POLITECNICO DI TORINO

POLITECNICO DI TORINO

Master Degree course in Computer Engineering

Master Degree Thesis

Design and development of program synthesis approaches to improve the generality of artificial intelligence.

Supervisors
Prof. Giovanni Squillero
Prof. Alberto Tonda

Candidate
Roberto Montesanto

Academic Year 2024-2025

Acknowledgements

At the end of this 5-year journey, I'd like to thank all the people that helped and supported me. Without them, I would probably have never been able to reach this important goal.

I'd like to first thank my parents, for always believing in me and pushing me to always do my best. I hope they are proud of what I managed to achieve.
I would also like to thank my friends and colleagues, for accompanying me during this journey, sharing the sorrows and joys of our academic career.

And finally, I'd like to thank professors Giovanni Squillero and Alberto Tonda, for their support and help in writing this thesis.

Thank you everyone

Abstract

The Abstraction and Reasoning Corpus (ARC) has emerged as a key benchmark for evaluating progress toward Artificial General Intelligence (AGI), as it emphasizes flexible problem solving and generalization beyond narrow, domain-specific methods. This thesis investigates the application of Genetic Programming (GP) to the ARC framework, with the aim of exploring the feasibility of evolutionary search as a path toward generalizable reasoning systems.

In this work, we describe our attempt to use the Byron fuzzer (Byron: A Fuzzer for Turing-complete Test Programs, 2024) to tackle ARC tasks, focusing on fitness evaluation, transformation functions and program structure. We analyze the performance of the system on different ARC challenges, highlighting its potential and limitations. The results provide insights into the role of evolutionary computation in AGI research and suggest avenues for new approaches that could make better use of the Byron framework. Ultimately, the thesis contributes to understanding how evolutionary search mechanisms can support progress toward more general, adaptable artificial intelligence.

Contents

Chapter 1

Introduction

Recent advances in the field of Artificial Intelligence (AI) have produced highly effective systems in specialized domains such as computer vision, natural language processing, and reinforcement learning. However, these systems are still limited to narrow tasks and lack the ability to flexibly generalize across different domains. The pursuit of Artificial General Intelligence (AGI) aims to overcome this limitation by developing systems capable of abstraction, reasoning, and problem solving in previously unseen settings [**?**].

The Abstraction and Reasoning Corpus (ARC), introduced by François Chollet (2019), has been proposed as a benchmark to evaluate progress toward AGI [1]. Unlike more traditional machine learning datasets, ARC presents tasks explicitly designed to assess a system's ability to infer generalizable rules from a small number of examples.

Current machine learning approaches face significant difficulties when applied to ARC. Data-driven statistical models, including deep learning architectures, generally fail to generalize when exposed to the limited number of examples provided for each task [2] [**?**].
Symbolic and program synthesis approaches, while capable of producing exact and interpretable solutions, often suffer from inefficiency due to the combinatorial growth of the search space [3].
Hybrid approaches that combine neural and symbolic methods have been investigated [4] [5], but their effectiveness remains limited to subsets of the benchmark.

This context motivates the investigation of evolutionary computation as an alternative paradigm for addressing ARC. Evolutionary algorithms (EAs) are population-based search methods inspired by natural selection [6], and are better suited to exploring large, discontinuous search spaces. Among these, Genetic Programming (GP) provides a direct mechanism for evolving executable programs expressed as compositional structures [7].
The ease of interpretation and modularity of GP make it particularly relevant to tasks such as ARC, where solutions can be naturally represented as transformations of symbolic structures.

This thesis examines the application of Genetic Programming to ARC as a means of assessing its suitability for tasks that demand generalization and abstraction. Particularly, the goal of the thesis work was to adapt the Byron fuzzer [8] to solve ARC tasks. Through experimental evaluation, the thesis aims to identify the strengths and limitations of this approach and to provide insights into the potential role of evolutionary computation in advancing AGI research.

Chapter 2

Introduction to ARC-AGI

The Abstraction and Reasoning Corpus (ARC) is a benchmark introduced by François Chollet in 2019 as a means of evaluating progress toward Artificial General Intelligence (AGI) [1]. While the majority of machine learning datasets focus on large-scale pattern recognition within a fixed domain (e.g., images, text, or speech), ARC is explicitly designed to test a systems ability to generalize to novel tasks using only a small number of examples.

As such, ARC plays a dual role. On one hand, it is a diagnostic tool, revealing the limitations of contemporary deep learning approaches. On the other, it is a research catalyst, motivating the exploration of alternative paradigms such as symbolic reasoning, program synthesis, evolutionary search, and hybrid models that aim to bridge the gap between narrow AI and more general, human-like intelligence.

2.1 Task structure

ARC is composed of a large collection of small tasks, each defined by a set of input-output examples. The tasks are presented on grids of colored cells, with a discrete number of possible colors, and the goal is to learn the underlying transformation that maps input to output. Examples of transformations include:

- Recoloring or reshaping objects
- Identifying and applying symmetries
- Counting and/or repeating patterns
- Applying compositional operations (e.g., reflection followed by translation)

Each task includes only a handful of demonstrations, typically three to five, which require the solver to infer the mapping rule and apply it to unseen test cases. Critically, the tasks are deliberately diverse and heterogeneous, making it difficult to approach them with predefined heuristics or pattern-matching strategies.

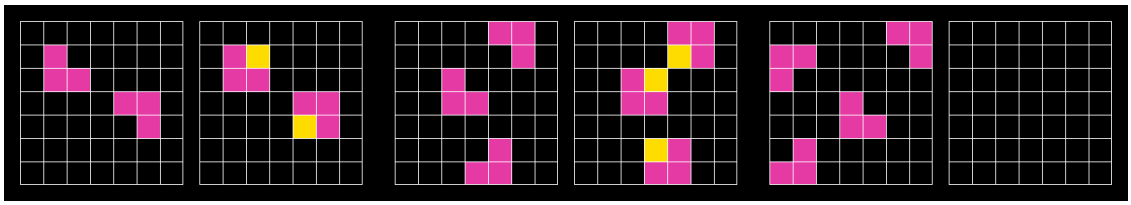


Figure 2.1: Example of an ARC task

2.2 Importance for Machine Learning

ARC challenges many of the assumptions underlying contemporary machine learning. Most successful AI systems are based on large-scale training data, statistical generalization, and optimization of task-specific objectives.
In direct contrast to these principles, ARC demands the ability to:

- Generalize from very few examples
- Discover and represent abstract concepts
- Adapt flexibly to entirely novel tasks

These requirements align more closely with human cognition than with conventional supervised learning. A human can generally grasp a new rule or pattern after seeing only one or two examples, whereas even the most advanced neural networks often fail in such situations.

2.3 Challenges and open questions

Working with ARC also introduces several challenges that remain largely unsolved:

- Representation: How should the abstract information presented in an image be encoded to be correctly interpreted during the rule inference process?
- Search and inference: What mechanisms enable the discovery of a correct rule given such limited evidence?
- Transferability: Can solutions learned from one task inform the solving of another, despite their differences on a surface level? And if so, how can they affect the learning process?
- Evaluation: To what extent does performance on ARC truly measure progress toward AGI, as opposed to specialized heuristics tailored to its structure?

These challenges illustrate why ARC has become a central benchmark in AGI research and why it continues to attract interest from multiple communities, including machine learning, cognitive science, and computational logic [9].

2.4 Current State-of-the-Art approaches and performance

The ARC Prize website has a publicly available leaderboard that shows the performance of different models [10]. Critically, all the models presented in the leaderboard are Large Language Models (LLMs).

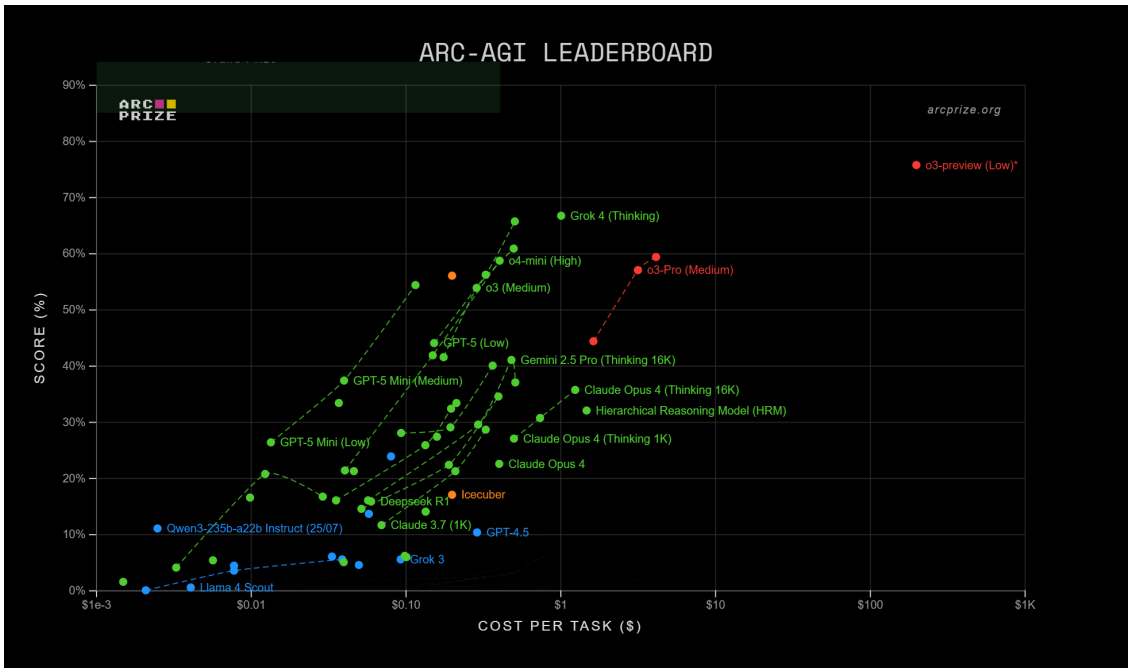


Figure 2.2: ARC leaderboard

The main models used to challenge ARC use Chain of Thought (CoT, colored green in the image) prompting, a prompt engineering technique that enhances the output of LLMs by guiding the model through a step-by-step reasoning process, using a coherent series of logical steps.
These models include GPT (by OpenAI), Gemini (by Google), and Grok (by xAI). Grok, in particular, is the best-performing model, with an accuracy of 66.7%.
These models present a critical flaw, though: as their performance increases, so does the maintenance cost, up to $4.16/task for OpenAI's o3 model.

Chapter 3

Evolutionary algorithms and Genetic Programming

Other than Large Language Models, some approaches to the ARC-AGI challenge utilize a category of algorithms known as Evolutionary Algorithms (EAs).

3.1 What are Evolutionary Algorithms?

Evolutionary Algorithms are a family of algorithms inspired by the principles of natural selection and biological evolution. Instead of directly searching for a solution, EAs maintain a population of candidate solutions (called individuals), which are evolved iteratively through processes analogous to mutation, selection, and inheritance. [6]
The guiding principle is that, over many generations, fitter individuals (better solutions) accumulate in the population, gradually producing solutions that adapt to the task environment.

An EA will generally initialize a population of random individuals, then follow an iterative process like this:

1. Evaluate the fitness of each individual in the population.
2. Check if a termination criterion is reached, and terminate the algorithm if so.
3. Select a number of individuals (preferably of higher fitness) as parents.
4. Produce offspring with optional crossover, in order to mimick reproduction.
5. Apply mutation operations on the offspring.
6. Select individuals (preferably of lower fitness) for replacement with new individuals (mimicking natural selection).

