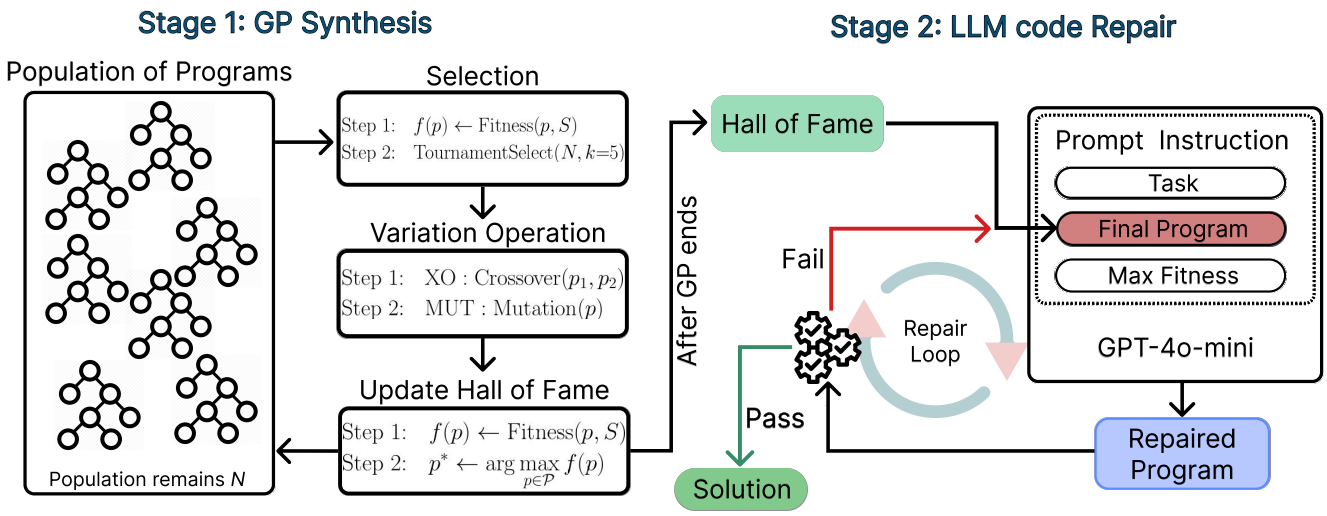


Figure 2: Framework Overview: A population of programs is evolved using Strongly Typed Genetic Programming (STGP) via crossover and mutation within a DSL. After each generation, programs are evaluated for task fitness. If no candidate solves the task, the best program is passed to a frozen LLM along with prompt instructions and fitness feedback. The LLM iteratively attempts to repair the program through semantic edits without fine-tuning.

System Overview

Stage 1: Genetic Programming-based Program Synthesis

We begin with STGP that synthesizes programs over a type-safe DSL, adopted from (Hodel 2024), implemented using the DEAP evolutionary computation library (Fortin et al. 2012). Our formulation and operators follow the classical genetic programming paradigm (Koza 1994b; Goldberg 1989). Each candidate program is denoted as $p \in \mathcal{P}$, where $\mathcal{P}$ denotes the program space, and is represented as a typed expression tree $T = [n_0, n_1, \ldots, n_k]$. Here, each internal node n_i corresponds to a DSL primitive of the form $r : (\tau_1, \ldots, \tau_m) \rightarrow \tau_{\text{out}}$, indicating a function with input types $\tau_1, \ldots, \tau_m$ and output type τ_{out}. Note that T denotes the program's expression tree, representing the syntactic structure of a candidate program, while p is the executable function that results from evaluating T on a given input. The leaf nodes are called terminals, whose values are either constants, booleans, or the input grid. All types τ are drawn from a finite set that has been defined in this work (Hodel 2024). Some of the DSL types defined are (`Grid`, `Colour`, `List[Grid]`). The GP system maintains a population $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$, where each individual p is evaluated on a training set $\mathcal{S} = \{(x_i, y_i)\}_{i=1}^{N}$. The fitness of a program is computed using a pixel-wise similarity function $\text{sim}(\cdot, \cdot)$ as used in most prior works (Xu, Khalil, and Sanner 2023).

$$f(p) = \sum_{i=1}^{N} \text{sim}(p(x_i), y_i)$$

The goal of evolution is to identify the highest-fitness program:

$$p^* = \arg\max_{p \in \mathcal{P}} f(p)$$

The initial population is created using `genFull` that creates a full program tree by first picking the target depth and then recursively expanding nodes until every root-to-leaf path has h depth. At each generation, the population undergoes three genetic operations adopted from (Fortin et al. 2012). The use of strong typing ensures that all generated programs are executable and meaningful. This not only avoids runtime errors but also reduces the search space significantly, enabling GP to explore functional program structures. Note that throughout this paper, we will use the terms STGP and GP interchangeably.

