


Politecnico di Torino

Politecnico di Torino

Master's Degree Course in Computer Engineering

A.y. 2025/2026

Graduation Session September 2026

# Co-Evolving Representation and Program: A Solver for ARC-AGI

## Evolving parsers, programs, and projectors over a multigraph

Supervisors:
Giovanni Squillero
Alberto Tonda
Giorgio Mongardi

Candidate:
Luca Favole

# Abstract

Machines write complex code and solve olympiad problems, yet fail at puzzles most people solve without training. The Abstraction and Reasoning Corpus (ARC) makes that gap measurable. Each task gives a few coloured grids and asks for the output grid a further input should produce.

Because every task is new, no training distribution can be fitted and data contamination cannot inflate a score. Humans average above 70 % on the training set, while plain language models manage about 10 %; reasoning models reach above 90 %, but on subsets of the tasks and at dollars per puzzle. An alternative symbolic route would be to search for a short program fitting the examples, which moves the difficulty to the representation: over raw pixels the programs needed are hopelessly long.

The question this thesis takes up is whether that representation can itself be a large parameterized space, searched per task alongside the program and within a budget that fits one machine.

This thesis shows that it can. The evolutionary algorithm developed here co-evolves three components per task: a parser turning a grid into a typed graph of objects and relations, a short program of rewrite rules over that graph, and a projector rendering the result back. The graph's node and edge families follow what developmental psychology considers innate: objects, geometry, counting, and space. Parser, rule language and projector are designed and built here; lifting grids into a graph is not new, but this one is a large configuration of evolvable parameters in the same genome as the program. Candidates are ranked first on reproducing every train pair exactly, then by a graph distance over those that fail, with a penalty for programs that need every pair.

This work implements the solver twice, in Python and then in a faster C++ version. The best single run, from the C++ engine, fits the train pairs of 211 of the 391 ARC-AGI-1 training tasks common to both editions and reproduces the withheld pair on 163 within two attempts, with no neural network in the loop. Across runs, 217 of the 400 are solved at least once, though no single run reaches that.

The winning programs are tiny: one rewrite rule at the median, nine in ten

using at most two, which is the point of co-evolving the representation, since the parser absorbs the modelling that a pixel-level language would spell out. Two findings cut against expectation. The limit is not perception, since tasks narrowly missed are parsed no less consistently from pair to pair than those solved; it is the expressiveness of the rules. And the gap between that 217 and any one run is a failure of choice, not of search: fitting programs are found reliably, but the ranking cannot always tell which will generalize.

The ceiling is therefore the rules, not the machine. On ARC-AGI-2, built to demand the multi-step composition a one-rule program cannot express, the solver fits 3 of the 120 public evaluation tasks and solves 2, one already in ARC-AGI-1. Mining reusable strategies is one attempt: they do recur across a third of held-out tasks with multi-rule solutions, but the prior built on them does not yet pay.

Despite the name, solving ARC-AGI-1 and 2 is not solving artificial general intelligence (AGI), and the interest lies less in the score than in the route. Each solution is a short program that can be read and checked rather than a set of weights, and the run fits on ordinary hardware, so it suits settings where a decision must be inspected or compute is scarce.

# Table of Contents

**1 Introduction** 1
1.1 From a moving target to a measurable one . . . 1
1.2 Goodhart's law, reversed . . . 2
1.3 Contributions . . . 3
1.4 Structure of the thesis . . . 4

**2 ARC-AGI and the Measure of Intelligence** 5
2.1 Measuring intelligence . . . 5
2.2 The Abstraction and Reasoning Corpus . . . 6
2.2.1 The kinds of transformation . . . 9
2.3 A timeline of the benchmark . . . 9
2.4 The ARC Prize competitions . . . 11
2.5 The performance landscape . . . 11
2.6 Published solvers, side by side . . . 13
2.7 The bitter lesson, and its limits . . . 16
2.8 A taxonomy of approaches . . . 16

**3 Foundations** 18
3.1 Core knowledge and compositionality . . . 18
3.2 Program synthesis . . . 19
3.3 Library learning . . . 20
3.4 Genetic programming and evolutionary computation . . . 21
3.5 Neurosymbolic AI . . . 22
3.6 Related work on ARC . . . 23