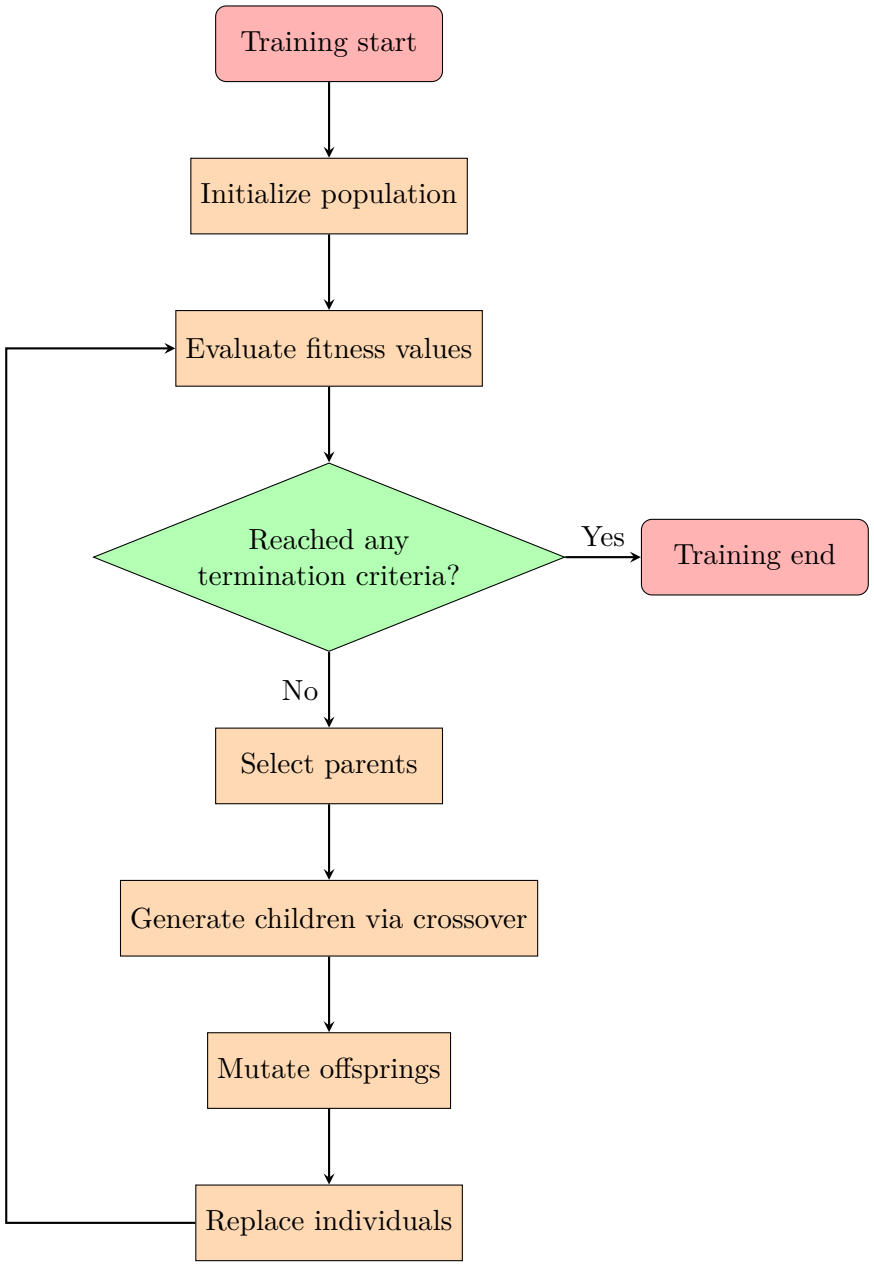


Figure 3.1: Training loop of a generic EA

As such, the key components a programmer has to define in order to create an Evolutionary Algorithm are the following:

3.1.1 Representation

The first thing to define is the algorithm's representation of candidate solutions, which defines how individuals are encoded within the algorithm. This can generally take the form of:

- Bit-strings, where each bit encodes a feature or parameter.
- Real-valued vectors, most often used in continuous optimization problems.
- Trees or expression graphs (most commonly used in genetic programming), where nodes represent functions or primitives, and leaves represent inputs or constants.
- Specialized data structures tailored to the problem domain (e.g., neural network architectures, symbolic rules, or grid transformations).

The choice of representation directly influences the search space, as well as the types of operators that can be applied.

3.1.2 Fitness function

The fitness function is used to measure the quality of a candidate solution. It acts as the guiding signal of the evolutionary process, determining which individuals are more likely to survive and reproduce.

A well-designed fitness function should take into account some key considerations:

- Task relevance: the metric must reflect how well the candidate solves the problem.
- Granularity: the function should provide meaningful intermediate feedback, instead of being strictly binary (correct/incorrect). This helps in creating a search space in which even smaller differences between individuals can be appreciated, simplifying the optimization process.
- Generalization: as with the majority of other Machine Learning algorithms, evaluation should discourage overfitting.

3.1.3 Operators

Operators are responsible for generating diversity in the population. There are two main types of operators: mutation operators and crossover operators.

Mutation operators are applied to a single individual and are used to modify it in a random manner. Examples include flipping bits in a string, altering numerical parameters, or replacing subtrees in a program. These operators tend to promote exploration, generating a set of diverse individuals to better explore the search space.

Crossover operators combine parts of two parent individuals to produce an offspring one. In genetic programming, for instance, crossover might exchange entire subtrees between two programs. Crossover promotes exploitation, allowing useful elements present in an individual to spread through the population.

Effective design and use of these operators require balancing exploration and exploitation: too much randomness leads to an unstructured search, while too little may trap the population in a local fitness well (where fitness does not improve unless a much different individual is proposed).

3.1.4 Selection mechanisms

Selection determines which individuals are retained for reproduction (crossover) and which are discarded.
Commonly used selection mechanisms are:

- Fitness-proportionate selection (also called roulette wheel selection), where the probability of selection is proportional to fitness.
- Tournament selection, where individuals compete in small groups and the fittest of each group is chosen to perform crossover.
- Rank-based selection, where individuals are ordered by fitness and selection probability depends on the resulting rank instead of the fitness value itself.

Selection introduces evolutionary pressure, steering the population toward fitter solutions. However, excessive pressure can lead to premature convergence, where the population diversity is lost and the algorithm stagnates. Conversely, too little pressure can result in an inefficient search, with little diversity and a slow adaptation to the task environment.

3.1.5 Termination criteria

The final element to define is a rule for when to stop the algorithm, since otherwise it could theoretically continue to loop indefinitely.
Generally, the termination criteria of an EA include reaching a certain number of generations or evaluations, discovering a solution that meets or exceeds a certain fitness threshold, or observing a stagnation in fitness improvement or diversity.

3.2 Program Synthesis

Program synthesis is a term used in Computer Science that defines the task of automatically constructing computer programs satisfying a given specification [11]. Unlike traditional programming, where a human explicitly writes the code, program synthesis aims to generate the code automatically from constraints such as examples, formal rules, or natural language descriptions.

The main motivation behind the pursuit of Program Synthesis is automation of reasoning: enabling computers to generate procedures for solving tasks without human intervention could create more time for programmers and engineers to debug and fix other critical systems.
In addition, allowing users to specify intent at a high level (through demonstrations or rules) while the system handles the low-level implementation details could allow even users with less technical knowledge to create programs and apps to help them and other in their communities.

The types of specification a program has to satisfy may vary. Generally, though, Program Synthesis is performed under three main types of constraints:

- Example-based synthesis (Programming by Example, PBE) enables the system to generate a program consistent with a small number of input-output pairs. ARC is a prime example of this mode.

- In constraint-based synthesis programs are synthesized to satisfy either logical or mathematical constraints (e.g., SAT or SMT solvers).

- Natural language-based synthesis instead generates programs starting from textual instructions or queries. This is a research area that is being increasingly explored with LLMs.

These constraints are not mutually exclusive either, in fact there exist hybrid approaches that combine them for greater robustness.

3.3 Genetic Programming

Genetic Programming (GP) is an approach to Program Synthesis that uses Evolutionary Algorithms to create a program [7].
The central idea is that, given a well-defined set of primitives (functions, operators, and terminals), a GP system can discover compositions of these primitives that solve a given task. Over successive generations, populations of candidate programs evolve following a normal EA training loop, until a well-defined and (hopefully) functioning program is generated.

3.3.1 Syntax Trees

Individuals in a GP algorithm are traditionally represented as tree structures, with every internal node representing an operator function and every leaf node representing an operand (a variable or a constant). This allows the tree to be easily evaluated in a recursive manner.

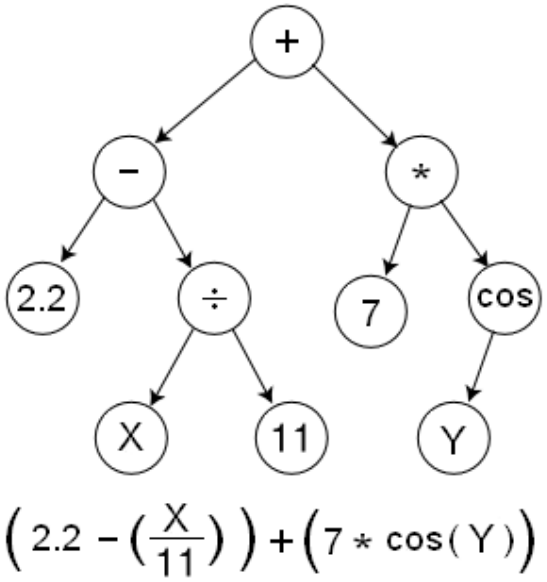


Figure 3.2: Example of a mathematical function represented as a tree

This traditional representation favors the generation of programs written in functional programming languages (such as Lisp), as they naturally embody tree structures.
Other solutions (and forms of representation) have been successfully implemented to work with more traditional imperative languages.

3.3.2 Sub-tree crossover

In traditionally tree-represented GP algorithms, crossover is generally performed by swapping sub-trees of the parents. This particular crossover operation is performed following a series of steps:

1. Choose a random sub-tree from each parent individual.

2. One of the parents is selected as the root donating parent. This parent has its selected sub-tree removed and replaced with the other parent's subtree.

3. If two child crossover is used, meaning the crossover operation generates two children instead of one, the same operation is performed on the other parent, removing its sub-tree and replacing with the root donating parent's.

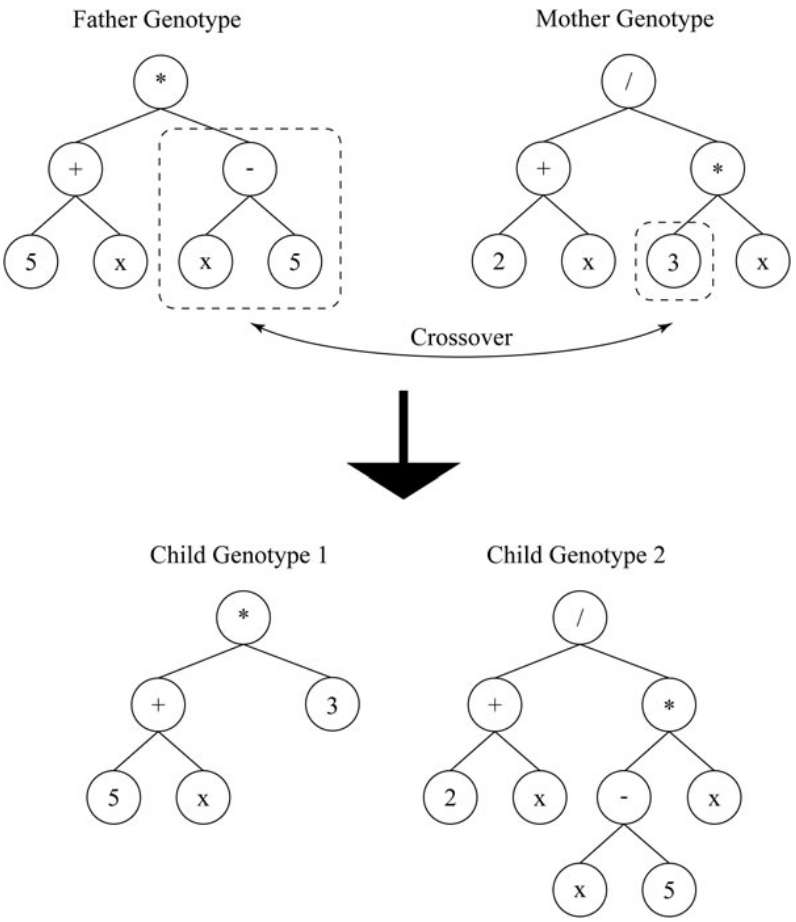
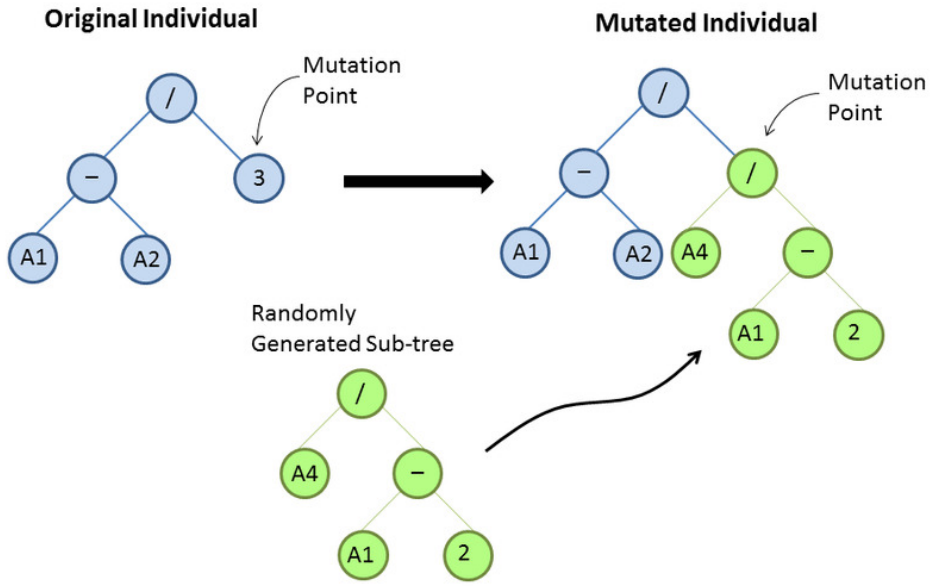


Figure 3.3: Example of sub-tree crossover

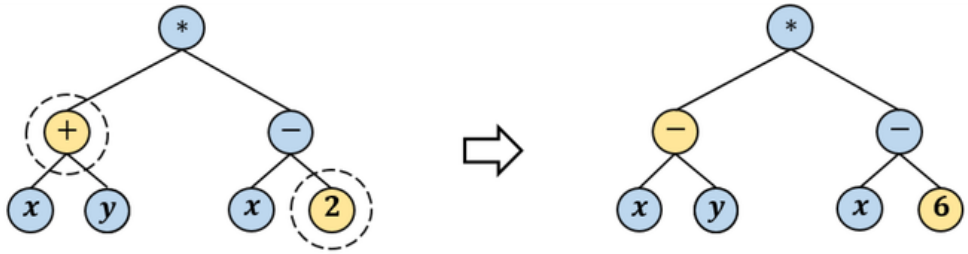
3.3.3 Tree mutation

There are different types of mutation operators that can be applied to tree structures. They all aim to mutate a syntactically correct individual, generating a new one, still syntactically correct.
There are four commonly used operators:

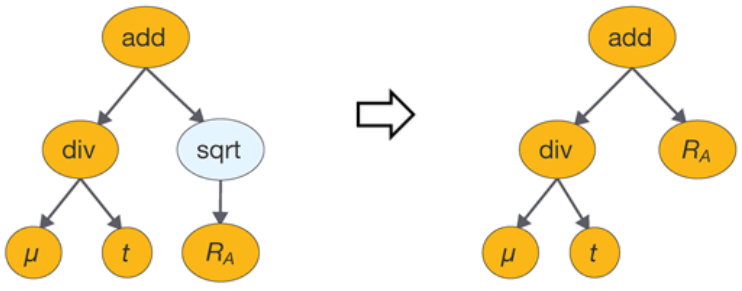
- Sub-tree mutation replaces a randomly selected sub-tree with a new randomly generated one. Naturally, the sub-tree generation must be programmed to always generate syntactically correct sub-trees.
- Point mutation replaces an individual node with another of the same type. A leaf node (variable or constant) will be replaced with another leaf node, while an internal node (function) will be replaced with another function of the same arity (number of inputs).
- Hoist mutation replaces a randomly chosen sub-tree with another randomly selected sub-tree within itself, thus guaranteeing to make the tree smaller.