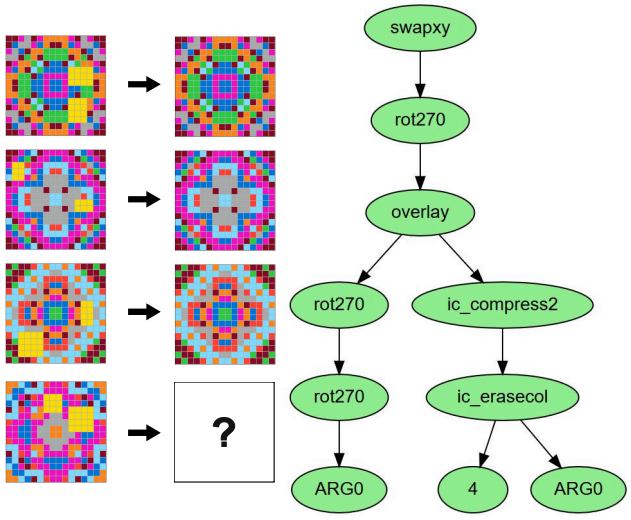


Figure 3: Task `b8825c91` and the symbolic expression tree generated by our GP pipeline.

Algorithm 1: Neuro-evolutionary Program Synthesizer via STGP and LLM Repair

Input: Training examples $\mathcal{S} = \{(x_i, y_i)\}_{i=1}^{N}$
Output: function `solve(inp)` mapping $x \mapsto y$

Stage 1: GP-based Program Synthesis
Initialize population $\mathcal{P}^{(0)} = \{p_1, \ldots, p_N\} \sim \mathcal{U}(\mathcal{P})$
for *generation* $g = 1$ **to** G **do**
  foreach $p \in \mathcal{P}^{(g)}$ **do**
    Compute fitness: $f(p) = \sum_{i=1}^{N} \text{sim}(p(x_i), y_i)$
  if $\exists\, p^* \in \mathcal{P}^{(g)}$ *such that* $f(p^*) = f_{\max}$ **then**
    return p^*
  Select parents via tournament selection
  Apply type-safe crossover and mutation
  Form next generation: $\mathcal{P}^{(g+1)}$
  Update Hall-of-Fame with top-performing individual
Let $p^* = \arg\max_{p \in \mathcal{P}^{(g)}} f(p)$

Stage 2: LLM-based Program Repair
for *repair round* $r = 1$ **to** R **do**
  Construct prompt π_r from I/O examples, fitness and DSL code for p^*
  Query LLM with π_r to obtain candidate: $p'_r \sim \text{LLM}(\pi_r)$
  Evaluate fitness: $f(p'_r) = \sum_{i=1}^{N} \text{sim}(p'_r(x_i), y_i)$
  if $f(p'_r) > f(p^*)$ **then**
    Update $p^* \leftarrow p'_r$
return Final program p^*

Program Representation. Our system represents candidate programs as symbolic expression trees. Figure 3 shows an example generated by our GP pipeline. The tree is built bottom-up from terminal nodes such as constants and input grids, which are composed through typed DSL primitives. In Task `b8825c91`, the program rotates the input, applies `ic_erasecol` and `ic_compress2`, combines results with `overlay`, and finishes with `swapxy`. These trees reflect the pixel-wise optimization objective, with type correctness enforced throughout by the DSL.

Stage 2: LLM-based Code Repair

While STGP enables type-safe exploration of the DSL space, we found that it often struggles to escape local optima. In this case, GP may converge to syntactically correct but semantically incorrect programs that partially match the outputs without solving the task completely, as shown in Figure 4. To address this and inspired by (Xu and ... 2023), we incorporate a second refinement phase using a frozen LLM, specifically GPT-4o-mini, to perform code repair. After GP completes 500 generations, if the best evolved program does not achieve the perfect fitness score, a repair loop is instantiated for up to R rounds. In each round, we constructed a prompt that includes the final program, the input and output training examples, the program's fitness score, and the max score it has to achieve for the given task. This repair continues until either perfect fitness is reached or the repair rounds are exhausted. Let $p^* \in \mathcal{P}$ be the best-evolved candidate after G generations. If $f(p^*) < f_{\max}$, the system enters a repair loop for up to R refinement rounds. At each round, a natural language prompt is constructed using three as follows.

