Solving the Abstraction and Reasoning Corpus with Genetic Programming

Paulo Silva[*] Alcides Fonseca[†]

July 24, 2025

Abstract

The Abstraction and Reasoning Corpus (ARC) is a benchmark for artificial general intelligence that requires systems to infer abstract rules from minimal examples. State-of-the-art solutions are now predominantly ensembles combining large neural models with symbolic program synthesizers. While this hybrid approach is powerful, the efficiency of the underlying symbolic search component remains a critical factor. This paper investigates how to optimize this component using Genetic Programming (GP), an inductive approach that seeks to find a general problem-solving program. We represent solutions as programs in a rich Domain-Specific Language (DSL) and systematically compare evolutionary selection strategies and fitness functions. Our results show that guided evolutionary search, specifically using Lexicase and Tournament selection with a simple pixel-based fitness function, significantly outperforms a Random Search baseline. Furthermore, we find a crucial trade-off in fitness design, where the simpler pixel-based function is more effective than a complex, multi-component one due to a 4x increase in evaluation speed. Analysis of the solved tasks reveals that different algorithms have complementary strengths, solving unique subsets of problems, which supports the case for an ensemble approach in developing next-generation hybrid ARC solvers.

1 Introduction

The Abstraction and Reasoning Corpus (ARC) [1] is a collection of visual reasoning tasks designed to measure fluid intelligence. Unlike traditional machine learning benchmarks, ARC requires a system to infer an underlying abstract rule from a few examples and apply it to a new test case. This frames the challenge as a problem of induction: the goal is to find a general program that explains the examples, rather than simply predicting the output for a specific test case.

The field of ARC research has been largely divided into two paradigms. The first is inductive program synthesis, where symbolic methods like Genetic Programming (GP) search for an explicit program in a Domain-Specific Language (DSL). The second is a transductive approach, often using Large Language Models (LLMs), which aims to directly predict the test output without necessarily finding a general rule. As of the 2024 ARC Prize, the state-of-the-art is dominated by ensembles that combine both paradigms [2]. This proves that neither approach is sufficient alone and underscores the continued importance of the inductive program synthesis component.

This paper therefore addresses a critical research question: how can we best optimize the evolutionary search for this inductive component? Our goal is to evaluate Genetic Programming (GP), using the GeneticEngine framework [3], as a symbolic search engine for ARC. We conduct a systematic comparison of core evolutionary mechanisms, including selection methods, fitness functions, and genetic operators, to determine the optimal configuration and to assess whether this optimized GP can outperform baseline algorithms. By strengthening this foundational component, we aim to contribute to the development of more powerful hybrid ARC solvers.

2 Related Work

Solving ARC is fundamentally a search problem. The evolution of solver paradigms reflects a progression from exhaustive symbolic search to complex neuro-symbolic ensembles. A common thread is the reliance

[*]Massachusetts Institute of Technology (MIT). This work was conducted during a summer internship at LASIGE.
[†]LASIGE, Faculdade Ciências, Universidade de Lisboa.

on a manually designed DSL to encode prior knowledge, a practice validated by the foundational ‘arc-dsl‘ [4].

The first major breakthroughs were purely symbolic and inductive. The winning solution to the 2020 Kaggle competition was a highly-optimized C++ enumerative search engine that established an impressive non-learning baseline, proving that many tasks were solvable through the composition of well-chosen primitives [5]. Given the combinatorial explosion of this approach, Genetic Programming (GP) emerged as a natural heuristic to guide the search more efficiently. Early work by Fischer et al. (2020) showed that Grammatical Evolution could significantly outperform random search, validating the evolutionary approach [6].

The 2024 ARC Prize marked a paradigm shift with the success of methods using LLMs. These fall into two main camps: transductive approaches like Test-Time Training (TTT), where a model is fine-tuned on task examples at inference time to directly predict the output [7, 8], and inductive approaches like large-scale program generation, where a frontier LLM generates thousands of candidate programs [9]. Critically, the top-performing systems are now ensembles of these different approaches.