(a) Subtree mutation



(b) Point mutation



(c) Hoist mutation

Figure 3.4: Examples of each mutation operator

3.4 Fuzzing

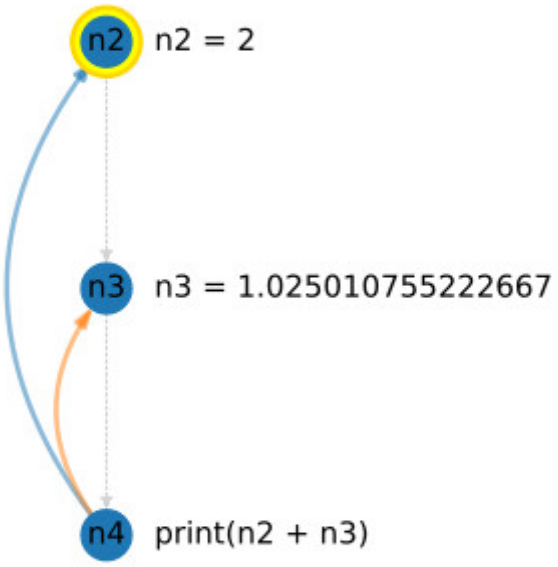
Fuzzing (or fuzz testing) is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program, then monitoring the program for exceptions such as crashes, failing built-in code assertions, or potential memory leaks [12].
Typically, fuzzers are used to test programs that take structured inputs. This structure is specified and is used to distinguish valid inputs from invalid ones. An effective fuzzer generates semi-valid inputs that are not directly rejected by the parser, but will create unexpected behaviors deeper in the program and are able to expose corner cases that have not been properly dealt with.

3.5 The Byron fuzzer

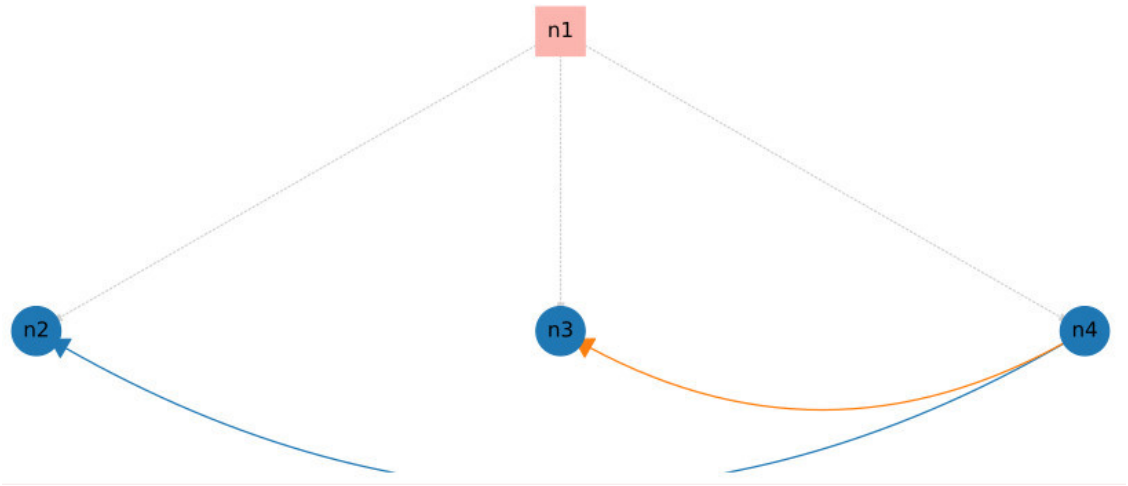
Byron is a source code fuzzer designed to support either assembly or higher level languages [8]. It works by generating a set of random programs, which are then iteratively improved by an evolutionary algorithm.
As such, it can also be adapted for Genetic Programming tasks, at least those for which a fitness function can be appropriately designed. In this project we used it to try to generate a Python program capable of solving ARC tasks.

Internally, it encodes candidate solutions as typed, directed multi-graphs, and can effectively handle complex, realistic program structures containing local and global variables, conditional and looping statements, and subroutines.

Candidate programs can be evaluated using a user-defined Python function or an external tool, such as an interpreter or a simulator. The framework also supports different types of parallelization out of the box, from simple multi-threading to the creation of temporary directories where multiple sub-processes are concurrently spawned.



(a) Generated code sequence



(b) Multigraph

Figure 3.5: Example of a simple Python script generated using Byron, and its corresponding graph

3.5.1 Program structure

Any candidate program in Byron is a composition of Macros and Frames.

Macros

Macros are essentially the building blocks used in the program generation. They represent fragments of code, with optional parameters. In the example used for Fig. 3.5, three different macros were created: two containing snippets for creating numeric variables (`n2` and `n3`), and one containing the code to print their sum (`n4`).

The parameters are objects that encode a value. They are used to add variability to the candidate programs, and their values are updated at each iteration of the evolution process.
There are different types of parameters:

- Integer and float parameters encode an integer and a floating-point value, respectively.
- Choice parameters encode a single value chosen at random from a list of possible elements.
- Array parameters encode a sequence of values, chosen at random from a list of possible elements, and with a specified length.

Macros can also use references, a special kind of parameter that is used to reference another point in the program. Specifically, a local reference points to another random macro in the same bunch frame (see below, in **Frames**), while a global reference points to a specific macro, independently of its position in the program.
Local references can be used mainly to create assembly jump instructions, while global references can be used to create variables. For example, in Fig. 3.5, the macro `n4` uses two global references to access the macros `n2` and `n3`, which created variables named after themselves.

Frames

Frames are used to contain and join together different macros or other frames, and act as parent nodes in the multi-graph, connected to each of their related macros or frames. In the example used for Fig. 3.5, the node `n1` is a frame containing the three macros used in the program.
When evaluating a frame, the child macros will be evaluated differently based on the frame's type:

- Sequence frames simply arrange its child macros in a sequential manner and evaluates them in order.

- Bunch frames will first generate a sequence of macros, randomly selected from their children, then evaluate them in order similarly to sequence macros.
- Alternative frames randomly select a single macro from their children, and the selected macro will be the only one to be evaluated.

3.5.2 Fitness functions

The Byron framework accepts user-defined fitness functions to use in its evolution process, but accepts only numeric or vector-based fitness values. These two types are compared differently.

Comparison between numeric fitness values, either integer or floating-point, is a simple algebraic difference, but additional parameters can be specified for floating-point values to define whether two individuals have comparable fitness.
Comparison between vector-based fitness values, on the other hand, is performed following a lexicographic order: the comparison result is given by the values in the first position of the vector; if they are the same, then the values at the second position are checked, then if they are also the same the third position is compared and so on. If all pairings are equal, then the two vectors are considered equal as well.

Chapter 4

Experimental setup

This chapter will present our experimental setup, with particular attention to the fitness function used to compare candidate programs, the operators designed to extrapolate data from the input image and use it on the generated output, and the way the program structure was set up (that is, the Macros and Frames used).

The model was trained on a set of 400 ARC tasks.
All images in the tasks were encoded as Numpy 2D integer arrays (bibref). Each element ranges between 0 and 10, and corresponds to a different colored pixel.

4.1 Fitness

The first thing that needed to be defined was a fitness function, used to encode the performance of a given candidate solution on a given ARC task. This fitness function would take a generated image and the corresponding expected output and return a measure of the difference between the latter and the former.

The first intuition was to simply define a pixel-correctness fitness, where the performance would be measured in terms of percentage of correct pixels in the generated image, compared to the expected output. The idea is to have a continuous measure to better incentivize the evolution process.
However, this fitness proved to be, at least partially, ill-designed, because there are a number of images where the majority of pixels are background color, and the actual task focuses on a lower number of colored pixels. Thus, the function was improved by ignoring the pixels that made up at least 50% of the image.
As an example of the measure provided by this function, consider Fig.4.1. Taking into account background pixels (black in this case), the fitness value would be $8/9 \approx 88\%$. Ignoring them, the fitness value drops to $2/3 \approx 66\%$, more representative of the actual difference between the images.

Another intuition came when designing the selectors (more on that in section 4.3). Since each selector captured a section of the image in the form of a list of pixels, a new

fitness function was designed to capture the difference in these selections. This fitness would then be minimized in the evolution process, instead of being maximized.
The idea behind this measure is to have a quantifiable difference between relevant features in the generated image and the expected output image, instead of a simple measure of the number of correct pixels.
The fitness would first extrapolate all the possible selections in both images, then compute a numeric difference between the results. Then these differences were placed in sequence, in random order. The difference was calculated by the following algorithm:

1. Since any given selector could extract a different number of pixels in the two images, the first thing to check is whether the number of selected pixels in the images is the same.

2. If true, then calculate the average Manhattan distance between the selected pixels.

3. Otherwise, the difference is computed as the absolute difference in the number of selected pixels, scaled by a factor of 100.

Taking again Fig.4.1 as an example, the color selectors `Red` and `Yellow` would produce a difference of 100.0 (since a single pixel would be selected in one image while none in the other), while the color selector `Grey` would produce a value of 0.0. Any other selector would produce a value of 0.0 as well, as no pixels would be extracted. All these values would then be put into a sequence in random order, and two different fitness sequences would be compared in lexicographic order.

(a) Generated image (b) Expected output image

Figure 4.1: Example of a generated image and its corresponding expected output

4.2 Operator functions

The next set of functions we defined are functions that operate on an image, applying different transformations on it:

- `connect_pixels(im, px1, px2, color, left_connect_px = None)`: this function takes an image and two pixels as input, and draws a straight line between them, of the defined color.

  If the pixels are not aligned (i.e. they have different `x` and `y` coordinates), two lines are drawn, connecting the pixels at a right angle following taxicab geometry.
  The optional parameter `left_connect_px` can be used in this case to specify which of the two pixels will have the connecting line start from its left.

- `color_selection(im, pixels, color, offset = (0, 0))`: this functions takes an image and a list of pixel positions and sets all the specified pixels to the specified color.

  The optional parameter `offset` can be used to set the color of pixels that are near the specified section.

- `copy(im, out, pixels, offset = (0, 0), cut = True)`: this function takes two images and a list of pixel positions as input, and tries to copy as many of the specified pixels from the `im` image to the `out` one. Any pixel that would end up outside the bounds of the `out` image is discarded.

  The optional parameter `offset` can be used to specify an offset vector to use during the copy process.
  The optional parameter `cut` can be set to `True` to remove the pixels from the `im` image during the copy process.

- `flip(out, pixels, pivot = (-1, -1), flip_x = False, flip_y = False)`: this function takes an image and a list of pixel positions as input, and tries to generate a symmetric copy of the specified section of the image while leaving the rest of it unaltered. Any pixel that would be out of the bounds of the image is discared.
  The parameters `flip_x` and `flip_y` are used to specify the axis/axes of symmetry to use.

  The optional parameter `pivot` can be used to specify the origin point used for the transformation.

- `rotate_90(out, pixels, times = 1, pivot = (-1, -1)`: this function takes an image and a list of pixel positions as input, and tries to generate a copy of the image with the selected section rotated 90 degrees anti-clockwise, while the rest of the

image is left unaltered. Any pixel that would be out of the bounds of the image during the rotation process is discarded.

The optional parameter `times` can be used to specify the number of rotations. The optional parameter `pivot` can be used to specify the center of rotation.