- The full set of input-output training pairs
- The symbolic DSL source code of p^*
- The current program and max fitness score of the task

This prompt (π_r) is submitted to the LLM, which returns a revised program $p'_r \sim \text{LLM}(\pi_r)$ expressed in Python. The revised program is executed and evaluated. If it yields a higher fitness score than the previous best, it is accepted.

$$f(p'_r) > f(p^*) \quad \Rightarrow \quad p^* \leftarrow p'_r$$

This procedure stops either when a perfect solution is found or when the maximum number of rounds R is reached. For the experiment result shown in Figure 5 and 6, the repair loop used is set to five ($R = 5$). Code edit round ($R = 0$, $R = 5$) refers to the number of times that the LLM can edit its own code. As shown in Figure 2, there is a loop letting the LLM edit its code, which is R. For our experiment setting, we compare the result when $R = 0$ and $R = 5$.

Case Study: Success and Failure Analysis

This case study serves as a concrete example illustrating both the overall functioning of our two-stage framework and the complementary roles of GP and LLM-based repair. The ARC task shown in Figure 4 is from the concept class with task number `Center7`. As shown in the Figure 4, the first stage processes the task using the GP synthesis process, resulting in the synthesized program shown in step 2 of Figure 4. This program represents the final output of the GP search process and uses the following primitives: `crement_tuple`,`sign_tuple`, `mostcolor_objs`, `interval_int`, `mostcolor_objs`,`substract_int`, `add_int` with terminals `4`, `2`, and `5` . While the program is syntactically well-formed and type-safe, it fails to capture the key semantic operation: detecting and reasoning over object-level color content in the input grid. As a result, the GP solution is semantically misaligned with the task requirements scoring fitness score of 0. Step 3 LLM Code Repair shows the Stage 2 of our framework, where the LLM-based code repair module is invoked. This module receives the best GP program, along with the input-output grid pairs, the fitness score, and a prompt requesting code edits. The program shown in stage 3 of Figure 4 is the edited code which raises the fitness score from 0 (fitness score by GP) to 0.7(fitness score of the repaired program). This shows how the LLM fixes high-level semantic errors that GP alone could not resolve and help it achieve high fitness score than GP alone.

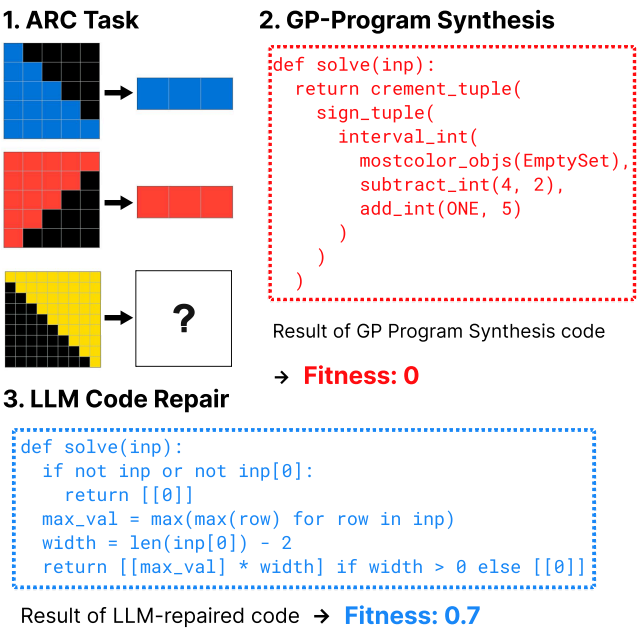


Figure 4: Case study showing GP program failure and LLM repair on ConceptARC task id: `Center7` (Moskvichev, Odouard, and Mitchell 2023). The flow of the process starts with **(1)** illustrating the input-output training examples**(2)** Stage 1 applies GP to evolve a program over a symbolic DSL. The resulting expression is syntactically valid but semantically incorrect, producing an invalid output grid. **(3)** Stage 2 invokes a frozen LLM to repair the GP-generated program by editing the function directly based on example pairs and fitness. The new function the LLM edited takes a grid as input and returns the grid as an output, which matches the DSL type system defined. This example demonstrates how LLM-based repair can resolve semantic failures that GP alone cannot.

Experiments

Experiment Setting

This section starts with the experimental settings and the benchmark problem, and visualizes the evaluation result of our framework on the 400 ARC training data set and 160 ConceptARC dataset (Chollet 2019; Moskvichev, Odouard, and Mitchell 2023), which include tasks requiring abstraction, pattern matching, and symbolic reasoning. Each task, as shown in Figure 1, consists of input-output grid pairs, and the objective is to synthesize a function `solve(inp)` that generalizes to unseen grids.