This new landscape confirms the continued relevance of the inductive symbolic component and motivates our work. While early research showed GP was viable, the question of how to best configure the evolutionary search for ARC’s multi-objective and sparse-reward environment remains under-explored. This paper addresses this gap by comparing selection strategies and fitness functions to make the inductive search engine, a key part of any modern ARC solver, more powerful and efficient.

3 Methodology

3.1 Program Representation: Context-Free Grammar GP

Each potential solution is represented as a program tree, structured according to a Context-Free Grammar (CFG). This ensures that every program generated through mutation or crossover is syntactically correct by design, preventing wasted evaluations on invalid code. Our DSL is an extension and optimization of the original functional DSL developed by Michael Hodel [4]. The DSL is highly expressive, containing over 150 base functions. A key challenge in representing this DSL as a typed grammar is that many functions are polymorphic: they can accept arguments of different types (e.g., a ‘Grid’ or an ‘Object’) or return values of different types. To enforce type safety and guide the evolutionary search, we implemented specific, strongly-typed classes for each function signature. Furthermore, to support higher-order functions like ‘map’ or ‘filter’, we created a separate set of classes that evaluate to function objects rather than data values. This comprehensive, type-safe representation resulted in a final grammar of 320 distinct classes, which serve as the building blocks for the program trees.

3.2 Grammar Weight Initialization

To bias the initial search towards more promising areas of the vast program space, the production rules of the grammar were assigned initial weights. These weights were not chosen arbitrarily but were derived empirically from the hand-written solutions for the ARC training tasks provided by Hodel [4]. We analyzed this set of human-written solutions and counted the frequency of each DSL function’s usage. The resulting frequency counts were then used as the initial weights for the corresponding nodes in our grammar. The heuristic behind this approach is that the functions most frequently used by a human expert to solve these puzzles are likely the most useful and common building blocks for human-like abstract reasoning in this domain.

3.3 The GeneticEngine Framework

The evolutionary algorithm was implemented using GeneticEngine, a Python library for Genetic Programming. GeneticEngine uses modern Python features, such as type annotations, to automatically extract the grammar from a set of user-defined classes and functions. This allows for a declarative and highly readable definition of the search space. The framework is flexible, supporting a wide range of search algorithms, representations, and evolutionary operators, which facilitated the comparative experiments conducted in this research.

3.4 DSL Optimization and Robustness

Initial implementations of the DSL functions relied on standard Python loops, which proved to be a significant performance bottleneck. To address this, the core grid manipulation functions were optimized by rewriting them to use high-performance libraries:

- **NumPy:** Vectorized operations were used to replace explicit loops, enabling much faster computations on the grid arrays.
- **Numba:** The `@njit` (Just-In-Time compilation) decorator was applied to computationally intensive functions, compiling them to highly optimized machine code at runtime.

Furthermore, it was discovered that the evolutionary search frequently generated programs that produced invalid outputs, such as grids with dimensions larger than the maximum allowed 30x30. This led to wasted time on expensive computations on large grids. To ensure stability, validation checks were added to all functions capable of increasing grid size, such as `upscale`, `hupscale`, `vupscale`, and `hconcat`. These functions now raise an error if an operation would result in an invalid dimension, making the search process more robust.

3.5 Search Optimizations

To further optimize the search process, two additional strategies were attempted:

- **Fitness Caching:** A caching mechanism was implemented to store and retrieve the fitness scores of previously evaluated program trees, aiming to avoid redundant computations.
- **Parallel Evaluation:** The fitness function evaluations were parallelized to use multiple CPU cores on the HPC cluster.

However, neither of these attempts resulted in a significant boost in the number of total evaluations performed within the time budget. This suggests that the evolutionary algorithm maintains a high level of population diversity, leading to a low cache-hit rate. Similarly, the overhead associated with process communication for parallel evaluation appeared to outweigh the benefits for the rapid, single-threaded fitness calculations.