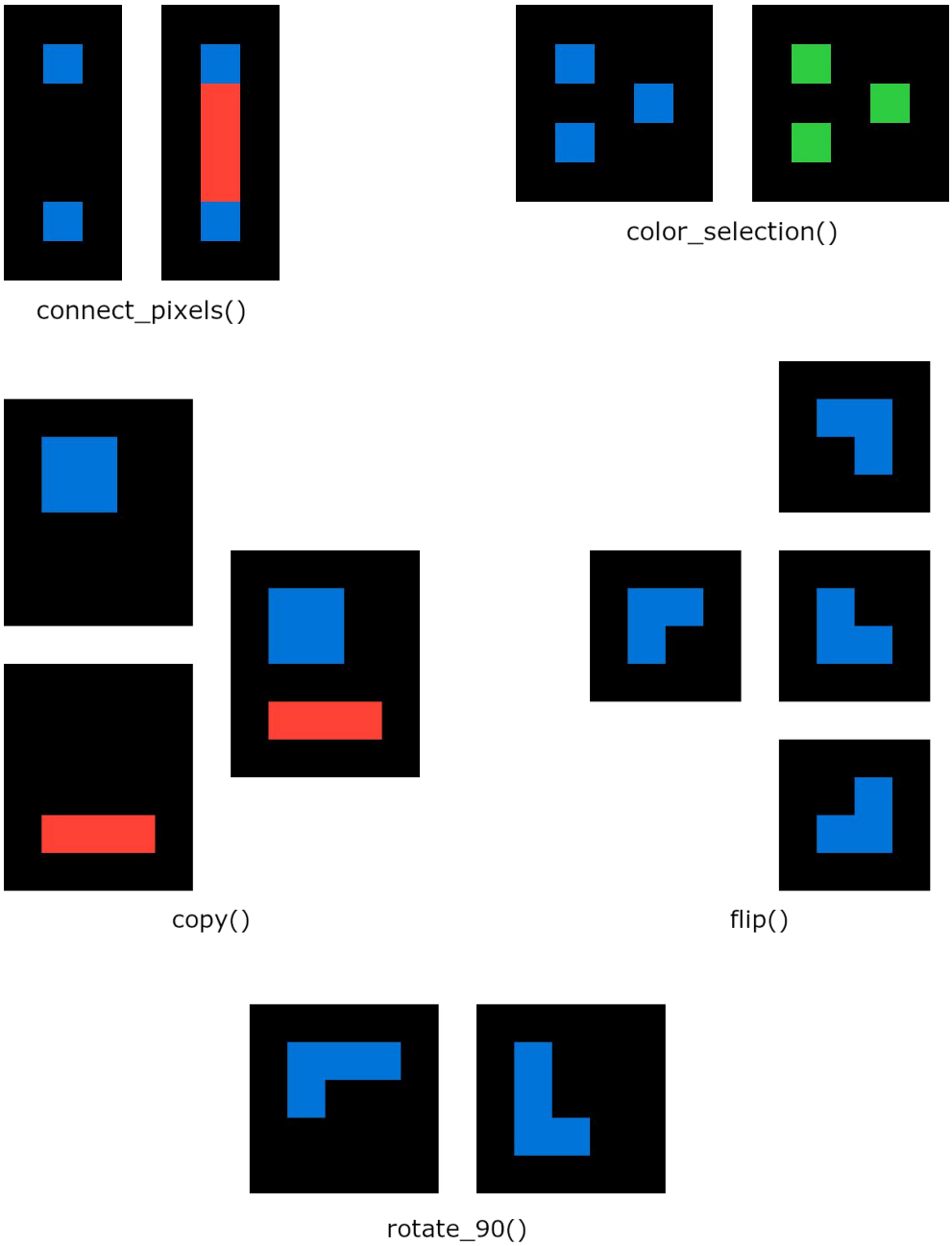


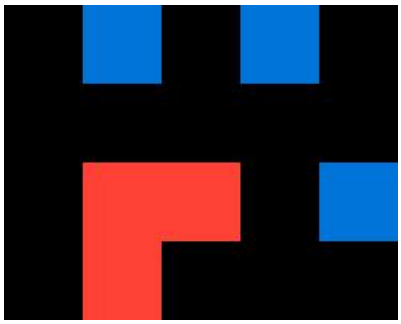
Figure 4.2: Examples of different tranformations

4.3 Selectors

Since each of the defined operators needs a selection of pixels to work with, we next designed a series of functions to extract sections of an image and return the corresponding pixel positions.

- `get_colored_pixels(im, colors)`: this function takes an image and a list of integers (encoding different colors) as input, and returns the positions of all the pixels in the image that are set to the specified color(s).
- `get_group(im, index, color)`: this function takes an image, an integer encoding a color, and an index as input, then performs a subroutine extracting all the groups of connected pixels that are set to the specified color, and finally returns the positions of the pixels in the group at the specified index.
- `get_aligned_pixels(im, px = None, colors = None, filter_same_color = False)`: this function takes an image and returns a list of pixel pairs, containing the positions of all pairs of pixels in the image that share a coordinate and are not contiguous.
  If the optional parameter `px` is specified, the function will instead return all pixels that share a coordinate with the specified pixel, paired with the latter.

  The optional parameter `colors` can be used to specify which colors to take into consideration. Any pixel set to a color not in the list will be ignored.
  The optional parameter `filter_same_color` can be set to True to only consider pair of pixels set to the same color.



`get_colored_pixels(image, [1]) = [(0, 1), (0, 3), (2, 4)]`
(selects only the blue pixels)

`get_group(image, 0, 2)` = [(2, 1), (2, 2), (3, 1)]
(selects the pixels in the first - and only - red group)

`get_aligned_pixels(im) = [((0, 1), (0, 3)), ((0, 1), (3, 1)), ((2, 2), (2, 4))]`
(selects all pairs of aligned pixels, indiscriminately)
`get_aligned_pixels(im, (2, 1)) = [((2, 1), (0, 1)), ((2, 1), (2, 4))]`
(selects all pixels aligned to - and paired with - the pixel in position `(2, 1)`, but not touching it)

The top-left pixel corresponds to the position `(y=0, x=0)`.
The color blue is encoded by the number 1, while red is encoded by the number 2.

Figure 4.3: Examples of selectors and their output

4.4 Training loop

For each task, we launched two separate consecutive iterations of the evolutionary algorithm, each with its own set of hyper-parameters and fitness evaluator: the first aimed only to find the correct dimensionality of the generated image, while the second one was in charge of generating the image itself.

4.4.1 Size estimation algorithm

The size algorithm utilizes a Manhattan distance-based fitness function, trying to minimize the distance between the generated size and the correct one.
For this process, we created two Macros:

- one initializes the image as a black image of fixed size (with the size being governed by parameters).
- the other generates a copy of the input image, applying a scaling factor and a size offset (both governed by parameters).

Then we have an Alternative Frame, selecting one of the two Macros to initialize the generated image.
The algorithm was set to run for 1000 iterations, but stop if a distance of 0.0 was reached.

4.4.2 Image generation algorithm

The image generation algorithm then has the responsibility of modifying the generated image to resemble as closely as possible the expected output, with the comparison being dictated by one of the fitness functions defined in Section 4.1. This process is the most complex, using a variety of different Macros and Frames.

The first Macros used are the selectors. We defined two Macros for extracting pixels from the input image and store them into variables:

- the first uses the `get_colored_pixels` to extract a number of pixels based on their color.
- the second uses the `get_group` function to instead try to extract information on contiguous sections of the image.

For both Macros, the arguments of the functions they call are governed by parameters. A sequence of Bunch Frames containing these Macros is then used to initialize these selectors.

A similar set of Macros and Frames is also used to extract pixels from the generated image, in the case relevant information is created during the generation process.
Since the number of variables holding selections of pixels is limited by the size of the

Bunch frames (a hyperparameter), a third set of Macros and Frames was also created to reassign variables to a new selection.

Next we implemented the Macros holding the transformation functions (where again the arguments of the functions were governed by parameters):

- A Macro connecting all pairs of aligned pixels in the image.
- A Macro connecting two selected pixels (assuming that the two selections contain only a single pixel each).
- A Macro to copy a section of either the input or generated image to the generated image itself.
- A Macro to flip a section of the generated image along one or both axes.
- A Macro to rotate a section of the generated image.
- A Macro to set a selection of pixels in the generated image to a different color.

These Macros are children of a Bunch frame, selecting them at random and improving the selection over time.

The image generation algorithm was tested with the two fitness functions defined in Section 4.1, and the results of both are reported in the next chapter.
The algorithm was set to run for 150 iterations. For the pixel-correctness fitness function, another stopping criterion was to reach a fitness value of 1.0.

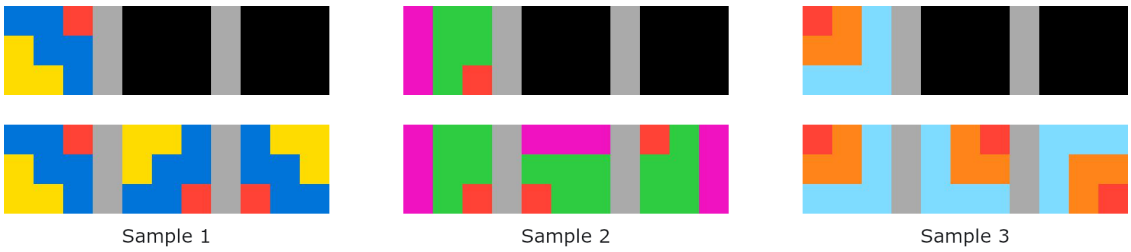
Chapter 5

Experimental Results

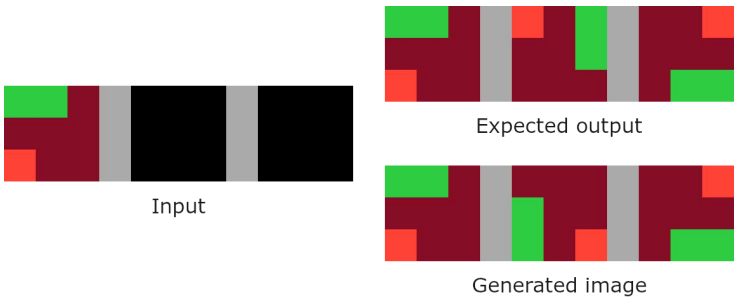
For each task, after training the model on the input samples, we executed the synthesized code on the evaluation sample(s) to generate the corresponding output images.

5.1 Visual results

The first assessment we can make is a visual comparison between the generated images and their corresponding expected outputs. Fig. 5.2 and show some examples of tasks and their generated output.

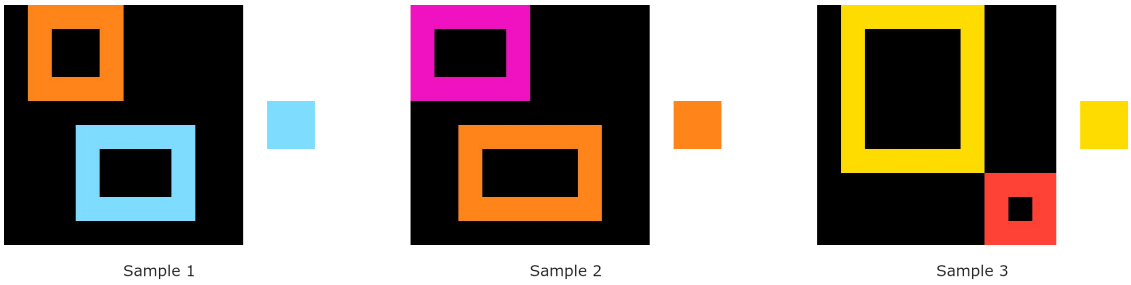


(a) Training samples

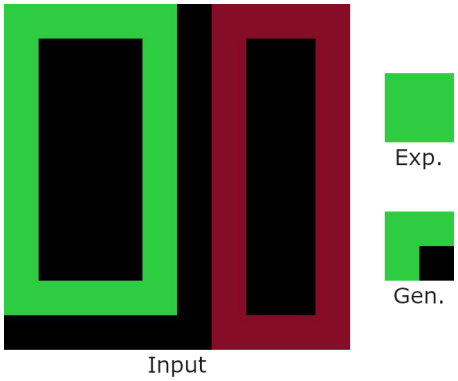


(b) Test sample and result

Figure 5.1: Task 8e5a5113



(a) Training samples



(b) Test sample and result

Figure 5.2: Task 445eab21

The model clearly has mixed performances.
Task 8e5a5113, for example, required a section of the input image to be rotated and translated along the output image. The model identified the correct rule but got a rotation wrong.
Task 445eab21 instead consisted of finding the larger rectangle in the input and filling a 2x2 square with the corresponding color for the output. In this case, the model copied a corner of the rectangle, trying to fill as much of the image as possible, but unable to add pixels on its own.

5.2 Quantitative results

Naturally, some kind of quantitative measure of the results is needed. To this end, we computed the pixel-correctness of each generated image (as a measure of accuracy) and compiled the results in Fig. 5.3.