For evaluation, we adopt two widely used metrics from the program synthesis literature. First, pixel-wise accuracy which measures the fraction of correctly predicted pixels across all test grids (Bober-Irizar and Banerjee 2024). Second, we report Pass@k, a probabilistic metric introduced by (Chen et al. 2021), which estimates the likelihood that at least one of the top-k synthesized programs produces a fully correct output on all test cases. We report Pass@k for evaluating both the GP baseline and the full framework that incorporates LLM-based repair.

The terminal set for the GP includes integer color values (ranging from `0 to 9`), boolean values (`True`, `False`), and an Input grid (`ARG0`). The primitive set includes DSLs from (Hodel 2024).

Each synthesis attempt runs the GP engine with a population of 1,000 individuals (population size remains the same throughout the generation) for up to 500 generations with crossover and mutation probability of 0.5 and max depth of tree set to 80. All results are averaged over five random seeds. If no perfect solution is found via GP alone, the LLM-based repair stage is invoked.

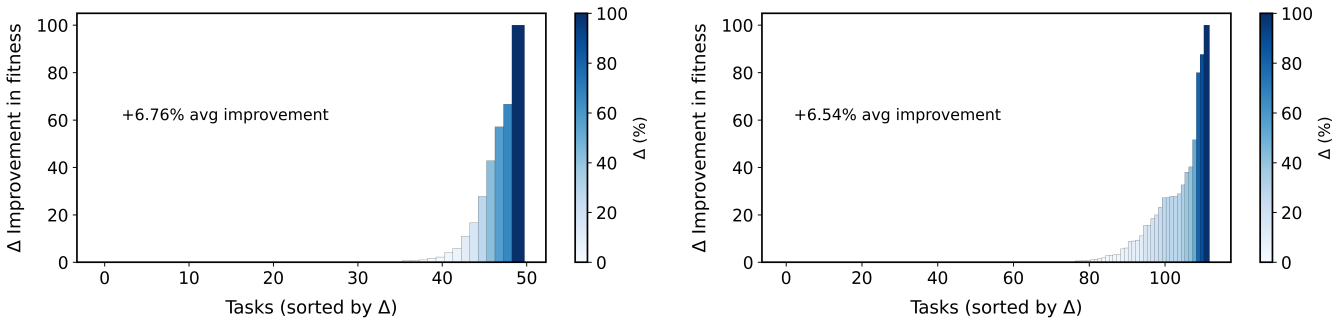
Table 1: Performance comparison between our approach and previous systems. Results for prior work are reported as presented in (Bober-Irizar and Banerjee 2024). All systems were assessed using the official ARC evaluation procedure, allowing up to three attempts per task, and a task is considered solved if it is completed correctly.

System	Solved Tasks
ARGA (Xu, Khalil, and Sanner 2023)	49 / 400
Dreamcoder (Alford 2021)	23 / 400
Dreamcoder-DC (Bober-Irizar and Banerjee 2024)	70 / 400
GP Program Synthesis (Stage 1 of our work)	65 / 400

Result Analysis

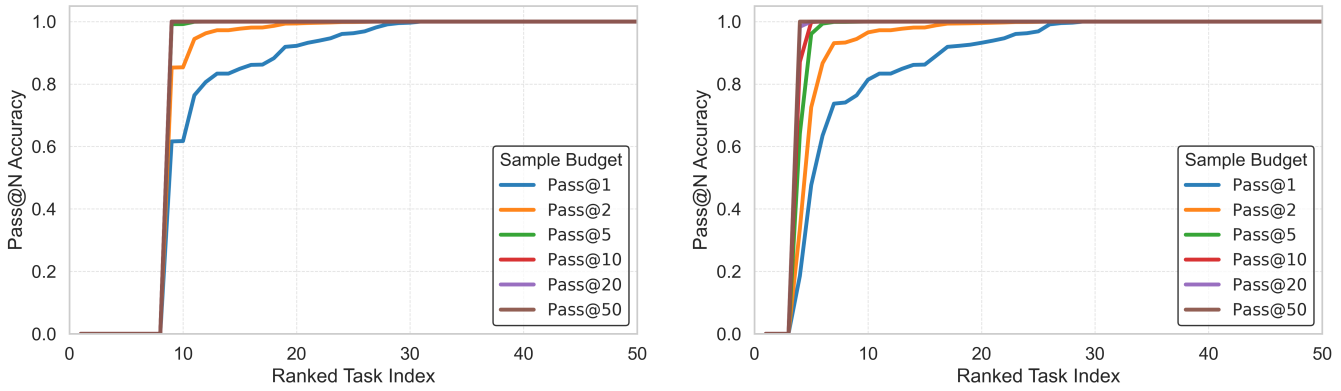
As shown in Table 1our approach solves 65 out of 400 tasks, outperforming ARGA solving 49 tasks and DreamCoder (Xu, Khalil, and Sanner 2023; Alford 2021). Prior results are taken directly taken as reported in (Xu, Khalil, and Sanner 2023). ARGA uses a hand-engineered DSL based on graph abstractions, which represents solutions as a sequence of symbolic graph transformations to solve 49 tasks. DreamCoder (Alford 2021) solves 23 tasks by representing programs using functional language and synthesized using enumeration. Its improved variant (Bober-Irizar and Banerjee 2024) represents programs as symbolic functional programs by incorporating learned priors and crafting DSLs for their system, achieving 70 tasks. Our system uses genetic programming over the DSL designed by Hodel, without introducing any new or custom DSLs. It solves 65 tasks, demonstrating that general-purpose evolutionary search can be competitive with more domain-specialized symbolic approaches. We do not compare our method to recent LLM-based techniques that rely heavily on fine-tuning with large synthetic datasets, as our focus is on zero-shot or low-resource synthesis without synthetic data augmentation.