3.6 Adaptive Grammar Weights

As a contribution to the GeneticEngine framework, we explored methods for guiding the search by dynamically adjusting the production weights of the grammar. The initial implementation, `WeightLearningStep`, updated the grammar based on the single best individual in each generation. The hypothesis was that this would reinforce the use of "good" building blocks. However, this approach often created a premature bias towards nodes found in mediocre solutions from the early generations, driving the search in the wrong direction.

To address this, the step was updated to learn from the entire Pareto front of non-dominated individuals, providing a more robust signal. Furthermore, a more sophisticated version of this step, `ConditionalWeightLearningStep`, was implemented. This step only considers individuals from the Pareto front that have also crossed a specific fitness threshold (e.g., an average score of 0.5), ensuring that the learning is based only on genuinely promising solutions.

Our experiments showed that while this adaptive grammar approach successfully outperformed the Random Search baseline, solving an average of 29.1 tasks, it did not reach the performance levels of the top-tier selection methods like Lexicase or Tournament selection (Table 1). This suggests that for ARC's rugged landscape, the implicit diversity preservation of methods like Lexicase may be more effective than an explicit grammar adaptation mechanism.

3.7 Fitness Evaluation

The design of the fitness function is a central focus of our investigation, as ARC's rugged, non-convex search space presents significant challenges. A primary issue is the prevalence of deceptive local optima. For instance, a program that simply returns the input grid often achieves a high fitness score due to pixel similarity with the target, trapping the search in a trivial solution, as illustrated in Figure 1.

To address these challenges and understand the impact of fitness landscape design, we systematically compare two distinct multi-objective fitness strategies:

1. **Pixel-Based Fitness:** This function provides a direct, but potentially sparse, evaluation signal. For each training pair, it calculates a single score based on pixel-wise accuracy. For a task with 3 training examples, this results in a 3-dimensional fitness vector. While computationally simple, this can create a "needle in a haystack" problem, where minor program changes lead to large fitness drops.

2. **Composite Fitness:** This function aims to create a smoother, more nuanced fitness landscape. For each training pair, it calculates three separate metrics: pixel accuracy, shape similarity (comparing downscaled grids), and placement similarity (comparing binarized grids). This results in a 9-dimensional fitness vector for a 3-example task. The goal is to reward partial progress, such as generating the correct shapes even if the colors are wrong, thereby providing a more informative gradient to guide the evolutionary search.

Both approaches include a specific penalty for the "identity program" to push the search towards more meaningful transformations. At the conclusion of each evolutionary run, a single program must be selected from the final population's Pareto front as the definitive solution. For this, we implemented a two-tiered selection criterion. The primary objective is to maximize the number of training examples solved perfectly (i.e., achieving a fitness score of 1.0). If multiple individuals are tied on this metric, the one with the highest average fitness score across all examples is chosen as the final solution. The heuristic behind this choice is that a program demonstrating the ability to solve at least one example completely is considered more promising than one that only achieves partial correctness across all examples. This experimental design allows for a direct analysis of the exploration-vs-exploitation trade-off in fitness engineering for this domain.