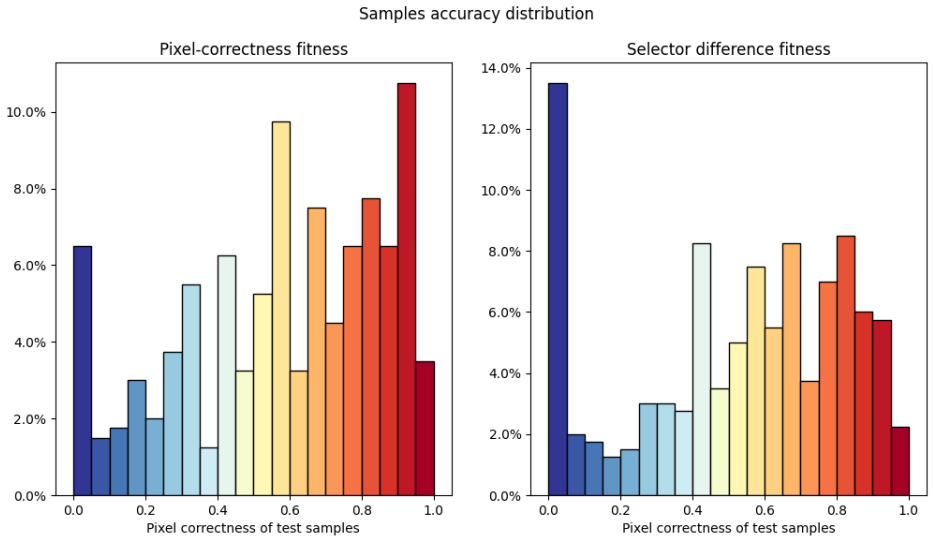


Figure 5.3: Tasks accuracy distribution

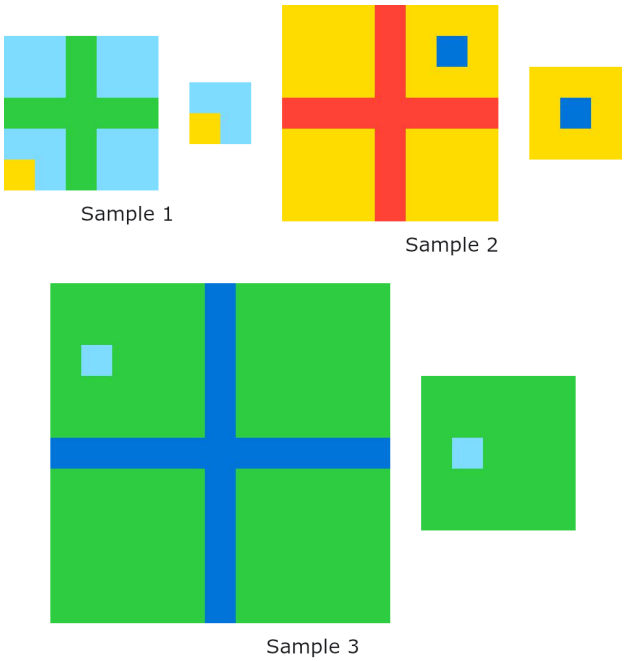
As shown in the graph, the fitness function used had a noticeable impact on the performance of the model.
While both functions resulted in about ∼8% of the samples reaching near-zero accuracy, the selectors difference fitness yielded the worst overall results, with more than 13% of the samples remaining almost or completely unsolved.
In contrast, the use of pixel-correctness fitness resulted in the best overall values, with most samples reaching an accuracy of 50% and above. Of particular note is also the fact that about ∼15% of the samples reached a high-level accuracy (95%+), and about ∼25% of these completely solved the task or came really close (99% - 100% accuracy).

5.3 Best case scenarios

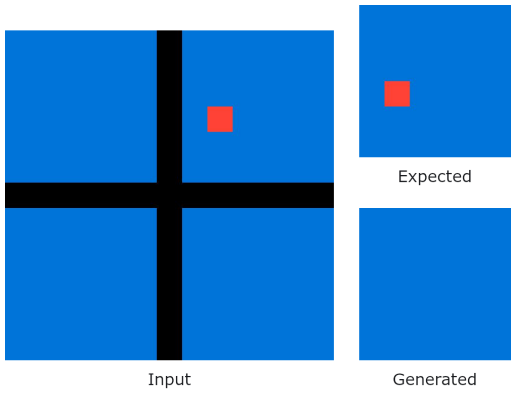
Another analysis we decided to perform was to check the best result with each fitness function, in order to gauge the potential of the algorithm.

Fig. 5.4 shows the best result obtained with the selector difference fitness. In this case, the task consisted of extracting the only colored square containing the blue dot.
The model did not manage to solve this task, despite reaching an accuracy of 97%. This clearly indicates a fault in the chosen metric, since pixel-correctness is not capable of meaningfully capturing a single-pixel difference, which in this case is critical to solve the

task.



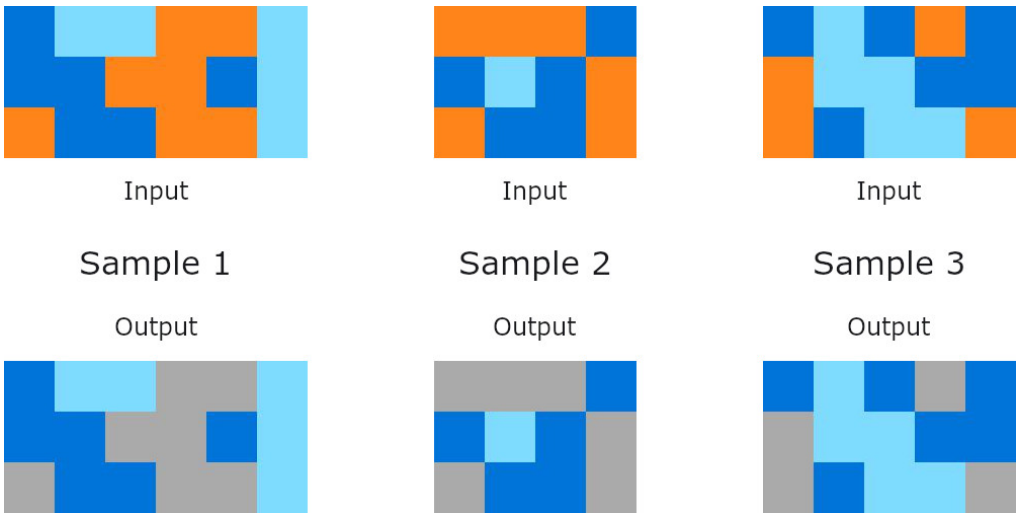
(a) Training samples



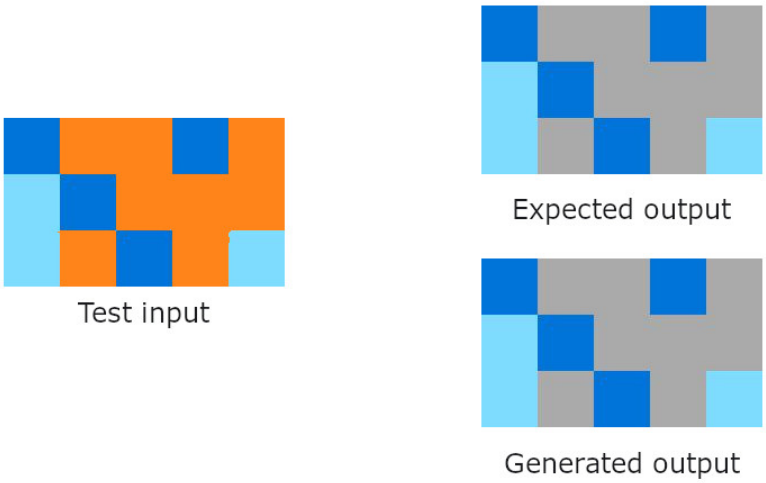
(b) Test sample and result

Figure 5.4: Best case results with selector difference fitness

Figs. 5.5, 5.6, and 5.7 instead show the best results obtained with the pixel-correctness fitness. Notice how all three are completed tasks, with no error.



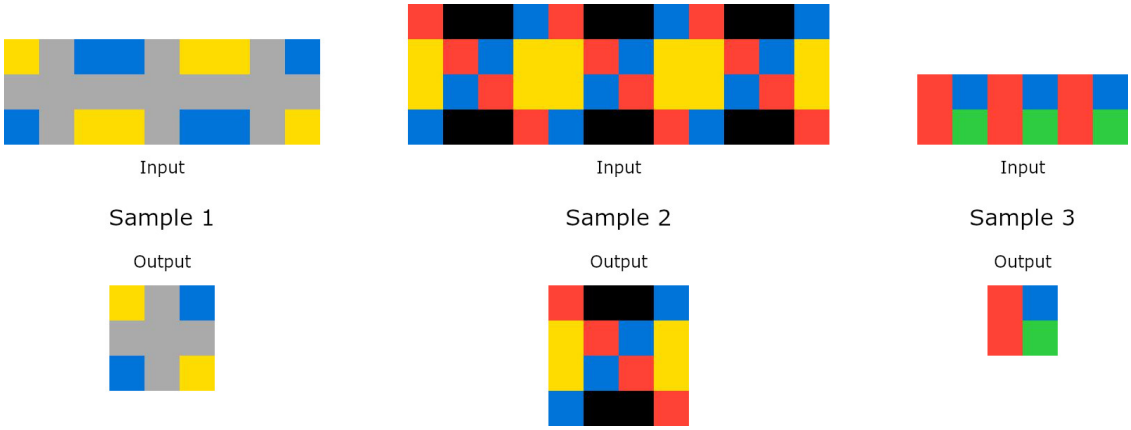
(a) Training samples



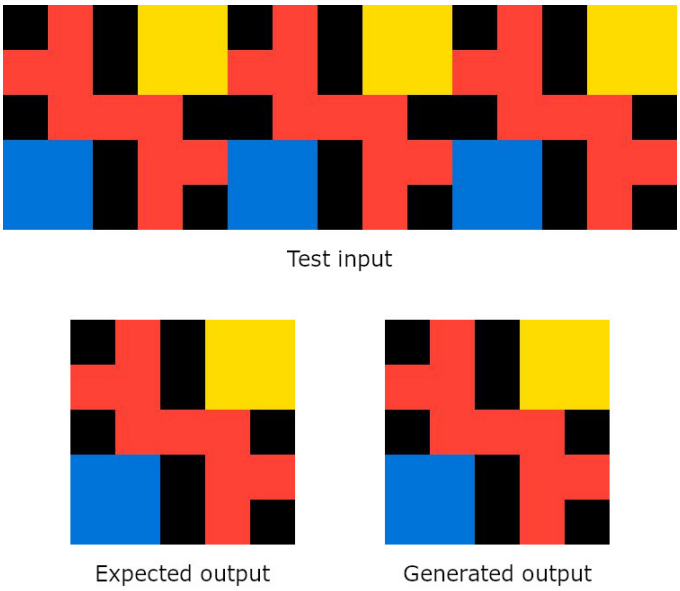
(b) Test sample and result

Figure 5.5: Task c8f0f002

The goal of this task was to re-color all orange sections in gray. The model managed to completely solve the task, probably thanks to the `color_selection()` function that was developed precisely for similar situations.



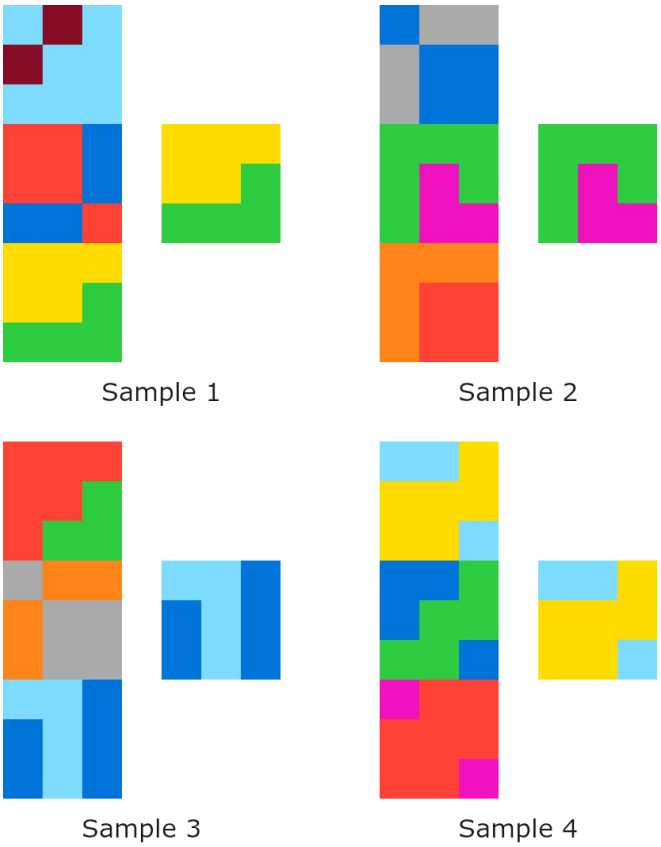
(a) Training samples



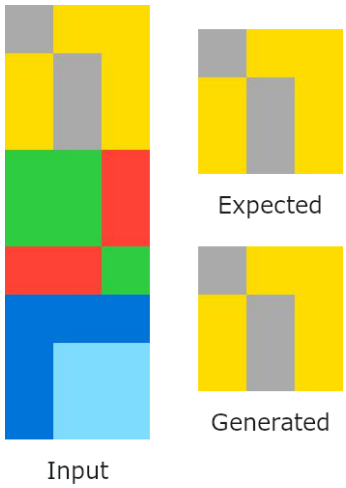
(b) Test sample and result

Figure 5.6: Task

In this task, the input consists of 3 copies of the same section. The goal is to isolate this section and return it as the output. Again, the model was capable of completely solving this task.



(a) Training samples



(b) Test sample and result

Figure 5.7: Task

Finally, this task also consisted in extracting a particular section of the input image, although the rule behind the choice of the section is not entirely clear.
In this case, the model managed to correctly solve the task, despite it being relatively difficult to understand.

These results are consistent with the accuracy distributions shown in Fig. 5.3, and show again that the pixel-correctness fitness function is more reliable than the selectors difference fitness.