The results in Figure 5 show the change in program fitness after applying LLM code edits on top of the GP-only baseline. Figure 5a shows the improvement across 50 tasks from the ConceptARC benchmark, with an average fitness gain of +6.54%. Figure 5b presents results on 112 tasks sampled from the 400 ARC training tasks, yielding an average gain of +6.54% in fitness. In both Figure 5a) and 5b), each bar represents a task and shows the fitness difference (improvement) called (Δ) after the LLM edits, with tasks sorted by



(a) 50 ConceptARC Tasks (b) 112 Tasks from ARC Training Set

Figure 5: Results show the improvement in fitness score (Δ) per task after LLM-based code edits for two different datasets. On average, the fitness score improved by +6.76% and +6.54% in each dataset. Tasks are sorted by fitness improvement in both.



(a) GP-only Program Synthesis (b) GP with LLM Code Editing

Figure 6: Simulated pixel-level Pass@k accuracy curves across 50 tasks. Each line estimates the probability that at least one of k samples achieves $\geq 90\%$ pixel-level similarity. Compared to GP-only, LLM code edits yield higher early-stage success probabilities (Pass@1, Pass@2), showing faster convergence and improved initial solution quality.

(Δ) value. These plots demonstrate that LLM code editing increases the fitness score across the tasks.

Figure 6 shows pixel-level Pass@k accuracy for 50 ARC tasks using (a) GP-only synthesis and (b) GP with LLM code editing. For each task, the fitness score achieved by the method is normalized by the maximum possible fitness for that task, resulting in a ratio between 0 and 1. This ratio is then treated as an estimator of how likely a generated program is to reach 90% pixel-level accuracy. Using this estimated per-sample probability, the Pass@k accuracy is computed as the probability that at least one of the K independent samples would reach that threshold. The key takeaway is that LLM code editing with GP improves estimated success probabilities in the early samples, such as Pass@1 and Pass@2, indicating that LLM code editing improves the quality of candidate programs and accelerates convergence toward high-accuracy outputs with limited samples.

Figure 7 presents a comparative analysis of the performance of the baseline GP, using LLM ($R = 0$), LLM ($R = 5$), and GP with LLM repair loop ($R = 5$). The evaluation covers 160 tasks from the ConceptARC (Moskvichev, Odouard, and Mitchell 2023), where each task belongs to one of 16 high-level concepts listed. Each high-level concept on the x-axis represents a group of 10 tasks that share a common abstract high-level reasoning. The y-axis reflects the average pixel-wise performance within each concept, averaged over its corresponding tasks. The results show that the GP with LLM code edits outperforms the baseline GP model in the majority of high-level concepts. Specifically, substantial improvements are observed in concepts such as `ExtractObjects`, `Count`, `FilledNotFilled`, `MoveToBoundary`, `ExtendToBoundary`, and `TopBottom`. The LLM with repair rounds consistently outperforms both the GP-only and LLM zero-shot baselines. In contrast, the differences are less pronounced in concepts like `CleanUp` and `CompleteShape`, where both models perform similarly, showing that the baseline GP model is already sufficiently effective at handling these tasks, and LLM-based edits offer limited benefit. The LLM with code repair round of 0 performs worse in most of the categories, showing the importance of the initial GP search in constraining the solution space before invoking the LLM refinement. These findings support the view that LLMs can complement