**4 Representation and Language** 25
4.1 Overview . . . 25
4.2 The parser: from grid to typed multigraph . . . 27
4.3 The intermediate representation . . . 30
4.4 The domain-specific language . . . 31
4.5 The executor . . . 32

4.6 The projector . . . . . 33

**5 The Evolutionary Search** 35
5.1 The evolutionary optimizer . . . . . 35
5.2 The fitness function . . . . . 38
5.3 Implementation, determinism, and reproducibility . . . . . 40
5.4 Auditing the parser against ground truth . . . . . 41
5.5 Library learning: mining reusable macros . . . . . 43

**6 Experimental Evaluation** 45
6.1 Experimental setup . . . . . 45
6.2 Results on ARC-AGI-1 . . . . . 46
6.3 A worked example . . . . . 50
6.4 Program parsimony . . . . . 52
6.5 Failure analysis . . . . . 53
6.6 ARC-AGI-2 . . . . . 56
6.7 Head to head on ARGA's own subset . . . . . 57
6.8 ConceptARC . . . . . 58
6.9 Where is the bottleneck? . . . . . 60
6.10 Trajectory drift and the noise floor . . . . . 61
6.11 The fitness problem is real . . . . . 63
6.12 Lessons from the development record . . . . . 63
6.13 Limitations and threats to validity . . . . . 65
6.14 Macro-library ablation . . . . . 67

**7 Future Work and Conclusions** 69
7.1 Future work on the static benchmark . . . . . 69
7.2 Beyond static puzzles . . . . . 70
7.3 Broader impact . . . . . 70
7.4 Conclusions . . . . . 71

**A Domain-Specific Language Reference** 73
A.1 Node types . . . . . 73
A.2 Edge families . . . . . 73
A.3 The parser pipeline . . . . . 74
A.4 Selectors . . . . . 74
A.5 Actions . . . . . 76
A.6 Global transforms . . . . . 78
A.7 Guards . . . . . 79
A.8 Composition . . . . . 79

**B Parser, Projector, and Distance Configuration** 80
B.1 Parser parameters . . . 80
B.2 Projector policy axes . . . 80
B.3 Distance components . . . 81

**C Reproducibility** 84
C.1 What pins a run . . . 84
C.2 Configuration layering . . . 85
C.3 The reported configurations . . . 85
C.4 The development record . . . 86

**D Example Programs and Operator Arguments** 88
D.1 A gallery of programs . . . 88
D.2 Operator arguments . . . 91
D.3 The highest-support macros . . . 91

**E Notation and Abbreviations** 94

**Bibliography** 96

# Chapter 1

# Introduction

*Artificial intelligence* has never had a single agreed definition. In their standard textbook, Russell and Norvig do not settle on one, and instead lay out a two-by-two grid of what people have meant by the term over the years: thinking versus acting, and human-like versus rational [1]. The field has lived with this ambiguity since it was named in the proposal for the 1956 Dartmouth workshop, and by now most researchers have made peace with it [2]. The phrase has drifted far enough from its parts that "artificial" and "intelligence" no longer add up to its meaning; it reads as one word.

*Artificial general intelligence* (AGI) is the version of the term that still carries the field's original ambition. Goertzel and Pennachin described it as systems with "a reasonable degree of self-understanding and autonomous self-control" that can "solve a variety of complex problems in a variety of contexts" and, most importantly, "learn to solve new problems that they didn't know about at the time of their creation" [3]. The phrase "a reasonable degree" is doing a lot of work in that sentence. Self-understanding and autonomous self-control are not within reach of current systems, and there is no clear engineering path to them. This thesis sets AGI in that strong sense aside. What it does not set aside is the narrower, testable question underneath it: can a machine adapt to a new problem it has never seen before, from a handful of examples?