Chapter 6

Conclusions

This thesis investigated the application of Genetic Programming to the Abstraction and Reasoning Corpus (ARC), a benchmark designed to evaluate progress toward Artificial General Intelligence. The study examined the ability of evolutionary computation to generate compositional, interpretable solutions for ARC tasks, and evaluated the performance of the Byron EA framework with two different fitness measures: Model A, which uses a pixel-correctness fitness with a form of data-balancing, and Model B, which instead uses a vector-based fitness to capture the information on particular sections of the images.

The comparative analysis revealed that both models left at least ∼8% of the evaluated tasks completely unsolved, highlighting the intrinsic difficulty of ARC and the limitations of current evolutionary approaches when confronted with sparse supervision and diverse task requirements.
Between the two models, notable differences were observed. Model A exhibited more consistent performance across tasks, suggesting a robustness that makes it less sensitive to variation in task structure. It also showed a high potential, managing to completely solve 3 different tasks. Model B, on the contrary, was overall less stable, with a higher number of completely unsolved tasks and, overall, less satisfactory results. It also did not manage to completely solve a single task.

These findings suggest that there is substantial room for improvement in both representation and search strategy. In particular, the use of alternative parameter settings or richer sets of operators could potentially lead to a significant performance improvement. Moreover, the reliance on pixel-correctness as the only evaluation metric may obscure other dimensions of progress; adopting alternative or complementary performance measures could provide a more nuanced assessment of the models strengths and weaknesses.

In summary, while neither of the proposed models achieves a comprehensive solution to ARC, the results demonstrate the promise and challenges of Genetic Programming in this domain. The contrast between consistency and ceiling performance provides a valuable perspective on the design space for evolutionary approaches, and the identified limitations point toward concrete avenues for future research.

Bibliography

[1] F. Chollet, "On the measure of intelligence," 2019.

[2] B. M. Lake, T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman, "Building machines that learn and think like people," *Behav. Brain Sci.*, vol. 40, no. e253, 2017.

[3] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Found. trends® program. lang.*, vol. 4, no. 1-2, pp. 1–119, 2017.

[4] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021.

[5] K. Ellis, C. Wong, M. Nye, M. Sable-Meyer, L. Cary, L. Morales, L. Hewitt, A. Solar-Lezama, and J. B. Tenenbaum, "DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning," 2020.

[6] J. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press, 1975.

[7] J. R. Koza, *Genetic programming*. Complex Adaptive Systems, Cambridge, MA: Bradford Books, Dec. 1992.

[8] G. Squillero, A. Tonda, D. Masetta, and M. Sacchet, "Byron: A fuzzer for turing-complete test programs," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, (New York, NY, USA), pp. 1691–1694, ACM, July 2024.

[9] F. Chollet, M. Knoop, G. Kamradt, and B. Landers, "ARC prize 2024: Technical report," 2024.

[10] "ARC Prize - Leaderboard — arcprize.org." https://arcprize.org/leaderboard. [Accessed 19-09-2025].

[11] D. Basin, Y. Deville, P. Flener, A. Hamfelt, and J. Fischer Nilsson, "Synthesis of programs in computational logic," in *Program Development in Computational Logic*, Lecture notes in computer science, pp. 30–65, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[12] B. Jeffries and L. Landauer, *Hunting Security Bugs*. Redmond, WA: Microsoft Press, Aug. 2006.

Appendix A

Script Python used to train the model and compile the results

```python
import itertools
import json
import traceback
import os

import byron
from byron.classes import FitnessABC, ParameterABC, Individual, SElement


import numpy as np
import scipy as sp


from tqdm import tqdm

import matplotlib.pyplot as plt
from matplotlib.ticker import PercentFormatter


from PIL import Image



train_set, test_set = None, None


# General utility functions
def sign(n: int) -> int:
    return int(np.copysign(1, n))

def distance(px1, px2) -> int:
    # Manhattan distance
    return abs(px1[0] - px2[0]) + abs(px1[1] - px2[1])


```

```
# Image utility functions

# Converts a list of 2-element tuples (encoding pixel positions) to the
    corresponding numpy indices
def pixels_list_to_indices(pixels: list) -> tuple[
    np.ndarray,
    np.ndarray
] | None:
    if len(pixels) == 0:
        return None

    stack: np.ndarray = np.vstack(pixels)
    indices: tuple[np.ndarray, np.ndarray] = stack[:, 0], stack[:, 1]

    return indices

# Returns the pixel value of a given image at a given pixel position
def get_image_value(im, px) -> int | None:
    index = pixels_list_to_indices([px])

    if index is None:
        return None

    try:
        return im[index]
    except IndexError:
        return None


# ARC task parser
def parse_arc_test_file(filename: str) -> tuple[list, list]:
    with open(filename) as file:
        data: dict[
            str, list[
                    dict[
                        str, list[list[int]]
                    ]
                ]
        ] = json.load(file)

        # List of train samples, encoded as Numpy arrays
        train_data: list = []

        for train_item in data['train']:
            train_data.append((
                np.array(train_item['input']).reshape(
                    len(train_item['input']),
                    len(train_item['input'][0])
                ),
                np.array(train_item['output']).reshape(
                    len(train_item['output']),
                    len(train_item['output'][0])
                )
            ))

```

```
        # List of test sample, encoded as Numpy arrays
        test_data: list = []

        for test_item in data['test']:
            test_data.append((
                np.array(test_item['input']).reshape(
                    len(test_item['input']),
                    len(test_item['input'][0])
                ),
                np.array(test_item['output']).reshape(
                    len(test_item['output']),
                    len(test_item['output'][0])
                )
            ))

    return train_data, test_data


# Code converter from Byron to Python
def extrapolate_code(element_or_genotype: Individual | Type[SElement] |
    ParameterABC | str) -> str:
    final_code = ''
    if isinstance(element_or_genotype, str):
        for line in element_or_genotype.splitlines():
            code = line.split(';')[0]

            if code.strip().strip(':').strip('n').isnumeric():
                continue
            if code.isspace():
                continue

            final_code += code.strip(' ') + '\n'
    else:
        for line in byron.f.as_text(element_or_genotype).splitlines():
            code = line.split(';')[0]

            if code.strip().strip(':').strip('n').isnumeric():
                continue
            if code.isspace():
                continue

            final_code += code.strip(' ') + '\n'
    return final_code.replace('\n\n\n', '\n').replace(':\n\n', ':\n')


# Operators

# Grouping operator
def get_connected_pixels_groups(im, color: int = 0) -> list:
    groups: list = []

    if color in range(1, 10):
        labels, num_features = sp.ndimage.label(im == color)
        for i in range(num_features):
            grp = np.argwhere(labels == i + 1).tolist()
```

```
144
145             if len(grp) > 1:
146                 groups.append(grp)
147     else:
148         labels, num_features = sp.ndimage.label(im != 0)
149         for i in range(num_features):
150             grp = np.argwhere(labels == i + 1).tolist()
151
152             if len(grp) > 1:
153                 groups.append(grp)
154
155     return groups
156
157 # Selectors
158 def get_colored_pixels(im, colors: list[int] = None) -> list:
159     if colors is None:
160         pixels: list = np.argwhere(im != 0).tolist()
161
162         if len(pixels) == 0:
163             return []
164
165         return pixels
166
167     if len(colors) <= 0:
168         return []
169
170     pixels: list = np.argwhere(
171         np.logical_or.reduce([np.array(im == color) for color in colors])
172     ).tolist()
173     return pixels
174
175 def get_group(im, index: int, color: int) -> list:
176     groups: list = get_connected_pixels_groups(im, color)
177     if index < len(groups):
178         return groups[index]
179
180     return []
181
182 # Alignment operator
183 def get_aligned_pixels(
184         im,
185         px = None,
186         colors: list[int] = None,
187         filter_same_color: bool = False
188 ) -> list[tuple]:
189     input_pixels: list = get_colored_pixels(im, colors)
190     if input_pixels is None:
191         input_pixels = get_colored_pixels(im)
192
193     if px is not None:
194         returned_pixels: list = [
195             (px, p) for p in input_pixels
196             if p[0] == px[0] or p[1] == px[1]
197         ]
198
```

```
        if px in returned_pixels:
            returned_pixels.remove(px)

        return returned_pixels

    aligned_pixels: list[tuple] = []

    groups: list = get_connected_pixels_groups(im)

    def check_same_value(px1, px2) -> bool:
        if not filter_same_color:
            return True

        value1 = get_image_value(im, px1)
        value2 = get_image_value(im, px2)

        if value1 is None or value2 is None:
            return False

        return value1 == value2

    same_x_pixels_combs = [
        c for c in itertools.combinations(input_pixels, 2)
        if c[0][1] == c[1][1] and not np.any([
            c[0] in grp and c[1] in grp for grp in groups
        ]) and check_same_value(c[0], c[1])
    ]
    same_y_pixels_combs = [
        c for c in itertools.combinations(input_pixels, 2)
        if c[0][0] == c[1][0] and not np.any([
            c[0] in grp and c[1] in grp for grp in groups
        ]) and check_same_value(c[0], c[1])
    ]

    aligned_pixels.extend(same_x_pixels_combs + same_y_pixels_combs)

    return aligned_pixels

# Connection operator
def connect_pixels(
        out,
        px1,
        px2,
        color: int | np.ndarray[tuple[int, ...], np.dtype[int]],
        left_connecting_pixel = None
) -> bool:
    if px1 is None or px2 is None:
        return False

    if px1[0] == px2[0]:
        out[px1[0], range(px1[1] + 1, px2[1])] = color
        return True

    if px1[1] == px2[1]:
        out[range(px1[0] + 1, px2[0]), px1[1]] = color
```

```
        return True

    connecting_point = [0, 0]
    if np.random.rand() < 0.5:
        connecting_point[0] = px1[0]
        connecting_point[1] = px2[1]
    else:
        connecting_point[0] = px2[0]
        connecting_point[1] = px1[1]

    if left_connecting_pixel == px1 and px2[1] < px1[1]:
        connecting_point[0] = px1[0]
        connecting_point[1] = px2[1]
    if left_connecting_pixel == px2 and px1[1] < px2[1]:
        connecting_point[0] = px2[0]
        connecting_point[1] = px1[1]

    out[
        range(
            min(px1[0], px2[0], connecting_point[0]) + 1,
            max(px1[0], px2[0], connecting_point[0])
        ),
        connecting_point[1]
    ] = color
    out[
        connecting_point[0],
        range(
            min(px1[1], px2[1], connecting_point[1]) + 1,
            max(px1[1], px2[1], connecting_point[1])
        )
    ] = color
    out[pixels_list_to_indices([connecting_point])] = color

    return True

# Copy operator
def copy(
        im,
        out,
        pixels: list = None,
        offset: tuple[int, int] = (0, 0),
        cut: bool = False
) -> bool:
    if pixels is None:
        pixels = get_colored_pixels(im)

    if pixels is None:
        return False

    error: bool = False
    for px in pixels:
        input_index: tuple[
            np.ndarray,
            np.ndarray
        ] = pixels_list_to_indices([px])
```

```
        if input_index is None:
            error = True
            continue

        image_value: int = get_image_value(im, px)
        if image_value is None:
            error = True
            continue

        try:
            if cut:
                im[input_index] = 0

            output_index: tuple[
                np.ndarray,
                np.ndarray
            ] = pixels_list_to_indices(
                [(px + np.array(offset)).tolist()]
            )
            if output_index is None:
                error = True
                continue
            if output_index[0] < 0 or output_index[1] < 0:
                error = True
                continue

            out[output_index] = image_value
        except (IndexError, ValueError):
            error = True

    return not error

# Symmetry operator
def flip(
        out,
        pixels: list = None,
        pivot = (-1, -1),
        flip_x: bool = False,
        flip_y: bool = False
) -> bool:
    if not (flip_x or flip_y):
        return False

    if pixels is None:
        pixels = np.argwhere(out != 0).tolist()

    if len(pixels) == 0:
        return False

    original_pivot = pivot
    if flip_x and pivot[1] == -1:
        pivot = [pivot[0], out.shape[1] // 2]
    if flip_y and pivot[0] == -1:
        pivot = [out.shape[0] // 2, pivot[1]]