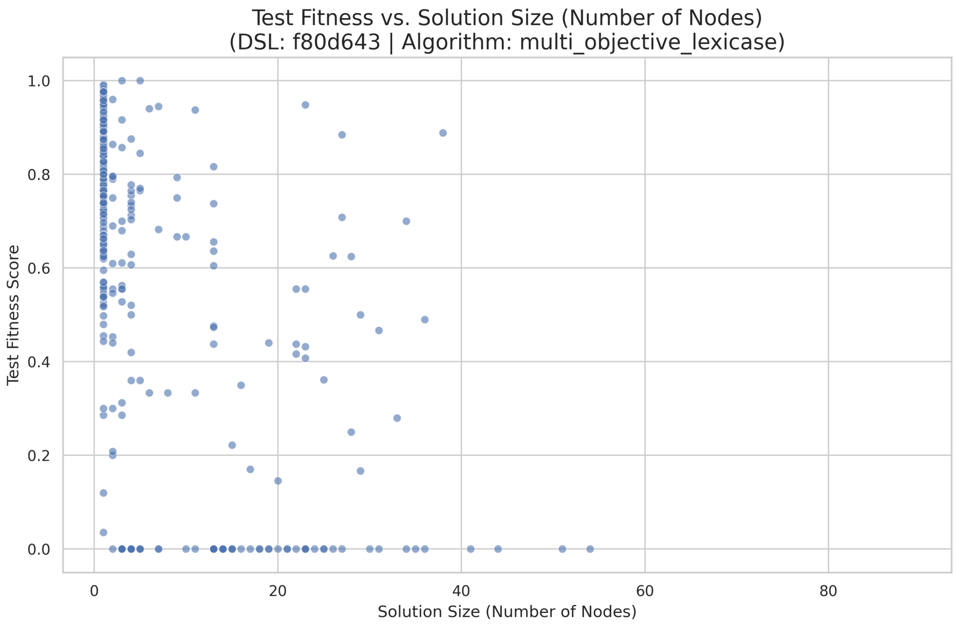

Figure 1: Test fitness vs. solution size for the Lexicase algorithm on the evaluation dataset. The dense cluster of points at a solution size of 1 represents the trivial "identity" program, which often achieves a high fitness score, creating a local optimum that can trap the search. This motivated the addition of a penalty for such solutions.

3.8 Evolutionary Algorithms

We compared several search strategies to identify the most effective for ARC's problem structure. This comparison is critical for determining how to best guide the search in a complex, multi-objective problem space.

- **Multi-Objective Genetic Programming:** The core of our approach. We treat each fitness metric for each training example as a separate objective.
- **Lexicase Selection:** A selection method that considers performance on individual objectives.

- **Informed Downsampling Selection:** A variant of Lexicase that uses a high-variance subset of objectives to speed up selection.
- **Tournament Selection:** A standard selection method where a single objective is chosen randomly each generation.
- **Crossover-only and Mutation-only GP:** To isolate the effects of the core genetic operators.
- **Random and Enumerative Search:** Used as baselines to evaluate the effectiveness of guided search.

All GP runs used a population size of 200 and a maximum tree depth of 5.

3.9 Experimental Setup

All experiments were performed on the 400 tasks from the ARC AGI 1 training dataset. The runs were executed on the High-Performance Computing (HPC) cluster at the University of Lisbon, running each configuration with 10 different random seeds for statistical robustness. To simulate competition constraints, we set a time budget of 178 seconds for each algorithm to find a solution for each individual task. The complete source code for our solver, grammar, and experiments is publicly available for review and reproduction [11].

4 Results and Analysis

Our experiments provide a clear picture of the performance trade-offs between different evolutionary configurations. A numerical summary of the overall performance is presented in Table 1, while the performance distributions and task overlaps are visualized in Figures 2 and 3.

Table 1: Overall Performance Summary of Different Solver Configurations

Algorithm	**Avg. Tasks Solved**	**Avg. Test Fitness**	**Avg. Evaluations**	**Avg. Solution Size**
Epsilon Lexicase	33.9 ± 2.3	0.60	41,451	13.2
Lexicase Pixel	33.2 ± 2.0	0.60	40,432	13.1
Tournament	33.1 ± 2.5	0.59	41,725	13.3
Conditional Weight Learning	29.1 ± 2.6	0.59	31,613	13.3
Random Search	27.7 ± 1.9	0.44	87,567	13.5
Lexicase Composite	23.1 ± 2.9	0.55	10,921	13.0

4.1 Overall Performance Distribution

Figure 2 shows the distribution of the number of tasks solved per run for each algorithm across 10 seeds. A key finding is that guided evolutionary search significantly outperforms the Random Search baseline. Epsilon Lexicase, Lexicase Selection, and Tournament Selection, when paired with a simple pixel-based fitness function, all demonstrate superior performance, solving an average of 33-34 tasks compared to 27.7 for Random Search. This result validates the effectiveness of GP for this problem, showing that an evolutionary approach can navigate the vast search space more efficiently than undirected exploration.