## 1.1 From a moving target to a measurable one

In 2019 François Chollet reframed the question in a way that made it measurable [4]. Rather than ask whether a system is intelligent, he proposed to measure how efficiently it acquires skill at tasks it was not built for: generalization, quantified. In the same paper he introduced the *Abstraction and Reasoning Corpus* (ARC): a set of small grid puzzles, each specified by a few input–output examples, that a solver

must complete by producing the grid the underlying rule would give. The puzzles are built on a handful of innate priors that Chollet calls *core knowledge* (objects and elementary physics, agentness and goal-directedness, numbers and counting, and elementary geometry and topology), and they avoid acquired knowledge like language or trivia. The design closes the usual escape route of "training on the test set": because every task is novel and the evaluation set is held out, a system cannot memorize its way to a good score. The gap this exposed was stark: people solve most of these tasks, while the best programs solved very few (Section 2.5), and that gap was the point.

The benchmark was later renamed ARC-AGI [5], then ARC-AGI-1 once successors arrived. ARC-AGI-2, released in 2025, kept the format but raised the difficulty, targeting the kinds of compositional reasoning that had started to fall to brute-force and to large models. ARC-AGI-3, released in 2026, changed the format entirely, moving from static puzzles to interactive, game-like environments. Alongside the official line, Moskvichev, Odouard, and Mitchell proposed *ConceptARC* [6], which organizes tasks by the specific abstraction each one exercises so that failures can be diagnosed rather than merely counted. Throughout this thesis, "ARC-AGI" is used in the broad sense to cover the static-puzzle benchmarks: ARC-AGI-1, ARC-AGI-2, ConceptARC, and their close relatives. ARC-AGI-3, being interactive, falls outside the scope of this thesis; Chapter 7 returns to what an interactive setting would demand of the representation.

ARC-AGI measures less than its name suggests. Chollet illustrates true broad generalization with Wozniak's coffee test [4], a robot walking into an unfamiliar kitchen and making a cup of coffee, a task that involves perception, manipulation, and open-ended physical interaction, and is far harder than any grid puzzle. ARC-AGI does not ask for that. Its tasks are well-defined and the space of possible transformations is bounded. This is not a criticism; it is what makes the benchmark tractable. Solving it does not amount to solving AGI, and this thesis does not claim otherwise. What the benchmark offers is a concrete, well-instrumented target for abstraction and program induction.

## 1.2 Goodhart's law, reversed

Goodhart's law warns that when a measure becomes a target, it stops being a good measure. The community's attention did shift, quickly, from "build AGI" to "score well on ARC-AGI." The argument of this thesis (and of the paper it grows from) is that in this particular case the shift has been productive rather than corrosive. Optimizing hard for the metric forces the very capabilities the metric was meant to stand in for: turning a scene into the right symbols, and finding a short program over those symbols that explains the examples. The metric is narrow enough to

attack directly and rich enough that the attack has to be principled.

Focusing on the metric also makes it possible to drop everything ARC-AGI does not require. There is no robot, no camera, no physical world; there is a grid of coloured cells and an output grid to construct. What remains is a two-part problem: choose a representation for the scene, and synthesize a procedure that operates on that representation. Both parts are a good fit for evolutionary computation. The representation can be assembled by tweaking and composing a bounded set of human-curated, domain-specific building blocks, which sits close to the core-knowledge and bootstrapping theories developed by Spelke, Carey, and colleagues since the 1990s [7, 8]. The procedure can be evolved with genetic programming, a lineage that runs back to Koza's work in the late 1980s [9]. An evolutionary algorithm can, in principle, search for the parser and the program at the same time, tailoring both to each task.

## 1.3 Contributions

This thesis designs, implements, and evaluates a graph-native co-evolutionary solver for ARC-AGI, and studies a library-learning extension for it. The specific contributions are:

- **A typed-multigraph representation grounded in core knowledge.** The solver parses each grid into a directed multigraph whose node and edge families track the core-knowledge priors: objects, geometry, counting, spatial relations, and a coarse physics. The parser is a configurable pipeline whose parameters sit in the same genome as the program, so the search can adapt the representation per task, although nearly half of the solved tasks need no adaptation at all (Chapter 6).

- **A graph-native domain-specific language and a co-evolutionary search.** Programs are expressed as typed rewrite rules over graph nodes rather than over raw pixels. The parser, the program, and the projector that renders a result graph back to a grid form a single individual, evolved under a per-task time budget in three phases. The fitness is lexicographic: it treats exact reproduction of the train pairs as a discontinuity to land on rather than a smooth surface to climb (Chapters 4 and 5).