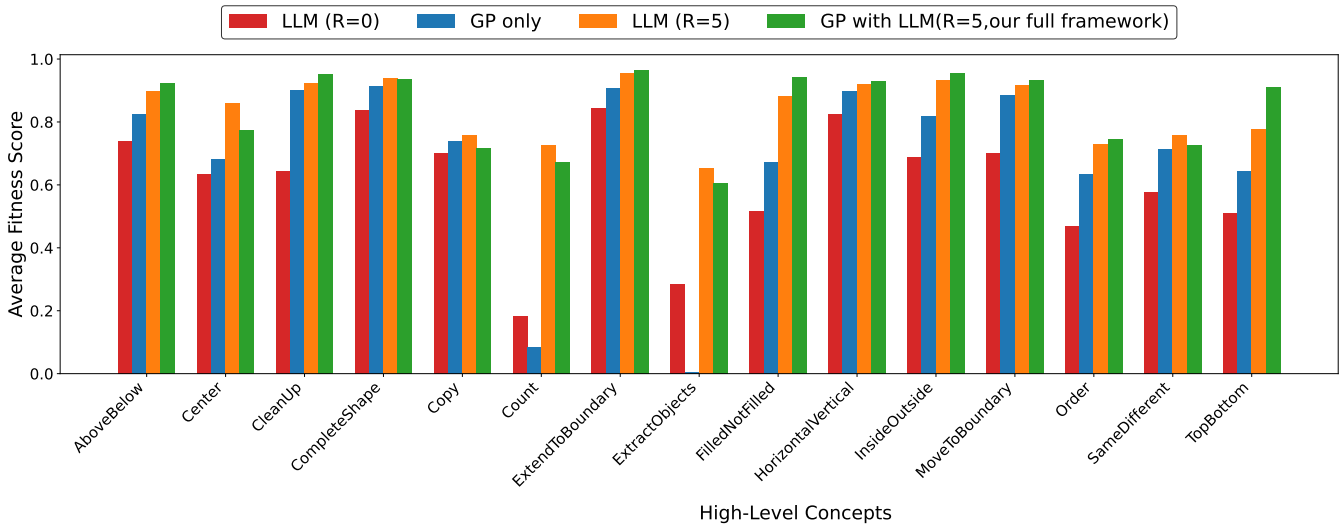


Figure 7: Performance comparison of LLM-only without repair round ($R = 0$), GP-only, LLM-only with five repair rounds ($R = 5$), and GP with LLM code edit with five repair rounds ($R = 5$, our full framework) on the ConceptARC over the high-level concept categorization.

evolutionary program synthesis by injecting semantic refinements into existing code. Moreover, as shown in the results, increasing the number of LLM refinement rounds from 1 to 5 improves performance. Integrating LLM-driven edits into evolutionary loops appears to be a promising strategy to enhance evolutionary program synthesis. Out of 160 tasks of ConceptARC dataset our full framework solve 25 tasks.

Discussion

Why is GP used as the base rather than starting from LLM-generated candidates or neural guidance?

We chose not to start from LLM-generated programs because our primary goal was to explore how GP, well-suited for multi-step transformation and planning tasks, can be enhanced by integrating with foundation models. Our aim was not to optimize ARC benchmark performance but to use ARC as a test bed for studying how evolutionary methods can be made more flexible and effective by leveraging LLMs. We are inspired by GP's strength in structured search spaces and sought to show that these classic methods can benefit from LLMs without being replaced. Although research applying GP to ARC is limited, we intend to show that evolutionary processes can complement foundation models. In this way, our work builds on and reinforces prior directions in evolutionary learning by showing how such methods can remain open-ended and adaptive in the LLM era.

What happens when the LLM produces a solution that no longer uses the existing DSL—can such code still be considered part of the symbolic search space, and how does this fit into the overall evolutionary process?

While our framework prompts the LLM to refine the symbolic programs produced by the genetic programming using the predefined DSL, we do not constrain the model to stay strictly within the space. In some cases, such as shown in Figure 4, the LLM synthesis solution uses Python operations, discarding the original DSL expressions. This case will be interpreted as the LLM dynamically generating a new DSL-like function, one that still maps input grids into output grids in a consistent and composable way, since we have other DSLs in our DSL pool that map input grids to output grids—this function made by the LLM also has the same functionality. From this perspective, the LLM is expanding the search space by injecting novel program representations that can be treated as valid individuals. This opens future works in which LLM-generated programs are not only fallbacks but can be reintegrated into the GP population for further evolution and recombination.