```

```
    already_updated_indices: list[tuple[np.ndarray, np.ndarray]] = []

    error: bool = False
    for px in pixels:
        input_index: tuple[
            np.ndarray,
            np.ndarray
        ] = pixels_list_to_indices([px])
        if input_index is None:
            error = True
            continue

        image_value = get_image_value(out, px)
        if image_value is None:
            error = True
            continue

        flipped_px = px

        if flip_x:
            flipped_px[1] = px[1] + 2 * (pivot[1] - px[1])
            if original_pivot[1] == -1 and out.shape[1] % 2 == 0:
                flipped_px[1] -= 1
        if flip_y:
            flipped_px[0] = px[0] + 2 * (pivot[0] - px[0])
            if original_pivot[0] == -1 and out.shape[0] % 2 == 0:
                flipped_px[0] -= 1

        try:
            flipped_index = pixels_list_to_indices([flipped_px])
            if flipped_index is None:
                error = True
                continue
            if flipped_index[0] < 0 or flipped_index[1] < 0:
                error = True
                continue

            if (
                input_index != flipped_index and
                not input_index in already_updated_indices
            ):
                out[input_index] = 0

            out[flipped_index] = image_value
            already_updated_indices.append(flipped_index)
        except (IndexError, ValueError):
            error = True

    return not error

# Rotation operator
def rotate_90(
        out,
        pixels: list = None,
        times: int = 1,
```

```
        pivot = (-1, -1)
) -> bool:
    if pixels is None:
        pixels = np.argwhere(out != 0).tolist()

    input_indices: tuple[
        np.ndarray,
        np.ndarray
    ] = pixels_list_to_indices(pixels)

    if input_indices is None:
        return False

    boundary_top_left: tuple[int, int] = (
        np.min(input_indices[0]),
        np.min(input_indices[1])
    )
    boundary_size: tuple[int, int] = (
        np.max(input_indices[0]) - boundary_top_left[0] + 1,
        np.max(input_indices[1]) - boundary_top_left[1] + 1
    )

    original_pivot = pivot
    if pivot[0] == -1 or pivot[1] == -1:
        if boundary_size[0] != boundary_size[1]:
            return False

        pivot = [
            boundary_top_left[0] + boundary_size[0] // 2,
            boundary_top_left[1] + boundary_size[1] // 2
        ]

    cosine: int = 0
    sine: int = 0
    no_pivot_offset = [0, 0]
    match times % 4:
        case 0:
            cosine, sine = 1, 0
        case 1:
            cosine, sine = 0, 1
            no_pivot_offset = [0, -1]
        case 2:
            cosine, sine = -1, 0
        case 3:
            cosine, sine = 0, -1
            no_pivot_offset = [-1, 0]

    already_updated_indices: list[tuple[np.ndarray, np.ndarray]] = []

    error: bool = False
    for px in pixels:
        input_index: tuple[
            np.ndarray,
            np.ndarray
        ] = pixels_list_to_indices([px])
```

```
        if input_index is None:
            error = True
            continue

        image_value = get_image_value(out, px)
        if image_value is None:
            error = True
            continue

        offset_from_pivot = (np.array(px) - np.array(pivot)).tolist()
        rotation_offset = [
            -offset_from_pivot[1] * sine - offset_from_pivot[0] * cosine,
            offset_from_pivot[0] * sine - offset_from_pivot[1] * cosine
        ]

        rotated_pixel = (pivot + np.array(rotation_offset)).tolist()
        if (
                original_pivot == (-1, -1) and
                pivot == [
                    boundary_top_left[0] + boundary_size[0] // 2,
                    boundary_top_left[1] + boundary_size[1] // 2
                ] and
                boundary_size[0] % 2 == 0 and boundary_size[1] % 2 == 0
        ):
            rotated_pixel = (
                rotated_pixel + np.array(no_pivot_offset)
            ).tolist()

        try:
            rotated_index: tuple[
                np.ndarray,
                np.ndarray
            ] = pixels_list_to_indices([rotated_pixel])

            if (
                input_index != rotated_index and
                not input_index in already_updated_indices
            ):
                out[input_index] = 0

            if rotated_index is None:
                error = True
                continue
            if rotated_index[0] < 0 or rotated_index[1] < 0:
                error = True
                continue

            out[rotated_index] = image_value
            already_updated_indices.append(rotated_index)
        except (IndexError, ValueError):
            error = True

    return not error

# Pixel-coloring operator
```

```
def color_selection(
    im,
    pixels: list,
    color: int,
    offset: tuple[int, int] = (0, 0)
) -> bool:
    if len(pixels) == 0:
        return False

    error: bool = False
    for px in pixels:
        try:
            output_index: tuple[
                np.ndarray,
                np.ndarray
            ] = pixels_list_to_indices(
                [(px + np.array(offset)).tolist()]
            )
            im[output_index] = color
        except (IndexError, ValueError):
            error = True

    return not error


# Fitness functions

# Image size fitness (for size estimation)
@byron.fitness_function()
def size_fitness(genotype: str) -> FitnessABC:
    fitness: list[int] = []

    for train_sample in train_set:
        input_image, output_image = train_sample

        exec(extrapolate_code(genotype), {
            'np': np,
            'copy': copy
        }, locals())

        fitness.append(-distance(
            locals()['generated_image'].shape,
            output_image.shape
        ))

    return byron.fit.Scalar(sum(fitness))

# Pixel-correctness fitness
@byron.fitness_function
def pixel_correctness_fitness(genotype: str) -> FitnessABC:
    fitness: list[float] = []

    for train_sample in train_set:
        input_image, output_image = train_sample

```

```
        try:
            exec(extrapolate_code(genotype), {
                'np':np,
                'sp':sp,
                'sign':sign,
                'pixels_list_to_indices':pixels_list_to_indices,
                'get_image_value':get_image_value,
                'get_colored_pixels':get_colored_pixels,
                'get_group':get_group,
                'get_aligned_pixels':get_aligned_pixels,
                'connect_pixels':connect_pixels,
                'get_connected_pixels_groups':get_connected_pixels_groups,
                'copy':copy,
                'flip':flip,
                'rotate_90':rotate_90,
                'color_selection':color_selection
            }, locals())
        except:
            return byron.fit.Scalar(-1.0)

        comparison_image = locals()['generated_image']

        if comparison_image.shape != output_image.shape:
            comparison_image = np.zeros(
                shape=output_image.shape,
                dtype=int
            )
            copy(locals()['generated_image'], comparison_image)

        pixel_correctness: float = 0.0

        uniq = np.unique_counts(output_image)
        index = np.argmax(uniq.counts)

        if (
            uniq.counts[index] < output_image.size and
            uniq.counts[index] / output_image.size >= 0.5
        ):
            output_image[output_image == uniq.values[index]] = -1
            comparison_image[comparison_image == uniq.values[index]] = -1

            output_image[np.nonzero(output_image - comparison_image)] = -2

            pixel_correctness = np.count_nonzero(
                output_image >= 0
            ) / np.count_nonzero(
                output_image + 1
            )
        else:
            pixel_correctness = 1 - np.count_nonzero(
                output_image - comparison_image
            ) / output_image.size

        fitness.append(pixel_correctness)

```

```
    return byron.fit.Scalar(np.mean(fitness))

# Selectors difference fitness
SELECTORS_DIFFERENCE_FITNESS_TYPE = byron.fit.reverse_fitness(
    byron.fit.Lexicographic
)
@byron.fitness_function(type_ = SELECTORS_DIFFERENCE_FITNESS_TYPE)
def selectors_difference_fitness(genotype: str) -> FitnessABC:
    fitness: list[list[float]] = []

    for train_sample in train_set:
        input_image, output_image = train_sample

        try:
            exec(extrapolate_code(genotype), {
                'np':np,
                'sp':sp,
                'sign':sign,
                'pixels_list_to_indices':pixels_list_to_indices,
                'get_image_value':get_image_value,
                'get_colored_pixels':get_colored_pixels,
                'get_group':get_group,
                'get_aligned_pixels':get_aligned_pixels,
                'connect_pixels':connect_pixels,
                'get_connected_pixels_groups':get_connected_pixels_groups,
                'copy':copy,
                'flip':flip,
                'rotate_90':rotate_90,
                'color_selection':color_selection
            }, locals())
        except:
            print('-------------------- Error --------------------')
            print(extrapolate_code(genotype))
            print(traceback.format_exc())
            print('-----------------------------------------------')

        comparison_image = locals()['generated_image']

        if comparison_image.shape != output_image.shape:
            comparison_image = np.zeros(
                shape = output_image.shape,
                dtype=int
            )
            copy(locals()['generated_image'], comparison_image)


        output_color_selections = []
        for combination in itertools.combinations_with_replacement(
            range(1, 10),
            MAX_COLORS_NUMBER
        ):
            output_color_selections.append(
                get_colored_pixels(output_image, combination)
            )

```

```
        output_group_selections = [
            get_group(output_image, i, color)
            for i in range(MAX_GROUP_INDEX) for color in range(0, 10)
        ]

        comparison_color_selections = []
        for combination in itertools.combinations_with_replacement(
            range(1, 10),
            MAX_COLORS_NUMBER
        ):
            comparison_color_selections.append(
                get_colored_pixels(comparison_image, combination)
            )

        comparison_group_selections = [
            get_group(comparison_image, i, color)
            for i in range(MAX_GROUP_INDEX) for color in range(0, 10)
        ]


        color_groups_diff = []
        for i in range(len(output_color_selections)):
            len_diff = abs(
                len(output_color_selections[i]) -
                len(comparison_color_selections[i])
            )
            if len_diff > 0:
                color_groups_diff.append(100 * len_diff)
                continue

            if len(output_color_selections[i]) == 0:
                color_groups_diff.append(0.0)
                continue

            pixel_distances = []
            for j in range(len(output_color_selections[i])):
                pixel_distances.append(distance(
                    output_color_selections[i][j],
                    comparison_color_selections[i][j]
                ))

            color_groups_diff.append(np.mean(pixel_distances))

        groups_diff = []
        for i in range(len(output_group_selections)):
            len_diff = abs(
                len(output_group_selections[i]) -
                len(comparison_group_selections[i])
            )
            if len_diff > 0:
                groups_diff.append(100 * len_diff)
                continue

            if len(output_group_selections[i]) == 0:
                groups_diff.append(0.0)
```

```
                continue

            pixel_distances = []
            for j in range(len(output_group_selections[i])):
                pixel_distances.append(distance(
                    output_group_selections[i][j],
                    comparison_group_selections[i][j]
                ))

            groups_diff.append(np.mean(pixel_distances))

        res = color_groups_diff + groups_diff
        np.random.shuffle(res)

        fitness.append(res)

    return byron.fit.Lexicographic(np.mean(fitness, axis = 0).tolist())


# Size estimation hyperparameters

MAX_SIZE_GENERATIONS = 1000

MAX_FIXED_SIZE = 15
MAX_DELTA_SIZE = 5


# Image generation hyperparameters

MAX_IMAGE_GENERATIONS = 150

MAX_COLORS_NUMBER = 5
MAX_GROUP_INDEX = 15

MAX_OFFSET = 10
MAX_PIVOT = 20

MAX_OPERATIONS = 75



# Single-task train + test function
def solve_task(fitness = selectors_difference_fitness) -> tuple[float, str
    ]:

    # ------------------------ SIZE ESTIMATION ------------------------

    # Image initializing macros
    init_image_fixed_size = byron.f.macro(
        'size_x = {x}\n'
        'size_y = {y}\n'
        'generated_image = np.zeros((size_y, size_x), dtype=int)',
        x=byron.f.integer_parameter(1, MAX_FIXED_SIZE),
        y=byron.f.integer_parameter(1, MAX_FIXED_SIZE)
    )
```

```
    init_image_input_based_size = byron.f.macro(
        'scale_x = {sx}\n'
        'scale_y = {sy}\n'
        'delta_x = {dx}\n'
        'delta_y = {dy}\n'
        '\n'
        'new_size_x = int(input_image.shape[1] * scale_x + delta_x)\n'
        'new_size_y = int(input_image.shape[0] * scale_y + delta_y)\n'
        '\n'
        'if new_size_x < 0:\n'
        '\tnew_size_x = 0\n'
        'if new_size_y < 0:\n'
        '\tnew_size_y = 0\n'
        '\n'
        'generated_image = np.zeros((new_size_y, new_size_x), int)\n'
        'copy(input_image, generated_image)',
        sx = byron.f.choice_parameter([0.25, 0.5, 1.0, 1.5, 2.0]),
        sy = byron.f.choice_parameter([0.25, 0.5, 1.0, 1.5, 2.0]),
        dx = byron.f.integer_parameter(-MAX_DELTA_SIZE, MAX_DELTA_SIZE+1),
        dy = byron.f.integer_parameter(-MAX_DELTA_SIZE, MAX_DELTA_SIZE+1)
    )

    # Initializer choice
    init_image = byron.f.bunch([
        init_image_fixed_size,
        init_image_input_based_size
    ])


    # Size estimation training + image initialization code extraction
    evaluator = byron.evaluator.PythonEvaluator(
        size_fitness,
        backend='joblib'
    )
    image_initialization_code = extrapolate_code(byron.ea.adaptive_ea(
        init_image, evaluator,
        max_generation=MAX_SIZE_GENERATIONS,
        max_fitness=byron.fit.Scalar(0.0)
    )[0])

    # ----------------------------------------------------------------


    # ----------------------- IMAGE GENERATION -----------------------

    # Input image selection macros
    input_select_colors = byron.f.macro(
        '{_node} = get_colored_pixels(input_image, [{colors}])',
        colors = byron.f.array_parameter(
            range(1, 10),
            MAX_COLORS_NUMBER,
            ', '
        ),
        _label = ''
    )
```

```
    input_select_group = byron.f.macro(
        '{_node} = get_group(input_image, {index}, {color})',
        index = byron.f.integer_parameter(0, MAX_GROUP_INDEX),
        color = byron.f.integer_parameter(0, 10),
        _label = ''
    )

    # Input image selections sequences
    input_selections = byron.f.sequence([
        byron.f.bunch([input_select_colors], (0, 10)),
        byron.f.bunch([input_select_group], (0, MAX_GROUP_INDEX + 1))
    ])


    # Generated image selection macros
    generated_select_colors = byron.f.macro(
        '{_node} = get_colored_pixels(generated_image, [{colors}])',
        colors = byron.f.array_parameter(
            range(1, 10),
            MAX_COLORS_NUMBER,
            ', '
        ),
        _label = ''
    )
    generated_select_group = byron.f.macro(
        '{_node} = get_group(generated_image, {index}, {color})',
        index = byron.f.integer_parameter(0, MAX_GROUP_INDEX),
        color = byron.f.integer_parameter(0, 10),
        _label = ''
    )

    # Generated image selections sequences
    generated_selections = byron.f.sequence([
        byron.f.bunch([generated_select_colors], (0, 10)),
        byron.f.bunch([generated_select_group], (0, MAX_GROUP_INDEX + 1)),
    ])


    # Input image reassignment macros
    reassign_input_color_selection = byron.f.macro(
        '\n{ref} = get_colored_pixels(input_image, [{colors}])\n\n',
        ref = byron.f.global_reference(input_selections),
        colors = byron.f.array_parameter(
            range(1, 10),
            MAX_COLORS_NUMBER,
            ', '
        )
    )
    reassign_input_group_selection = byron.f.macro(
        '\n{ref} = get_group(input_image, {index}, {color})\n\n',
        ref = byron.f.global_reference(input_selections),
        index = byron.f.integer_parameter(0, MAX_GROUP_INDEX),
        color = byron.f.integer_parameter(0, 10)
    )