- **A library-learning extension.** Reusable, role-typed *macros* are mined from previously solved programs and reinjected into the search as a sampling prior, in the spirit of DreamCoder's wake–sleep cycle. Transfer to unseen tasks is measured with a held-out parity split and the strategies do recur, but the ablation finds no held-out gain and the library ships disabled, so the

contribution here is the mechanism and a diagnosis of why it does not yet pay (Sections 5.5 and 6.14).

- **An empirical study on ARC-AGI-1 and ARC-AGI-2.** On the full 400-task ARC-AGI-1 training set, the solver reproduces the withheld pair on a third of the tasks, the whole run taking under three hours on a single 30-core machine; the faster C++ engine reaches 163 of the 391 tasks shared with ARC-AGI-2, and 217 of the 400 fall at least once across runs. The evaluation points at the domain-specific language (DSL) rather than the parser as the current bottleneck, quantifies the search's stochastic noise floor, and reports where simple additions help and where they do not. On the harder ARC-AGI-2 the same solver fits only 3 of the 120 public evaluation tasks and solves 2, only one of which is a task ARC-AGI-2 does not share with ARC-AGI-1, a negative result reported in full because it localizes the limit precisely (Chapter 6).

Parts of this thesis are drawn from a companion paper, *GP and the Quest for AGI: When Goodhart's Law Turned Out to Be Useful* [10].

## 1.4 Structure of the thesis

Chapter 2 tells the story of ARC-AGI: how Chollet's measure-of-intelligence framework gave rise to the benchmark, how the benchmark evolved across its versions, and where humans, large language models, and dedicated solvers currently stand. Chapter 3 lays the theoretical groundwork (core knowledge and compositionality from cognitive science, program synthesis and library learning from computer science, and genetic programming from evolutionary computation) and positions the present work against related ARC solvers. Chapter 4 presents the static half of the solver: the parser, the intermediate representation, the domain-specific language, the executor, and the projector. Chapter 5 presents the search over them: the evolutionary optimizer, the fitness function, the ground-truth machinery used to audit the parser in isolation, and the macro-mining library-learning extension of Section 5.5. Chapter 6 reports the experimental evaluation on ARC-AGI-1 and ARC-AGI-2 and analyzes where the approach succeeds and where it does not. Chapter 7 sets out future work and concludes. The appendices collect the reference material: the operator tables of the domain-specific language, the parser, projector, and distance configurations, what pins a run for reproducibility, example programs with their arguments, and the notation used throughout.

# Chapter 2

# ARC-AGI and the Measure of Intelligence

The benchmark this thesis attacks has a history worth knowing. It begins with the framework that motivated it, Chollet's 2019 attempt to make "intelligence" measurable, runs through the corpus itself and the three versions and variants it grew into, and ends where humans, large language models, and dedicated solvers currently stand. The aim is to give enough context that the design decisions in Chapter 4 read as responses to the benchmark's actual structure rather than arbitrary choices.

## 2.1 Measuring intelligence

Most AI benchmarks measure skill at a fixed task: classify these images, translate these sentences, win these games. Chollet's objection, laid out in *On the Measure of Intelligence* [4], is that skill at a known task says little about intelligence, because skill can always be bought with more data, more compute, or more hand-engineering. A chess engine is superhuman at chess and helpless at anything else. To measure intelligence, he argues, one should measure the *efficiency* with which a system turns experience into skill at tasks it was not prepared for: generalization, relative to the prior knowledge and experience the system was given.

The paper makes this precise. It distinguishes degrees of generalization, from absorbing new data points within a known distribution up to the "broad generalization" that lets an agent handle an open-ended range of tasks it was never designed for. It defines intelligence, informally, as skill-acquisition efficiency over a scope of tasks, holding priors and experience fixed. And it argues that a fair test must control for prior knowledge: a benchmark should be built on a small, explicit set of innate priors shared by the systems being compared, so that no contestant can win

by importing knowledge the others lack. For humans, Chollet takes those priors to be the *core knowledge* identified by developmental psychology [8], an inventory of objectness, elementary geometry and topology, numbers, and basic agent-like behaviour. The benchmark that accompanies the paper is designed to require only these.