The analysis also reveals a critical trade-off in fitness design, as shown in Table 1. The Lexicase Composite configuration performs the worst, solving only 23.1 tasks on average. The reason is evident in the evaluation count: its complex fitness function is computationally expensive, allowing for only $\sim 11,000$ evaluations within the time budget. In contrast, the pixel-based methods perform nearly four times as many evaluations ($\sim 41,000$). This suggests that for ARC, the volume of exploration enabled by a fast, simple fitness function is more beneficial than the nuanced guidance of a more complex one.

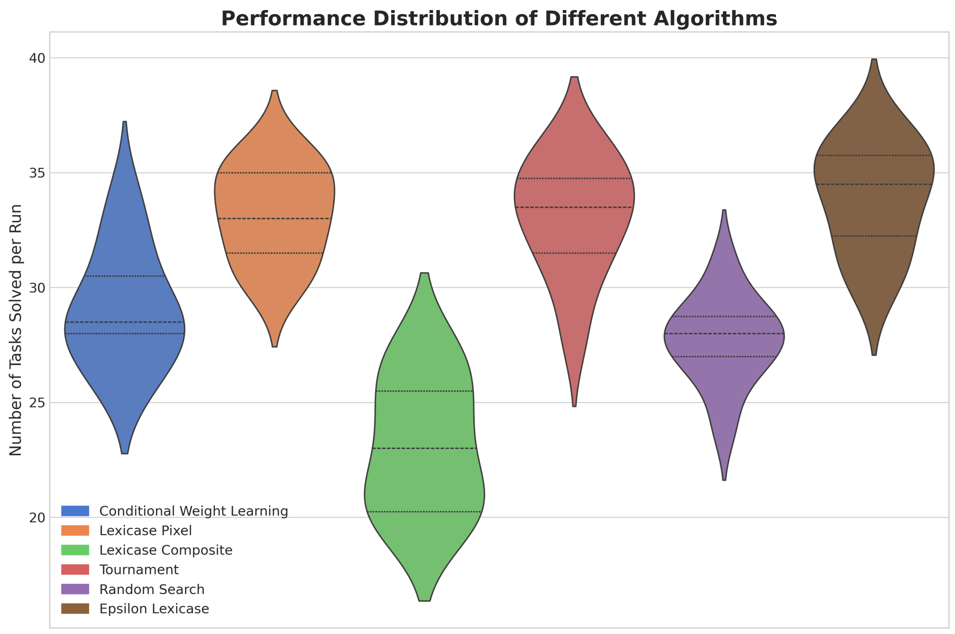

Figure 2: Performance distribution of the different solver configurations across 10 seeds. The width of each violin represents the density of results at that performance level. Lexicase Pixel, Epsilon Lexicase, and Tournament selection all show superior median performance compared to Random Search, Conditional Weight Learning, and Lexicase Composite.

4.2 Complementary Strengths and Task Overlap

Figure 3 provides a deeper insight into the behavior of the different algorithms by visualizing the overlap of the specific tasks they solved. The plot reveals two crucial points. First, there is a large core set of 32 tasks that were solved by all six algorithms, representing a baseline of commonly solvable problems.

Second, and more importantly, the plot shows that the top-performing algorithms solve distinct subsets of the more difficult tasks. Both Lexicase Pixel and Tournament Selection solved the highest number of unique tasks (60 each). However, the intersections between them are not total. For instance, the second-largest intersection (7 tasks) is shared by all algorithms except Lexicase Composite, while another set of 4 tasks is solved only by Lexicase, Epsilon Lexicase, Tournament, and Conditional Weight Learning. This lack of complete overlap suggests that the different selection pressures of distinct algorithms lead them to explore different regions of the search space, giving them complementary strengths. This is a powerful argument for an ensemble approach, where the solutions from multiple different algorithms are combined. The total number of unique tasks solved by such an ensemble would be significantly higher than any individual solver's score.