Conclusion

We propose a neuro-evolutionary framework that combines STGP with LLMs to synthesize programs for ARC tasks. Our approach evolves symbolic programs over a DSL, and when GP alone is insufficient, it invokes an LLM to refine them through semantic code edits. Experiments on the ARC and ConceptARC benchmarks show that LLM-based repair improves upon GP-only baselines, enabling synthesis on tasks otherwise unreachable by GP. While our framework currently treats LLM-edited programs as a terminal repair step, our findings suggest that these programs, which satisfy the type constraints of the DSL primitives, could also be reintegrated into the evolutionary population. This opens a promising direction where LLM-generated code not only patches failures but expands the search space, enabling open-ended and adaptive synthesis.

References

Alford, S. 2021. *A neurosymbolic approach to abstraction and reasoning*. Ph.D. thesis, Massachusetts Institute of Technology.

Berman, J. 2024. How I Got a Record 53.6% on ARC-AGI-Pub Using Claude 3.5 and Evolutionary Test-Time Compute. https://jeremyberman.substack.com/p/how-i-got-a-record-536-on-arc-agi. Blog post, accessed 17 Jul 2025.

Bober-Irizar, M.; and Banerjee, S. 2024. Neural networks for abstraction and reasoning: Towards broad generalization in machines. *arXiv preprint arXiv:2402.03507*.

Chen, M.; Tworek, J.; Jun, H.; and et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*.

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

Chollet, F.; Knoop, M.; Kamradt, G.; and Landers, B. 2024. ARC Prize 2024: Technical Report. Technical Report arXiv:2412.04604, ARC Prize Foundation.

Coombe, C.; Howard, D. G.; and Browne, W. N. 2024. Learning Classifier Systems as a Solver for the Abstraction and Reasoning Corpus. In *Companion Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '24)*.

Fischer, R.; Jakobs, M.; Mücke, S.; and Morik, K. 2020. Solving Abstract Reasoning Tasks with Grammatical Evolution. In *LWDA 2020 – Proceedings of the Workshop Day "Lernen, Wissen, Daten, Analysen"*, volume 2738 of *CEUR Workshop Proceedings*, 65–76.

Fortin, F.-A.; De Rainville, F.-M.; Gardner, M.-A.; Parizeau, M.; and Gagné, C. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research*, 13: 2171–2175.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley.

Hodel, M. 2024. arc-dsl: A Domain-Specific Language for ARC. https://github.com/michaelhodel/arc-dsl. Accessed: 2025-07-25.

Koza, J. R. 1994a. Genetic Programming as a Means for Programming Computers by Natural Selection. *Statistics and Computing*, 4(2): 87–112.

Koza, J. R. 1994b. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press.

Montana, D. J. 1995. Strongly typed genetic programming. *Evolutionary computation*, 3(2): 199–230.

Moskvichev, A.; Odouard, V. V.; and Mitchell, M. 2023. The conceptarc benchmark: Evaluating understanding and generalization in the arc domain. *arXiv preprint arXiv:2305.07141*.

Muslea, I. 1997. SINERGY: A linear planner based on genetic programming. In *European Conference on Planning*, 312–324. Springer.

O'Neill, M.; and Spector, L. 2020. Automatic programming: The open issue? *Genetic Programming and Evolvable Machines*, 21(1): 251–262.

Pantridge, E.; and Helmuth, T. 2023. Solving novel program synthesis problems with genetic programming using parametric polymorphism. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 1175–1183.

Parisotto, E.; Mohamed, A.-r.; Singh, R.; Li, L.; Zhou, D.; and Kohli, P. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*.

Pourcel, J.; Colas, C.; and Oudeyer, P. 2025. Self-Improving Language Models for Evolutionary Program Synthesis: A Case Study on ARC-AGI. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*. Poster.

Shi, K.; Hong, J.; Deng, Y.; Yin, P.; Zaheer, M.; and Sutton, C. 2023. Exedec: Execution decomposition for compositional generalization in neural program synthesis. *arXiv preprint arXiv:2307.13883*.

Sobania, D.; Schweim, D.; and Rothlauf, F. 2021. Recent developments in program synthesis with evolutionary algorithms. *arXiv preprint arXiv:2108.12227*.

Tang, H.; Hu, K.; Zhou, J.; Zhong, S. C.; Zheng, W.-L.; Si, X.; and Ellis, K. 2024. Code repair with llms gives an exploration-exploitation tradeoff. *Advances in Neural Information Processing Systems*, 37: 117954–117996.

Van Deursen, A.; Klint, P.; and Visser, J. 2000. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6): 26–36.

Wei, Y.; Xia, C. S.; and Zhang, L. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 172–184.

Xu, W.; and ... 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *NeurIPS*.

Xu, Y.; Khalil, E. B.; and Sanner, S. 2023. Graphs, constraints, and search for the abstraction and reasoning corpus. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, 4115–4122.