One consequence of defining intelligence as an *efficiency* rather than a level of skill is worth drawing out, because Chapter 6 leans on it. Two systems that reach the same skill are not equally intelligent if one needed a thousand times the experience or a far richer set of priors. On this account a system that scores well by consuming enormous compute or task-specific data is buying skill rather than demonstrating intelligence, which is the lens this thesis applies to the large-model results.

## 2.2 The Abstraction and Reasoning Corpus

An ARC task is a small number of train pairs plus one or more test inputs. Each input and output is a grid of cells; grids range from $1 \times 1$ up to $30 \times 30$, and each cell carries one of ten colours (including the background). The solver sees the train pairs and must produce the exact output grid for each test input. The rule itself is never asked for and never scored. A task is solved only if the predicted grid matches the ground truth cell-for-cell; there is no partial credit.[1] The transformations are varied: recolouring, translation and reflection, gravity, object counting, symmetry completion, cropping to a region of interest, tiling, and many compositions of these. What they have in common is that a human usually recognizes the rule from one or two examples, and that the rule is almost never expressible as a simple pixel-level filter. Figure 2.1 is one task in full, and it makes the point better than the description does: most readers will have the rule before finishing the second pair, and none of them will have written it down.

The corpus that shipped with the 2019 paper contained 1 000 tasks, split into a public training set of 400, a public evaluation set of 400 for validating approaches, and and 200 hidden tasks, split into a 100-task semi-private set and a 100-task private set, used to score entries in the competition.[2] The split matters. A solver may be developed against the training tasks, but the evaluation and test tasks are novel, and the two sets are meant to share format without sharing solutions. This structure is what lets ARC sidestep the "training on the test set" problem that

[1] The public competitions allow a small number of attempts per test input (historically up to three, later two) and count a task as solved if any attempt is exact.

[2] ARC-AGI-1, `https://github.com/fchollet/arc-agi`. See Section 2.3 for the later versions, whose splits differ.