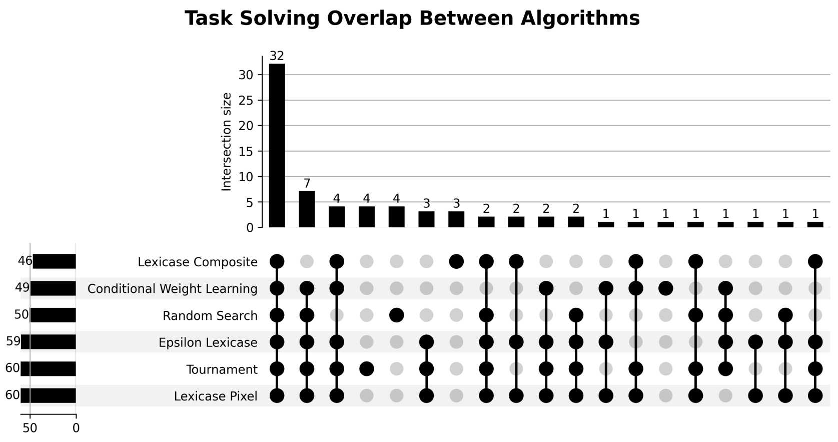

Figure 3: Upset plot showing the intersection of tasks solved by each algorithm. The top bars show the number of tasks in each intersection, while the left bars show the total number of unique tasks solved by each algorithm. The large bar on the far left indicates 32 tasks were solved by all methods. Lexicase Pixel and Tournament selection solved the most tasks overall (60 each).

4.3 Impact of Genetic Operators

To isolate the effects of the core genetic operators, we conducted an ablation study comparing different probabilities of crossover and mutation for the Lexicase algorithm. The results, shown in Figure 4, reveal that a mutation-only strategy (100% mutation, 0% crossover) achieves the highest median performance and is the most consistent. As the probability of crossover increases, performance tends to become lower and more variable. The configuration with 75% crossover, for instance, shows the widest distribution, indicating a high-risk, high-reward strategy that is unreliable.

This suggests that for ARC, crossover can be a disruptive operator, potentially breaking apart the useful, semantically-cohesive building blocks discovered by the search. However, it is important to note that the top-performing configuration in this isolated test (100% mutation) did not perform significantly better than the standard high-crossover configuration (80% crossover, 15% mutation) used for the main algorithm comparisons. The median performance of the mutation-only approach is comparable to the average of 33.9 tasks solved by Epsilon Lexicase in Table 1, validating our choice of a standard operator setup for the main experiments.

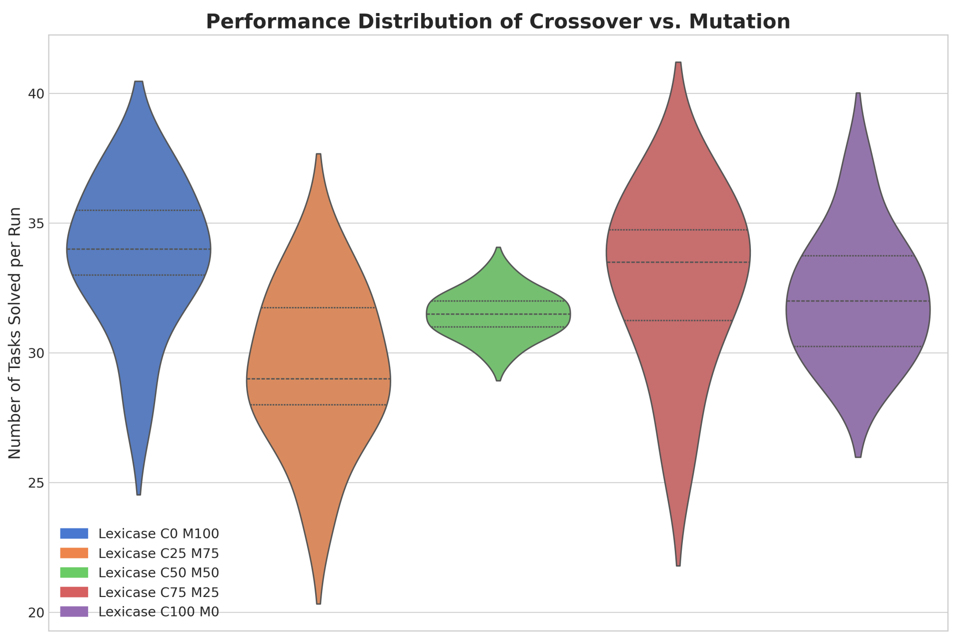

Figure 4: Performance distribution of the Lexicase algorithm with different crossover (C) and mutation (M) probabilities. The results show that a mutation-only approach (C0 M100) yields the best performance.