```

```
    # Input image reassignments sequences
    reassign_input_selection = byron.f.bunch([
        reassign_input_color_selection,
        reassign_input_group_selection
    ])


    # Generated image reassignment macros
    reassign_generated_color_selection = byron.f.macro(
        '\n{ref} = get_colored_pixels(generated_image, [{colors}])\n\n',
        ref = byron.f.global_reference(input_selections),
        colors = byron.f.array_parameter(
            range(1, 10),
            MAX_COLORS_NUMBER,
            ', '
        )
    )
    reassign_generated_group_selection = byron.f.macro(
        '\n{ref} = get_group(generated_image, {index}, {color})\n\n',
        ref = byron.f.global_reference(input_selections),
        index = byron.f.integer_parameter(0, MAX_GROUP_INDEX),
        color = byron.f.integer_parameter(0, 10)
    )

    # Generated image reassignments sequences
    reassign_generated_selection = byron.f.bunch([
        reassign_generated_color_selection,
        reassign_generated_group_selection
    ])


    # Image manipulation macros and frames

    # Frame - Connect operator + alignment operator
    connect_aligned_pixels = byron.f.sequence([
        'px = None',
        byron.f.bunch([
            byron.f.macro(
                'if len({ref}) == 1:\n'
                '\tpx = {ref}[0]',
                ref = byron.f.global_reference(generated_selections)
            )
        ], (0, 2)),
        byron.f.bunch([
            byron.f.macro(
                'colors = None'
            ),
            byron.f.macro(
                'colors = [{colors}]',
                colors = byron.f.array_parameter(range(1, 10), 9, ', ')
            )
        ]),
        '\n',
        'for px1, px2 in get_aligned_pixels(generated_image, px, colors):'
    ,
```

```
        byron.f.bunch([
            byron.f.macro(
                '\tcolor = {color}',
                color = byron.f.integer_parameter(1, 10)
            ),
            byron.f.macro(
                '\tcolor = get_image_value(generated_image, {px})',
                px = byron.f.choice_parameter(['px1', 'px2'])
            )
        ]),
        '\n',
        '\tconnect_pixels(generated_image, px1, px2, color)'
    ])

    # Frame - Connect operator (with selections global reference
    parameters)
    connect_selections = byron.f.sequence([
        byron.f.macro(
            'if len({ref1}) == 1 and len({ref2}) == 1:\n'
            '\tpx1 = {ref1}[0]\n'
            '\tpx2 = {ref2}[0]',
            ref1 = byron.f.global_reference(generated_selections),
            ref2 = byron.f.global_reference(generated_selections)
        ),
        byron.f.bunch([
            byron.f.macro(
                '\tcolor = {color}',
                color = byron.f.integer_parameter(1, 10)
            ),
            byron.f.macro(
                '\tcolor = get_image_value(generated_image, {px})',
                px = byron.f.choice_parameter(['px1', 'px2'])
            )
        ]),
        byron.f.macro(
            '\tleft_px = {px}',
            px=byron.f.choice_parameter(['px1', 'px2', 'None'])
        ),
        '\n',
        '\tconnect_pixels(generated_image, px1, px2, color, left_px)'
    ])

    # Frame - Copy operator
    copy_to_generated = byron.f.sequence([
        byron.f.bunch([
            byron.f.sequence([
                'im = input_image',
                '\n',
                byron.f.bunch([
                    byron.f.macro(
                        'pixels = None',
                    ),
                    byron.f.macro(
                        'pixels = {ref}',
                        ref = byron.f.global_reference(input_selections)
```

```
                    )
                ])
            ]),
            byron.f.sequence([
                'im = generated_image',
                '\n',
                byron.f.bunch([
                    byron.f.macro(
                        'pixels = None',
                    ),
                    byron.f.macro(
                        'pixels = {ref}',
                        ref = byron.f.global_reference(
                            generated_selections
                        )
                    )
                ])
            ])
        ]),
        '\n',
        byron.f.bunch([
            byron.f.macro(
                'offset = (0, 0)'
            ),
            byron.f.macro(
                'offset = ({offset})',
                offset = byron.f.array_parameter(
                    range(1, MAX_OFFSET),
                    2,
                    ', '
                )
            )
        ]),
        '\n',
        byron.f.macro(
            'copy(im, generated_image, pixels, offset, {cut})',
            cut = byron.f.choice_parameter([True, False])
        )
    ])

    # Frame - Symmetry operator
    flip_generated = byron.f.sequence([
        byron.f.bunch([
            byron.f.macro('pixels = None'),
            byron.f.macro(
                'pixels = {ref}',
                ref = byron.f.global_reference(
                    generated_selections,
                    creative_zeal = 1
                )
            )
        ]),
        '\n',
        byron.f.bunch([
            byron.f.macro('pivot = (-1, -1)'),
```

```
            byron.f.macro(
                'pivot = ({pivot})',
                pivot = byron.f.array_parameter(
                    range(1, MAX_PIVOT),
                    2,
                    ', '
                )
            )
        ]),
        '\n',
        byron.f.bunch([
            byron.f.macro('flip_x, flip_y = True, False'),
            byron.f.macro('flip_x, flip_y = False, True'),
            byron.f.macro('flip_x, flip_y = True, True'),
        ]),
        '\n',
        'flip(generated_image, pixels, pivot, flip_x, flip_y)'
    ])

    # Frame - Rotation operator
    rotate_generated = byron.f.sequence([
        byron.f.bunch([
            byron.f.macro('pixels = None'),
            byron.f.macro(
                'pixels = {ref}',
                ref = byron.f.global_reference(
                    generated_selections,
                    creative_zeal = 1
                )
            )
        ]),
        '\n',
        byron.f.bunch([
            byron.f.macro('pivot = (-1, -1)'),
            byron.f.macro(
                'pivot = ({pivot})',
                pivot = byron.f.array_parameter(
                    range(1, MAX_PIVOT),
                    2,
                    ', '
                )
            )
        ]),
        '\n',
        byron.f.macro(
            'rotate_90(generated_image, pixels, {times}, pivot)',
            times = byron.f.integer_parameter(1, 5)
        )
    ])

    # Frame - Pixel-coloring operator
    color_generated = byron.f.sequence([
        byron.f.bunch([
            byron.f.macro('offset = (0, 0)'),
            byron.f.macro(
```

```
                'offset = ({offset})',
                offset = byron.f.array_parameter(
                    range(1, MAX_OFFSET),
                    2,
                    ', '
                )
            )
        ]),
        byron.f.macro(
            'color_selection(generated_image, {ref}, {color}, offset)\n\n',
            ref=byron.f.global_reference(
                generated_selections,
                creative_zeal = 1
            ),
            color=byron.f.integer_parameter(1, 10)
        )
    ])


    # Bunch frame containing all operator frames.
    operations = byron.f.bunch([
        reassign_input_selection,
        reassign_generated_selection,
        connect_aligned_pixels,
        connect_selections,
        copy_to_generated,
        flip_generated,
        rotate_generated,
        color_generated
    ], size=(1, MAX_OPERATIONS + 1))

    # Final image generation frame, contains all selections and operators
    generate_image = byron.f.sequence([
        input_selections,
        '\n',
        generated_selections,
        '\n',
        operations
    ])


    # Final training frame, contains the initialization code previously
    evolved and the image generation frame
    train = byron.f.sequence([
        image_initialization_code,
        '\n',
        generate_image
    ])


    # Image generation training + final code extraction
    evaluator = byron.evaluator.PythonEvaluator(
        fitness,
        backend = 'joblib'
```

```
    )
    final_code = extrapolate_code(byron.ea.adaptive_ea(
        train,
        evaluator,
        max_generation = MAX_IMAGE_GENERATIONS,
        max_fitness =
            byron.fit.Scalar(1.0) if fitness == pixel_correctness_fitness
            else None
    )[0])

    # ---------------------------------------------------------------


    # ----------------------------- TEST ------------------------------

    fitness: list[float] = []

    for index, test_sample in enumerate(test_set):
        input_image, output_image = test_sample

        # Execute the final code on the test samples
        exec(final_code, {
            'np': np,
            'sp': sp,
            'sign': sign,
            'pixels_list_to_indices': pixels_list_to_indices,
            'get_image_value': get_image_value,
            'get_colored_pixels': get_colored_pixels,
            'get_group': get_group,
            'get_aligned_pixels': get_aligned_pixels,
            'connect_pixels': connect_pixels,
            'get_connected_pixels_groups': get_connected_pixels_groups,
            'copy': copy,
            'flip': flip,
            'rotate_90': rotate_90,
            'color_selection': color_selection
        }, locals())

        # Isolate the generated image, and fit it to the output size
        comparison_image = locals()['generated_image']
        if comparison_image.shape != output_image.shape:
            comparison_image = np.zeros(
                shape=output_image.shape,
                dtype=int
            )
            copy(locals()['generated_image'], comparison_image)

        # Compute the pixel-correctness value for the sample
        pixel_correctness = 1 - np.count_nonzero(
            output_image - comparison_image
        ) / output_image.size

        fitness.append(pixel_correctness)

```

```
    # Return the mean pixel-correctness across all test samples, as well as the final code
    return np.mean(fitness), final_code


# Utility function for graph drawing
def draw_gradient_histogram(axis, data: list, title: str):
    cm = plt.cm.get_cmap('RdYlBu_r')

    _, bins, patches = axis.hist(
        data,
        range=np.arange(0.0, 1.1, 0.1),
        bins=np.arange(0.0, 1.05, 0.05),
        weights=np.ones(len(data)) / len(data),
        edgecolor='black'
    )
    axis.yaxis.set_major_formatter(PercentFormatter(1))

    bins_center = 0.5 * (bins[:-1] + bins[1:])

    col = bins_center - min(bins_center)
    col /= max(col)

    for c, p in zip(col, patches):
        plt.setp(p, 'facecolor', cm(c))

    axis.set_title(title)
    axis.set(xlabel='Pixel correctness of test samples')


# Numpy array to Image converter
def ARC_grid_to_image(grid: np.ndarray) -> Image:
    color_map = {
        0: (0, 0, 0),
        1: (0, 116, 217),
        2: (255, 65, 54),
        3: (46, 204, 64),
        4: (255, 220, 0),
        5: (170, 170, 170),
        6: (240, 18, 190),
        7: (255, 133, 27),
        8: (127, 219, 255),
        9: (135, 12, 37)
    }

    image = Image.new(mode='RGB', size=grid.shape[::-1])

    for y in range(grid.shape[0]):
        for x in range(grid.shape[1]):
            image.putpixel((x, y), color_map[grid[y, x]])

    image = image.resize(
        (image.size[0] * 50, image.size[1] * 50),
        Image.Resampling.NEAREST
    )
```

```
1292     return image
1293
1294
1295 # File execution entry point
1296 if __name__ == '__main__':
1297     byron.logger.setLevel(byron.logging.CRITICAL)
1298
1299     # Compute test results for all 400 tasks in 'data/training', using the
          pixel-correctness fitness function during training
1300     pixel_correctness_fitness_results = []
1301     for filename in tqdm(os.listdir('data/training')):
1302         train_set, test_set = parse_arc_test_file(
1303             f'data/training/{filename}'
1304         )
1305
1306         acc, _ = solve_task(fitness=pixel_correctness_fitness)
1307         pixel_correctness_fitness_results.append(acc)
1308
1309     # Compute test results for all 400 tasks in 'data/training', using the
          pixel-correctness fitness function during training
1310     selector_difference_fitness_results = []
1311     for filename in tqdm(os.listdir('data/training')):
1312         train_set, test_set = parse_arc_test_file(
1313             f'data/training/{filename}'
1314         )
1315
1316         acc, _ = solve_task(fitness=selectors_difference_fitness)
1317         selector_difference_fitness_results.append(acc)
1318
1319     # Plot results
1320     fig, (ax1, ax2) = plt.subplots(1, 2)
1321     fig.suptitle('Samples accuracy distribution')
1322     fig.set_size_inches(12.0, 6.0)
1323
1324     draw_gradient_histogram(
1325         ax1,
1326         pixel_correctness_fitness_results,
1327         'Pixel-correctness fitness'
1328     )
1329     draw_gradient_histogram(
1330         ax2,
1331         selector_difference_fitness_results,
1332         'Selector difference fitness'
1333     )
1334
1335     plt.show()
```