Reproducibility Checklist

Instructions for Authors:
This document outlines key aspects for assessing reproducibility. Please provide your input by editing this `.tex` file directly.

For each question (that applies), replace the "Type your response here" text with your answer.

Example: If a question appears as

```
\question{Proofs of all novel claims
are included} {(yes/partial/no)}
Type your response here
```

you would change it to:

```
\question{Proofs of all novel claims
are included} {(yes/partial/no)}
yes
```

Please make sure to:

- Replace ONLY the "Type your response here" text and nothing else.
- Use one of the options listed for that question (e.g., **yes**, **no**, **partial**, or **NA**).
- **Not** modify any other part of the `\question` command or any other lines in this document.

You can `\input` this .tex file right before `\end{document}` of your main file or compile it as a stand-alone document. Check the instructions on your conference's website to see if you will be asked to provide this checklist with your paper or separately.

1. General Paper Structure

1.1. Includes a conceptual outline and/or pseudocode description of AI methods introduced (yes/partial/no/NA) yes

1.2. Clearly delineates statements that are opinions, hypothesis, and speculation from objective facts and results (yes/no) yes

1.3. Provides well-marked pedagogical references for less-familiar readers to gain background necessary to replicate the paper (yes/no) yes

2. Theoretical Contributions

2.1. Does this paper make theoretical contributions? (yes/no) no

If yes, please address the following points:

2.2. All assumptions and restrictions are stated clearly and formally (yes/partial/no)

2.3. All novel claims are stated formally (e.g., in theorem statements) (yes/partial/no)

2.4. Proofs of all novel claims are included (yes/partial/no)

2.5. Proof sketches or intuitions are given for complex and/or novel results (yes/partial/no)

2.6. Appropriate citations to theoretical tools used are given (yes/partial/no)

2.7. All theoretical claims are demonstrated empirically to hold (yes/partial/no/NA)

2.8. All experimental code used to eliminate or disprove claims is included (yes/no/NA)

3. Dataset Usage

3.1. Does this paper rely on one or more datasets? (yes/no) yes

If yes, please address the following points:

3.2. A motivation is given for why the experiments are conducted on the selected datasets (yes/partial/no/NA) yes

3.3. All novel datasets introduced in this paper are included in a data appendix (yes/partial/no/NA) NA

3.4. All novel datasets introduced in this paper will be made publicly available upon publication of the paper with a license that allows free usage for research purposes (yes/partial/no/NA) NA

3.5. All datasets drawn from the existing literature (potentially including authors' own previously published work) are accompanied by appropriate citations (yes/no/NA) yes

3.6. All datasets drawn from the existing literature (potentially including authors' own previously published work) are publicly available (yes/partial/no/NA) yes

3.7. All datasets that are not publicly available are described in detail, with explanation why publicly available alternatives are not scientifically satisficing (yes/partial/no/NA) NA

4. Computational Experiments

4.1. Does this paper include computational experiments? (yes/no) yes

If yes, please address the following points:

4.2. This paper states the number and range of values tried per (hyper-) parameter during development of the paper, along with the criterion used for selecting the final parameter setting (yes/partial/no/NA) yes

4.3. Any code required for pre-processing data is included in the appendix (yes/partial/no) partial

4.4. All source code required for conducting and analyzing the experiments is included in a code appendix (yes/partial/no) yes

4.5. All source code required for conducting and analyzing the experiments will be made publicly available upon publication of the paper with a license that allows free usage for research purposes (yes/partial/no) yes

4.6. All source code implementing new methods have comments detailing the implementation, with references to the paper where each step comes from (yes/partial/no) yes

4.7. If an algorithm depends on randomness, then the method used for setting seeds is described in a way sufficient to allow replication of results (yes/partial/no/NA) no

4.8. This paper specifies the computing infrastructure used for running experiments (hardware and software), including GPU/CPU models; amount of memory; operating system; names and versions of relevant software libraries and frameworks (yes/partial/no) No

4.9. This paper formally describes evaluation metrics used and explains the motivation for choosing these metrics (yes/partial/no) yes

4.10. This paper states the number of algorithm runs used to compute each reported result (yes/no) yes

4.11. Analysis of experiments goes beyond single-dimensional summaries of performance (e.g., average; median) to include measures of variation, confidence, or other distributional information (yes/no) yes

4.12. The significance of any improvement or decrease in performance is judged using appropriate statistical tests (e.g., Wilcoxon signed-rank) (yes/partial/no) no

4.13. This paper lists all final (hyper-)parameters used for each model/algorithm in the paper's experiments (yes/partial/no/NA) yes