5 Future Work

The findings from this paper suggest promising directions for future research in the field, focusing on improving the core components of the symbolic search engine. The work presented here could be further expanded in the following directions:

- **Improving the Fitness Function:** Our results highlight the critical challenge of fitness engineering. Future work could move beyond pixel-based comparisons towards more abstract evaluation metrics. For instance, a pre-trained Convolutional Neural Network (CNN) could be used to measure the "perceptual distance" between two grids, providing a richer, learned similarity metric that might create a smoother and more informative fitness landscape.

- **Automating DSL Improvement:** The current DSL, while expressive, is manually designed, which introduces human bias and is a significant bottleneck. A major avenue for future research is the automatic discovery or evolution of DSL primitives. Techniques like library learning, as seen in systems like DreamCoder [10], could be used to identify common sub-routines in successful programs and add them to the DSL as new, higher-level functions.

- **Changing the Problem Representation:** The current approach operates directly on raw pixel grids. A more powerful approach could involve changing the representation to a more structured format. For example, an ARC grid can be represented as a multi-level graph, where nodes at the lowest level are pixels, which are grouped into objects at the next level, which in turn form relationships with other objects. Evolving programs that operate on this graph structure, rather than the raw grid, could allow the system to learn more abstract and relational concepts.

6 Conclusion

The ARC research landscape has evolved rapidly, with state-of-the-art systems now relying on ensembles of neural and symbolic methods. This paper contributes to this new context by evaluating Genetic Programming as a symbolic search engine. Our systematic comparison of evolutionary configurations reveals key insights. First, guided search methods like Lexicase and Tournament selection, when paired with a computationally efficient pixel-based fitness function, significantly outperform a Random Search baseline. Second, we identified a critical trade-off in fitness function design, where a simpler, faster function yielded better results than a more complex one due to the ability to perform more evaluations within a fixed time budget. Finally, our analysis shows that the top-performing algorithms solve distinct subsets of tasks, suggesting that an ensemble of different solvers is a promising path towards higher performance on the ARC challenge.

References

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

[2] ARC Prize, *ARC Prize 2024 Technical Report*, `https://arcprize.org/blog/2024/06/03/arc-prize-2024-technical-report/`, 2024.

[3] A. Fonseca, et al., *GeneticEngine*, GitHub repository, `https://github.com/alcides/GeneticEngine`.

[4] M. Hodel, *arc-dsl*, GitHub repository, `https://github.com/michaelhodel/arc-dsl`.

[5] J. Wind, *ARC-solution Documentation*, Technical Documentation, `https://github.com/top-quarks/ARC-solution/blob/master/ARC-solution_documentation.pdf`, 2020.

[6] M. Fischer, M. Tenorth, and M. Beetz, *Solving Abstract Reasoning Tasks with Grammatical Evolution*, In Proceedings of the LWDA 2020 Conference, 2020.

[7] A. Franzen, et al., *The ARChitects' Submission to the ARC Prize 2024*, `https://arcprize.org/blog/2024/06/03/the-architects-technical-report/`, 2024.

[8] E. Akyürek, et al., *Combining Induction and Transduction for Abstract Reasoning*, arXiv preprint arXiv:2405.15410, 2024.

[9] R. Greenblatt, *Solving ARC-AGI with GPT-4*, Personal Blog, `https://redwoodresearch.substack.com/p/getting-50-sota-on-arc-agi-with-gpt`, 2024.

[10] K. Ellis, et al., *DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning*, In Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2021.

[11] P. Silva, *ARC-Solver: A Genetic Programming Approach to the Abstraction and Reasoning Corpus*, GitHub repository, `https://github.com/PauloHS-Silva/ARC-Solver`, 2024.