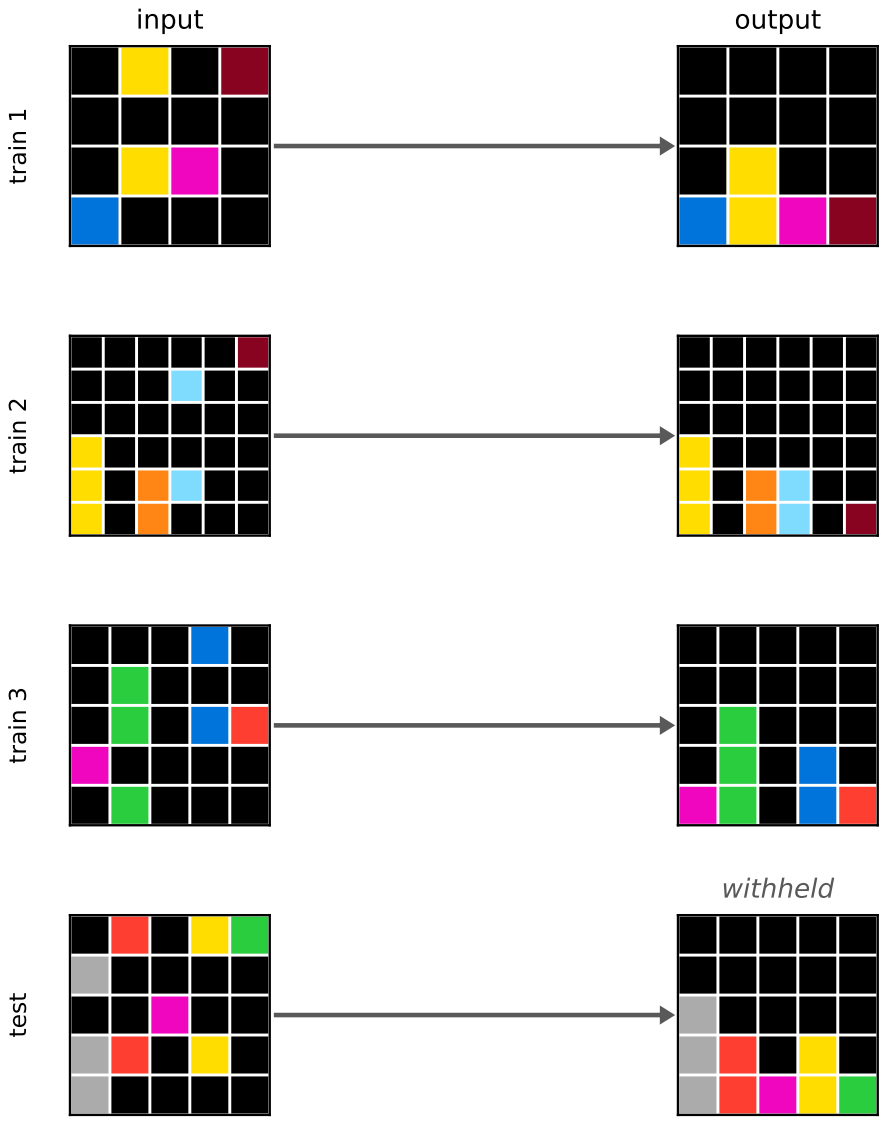



**Figure 2.1:** One ARC-AGI-1 task, `1e0a9b12`, as a solver receives it: three train pairs and one test input, whose output is withheld. Every object falls to the bottom of its column, keeping its colour and its order. Nothing in the task states that rule, and nothing asks for it; what is scored is the withheld grid alone.

had quietly inflated scores on earlier benchmarks: there is no distribution to fit, only new problems to solve.

Two properties of ARC shape the design that follows. First, the tasks are *few-shot*: two to five train pairs is typical, so a solver cannot rely on statistical estimation and must instead commit to a hypothesis from very little evidence. Second, the answer is a *constructed object*, not a label: the solver emits a whole grid, which makes the output space enormous and makes any smooth notion of "almost right" delicate to define. Both properties recur as design pressures in later chapters: the first motivates a strong representational prior, the second complicates the fitness function.

**What a task looks like.** A concrete example fixes the intuition. In a common kind of ARC task, each train input shows a few coloured shapes scattered on a black background, and each corresponding output shows the same shapes with the single largest one recoloured (to red, say) and the rest left alone. A human reads the rule off the first pair, checks it against the second, and applies it to the test input without hesitation: "paint the biggest object red." Nothing in the pixels announces this rule. The solver has to notice that the grids decompose into objects, that objects have a size, that one object is largest, and that the transformation is a recolouring keyed to that size. It also has to prefer this hypothesis over the endless alternatives that also fit two or three examples ("paint the object in the top-left red," "paint the object with five cells red," and so on) but would fail on the test input. The difficulty is never perception in the machine-vision sense; the grids are tiny and noiseless. The difficulty is committing to the *right abstraction* from a handful of examples. This is why a representation that surfaces objects and their sizes as first-class things, rather than leaving them implicit in a pixel array, is such a natural fit: it turns "paint the biggest object red" into a short, findable program instead of a search over pixel maps.

**Why not a neural classifier.** It is tempting to ask why a large model trained on many ARC tasks cannot simply learn the distribution of transformations. The benchmark is built to defeat that move. Because every evaluation task is novel and the transformations are drawn from an open-ended space, there is no fixed distribution to fit: a model that has memorized "paint the biggest object red" from training has learned nothing that transfers to a test task whose rule is "reflect each object across the vertical axis." Skill at the seen tasks does not convert into skill at the unseen ones, which is the property Chollet's framework was designed to reward, and the property ordinary supervised learning does not have.

### 2.2.1 The kinds of transformation

The transformations that ARC tasks ask for do not form a tidy taxonomy, and part of the benchmark's difficulty is that they resist one, but a few recurring families are worth naming, because the solver of Chapter 4 is organized around being able to express them. *Recolouring* tasks change the colours of objects according to some rule (their size, their shape, their count, their relation to other objects) while leaving positions fixed. *Geometric* tasks apply a rigid motion (translation, reflection, rotation) to the whole grid or to individual objects. *Counting* tasks turn a count into an output: draw as many marks as there are blue objects, or keep only the colour that appears most often. *Topological* tasks turn on containment and connectivity: fill the interior of every closed loop, or remove the objects that touch the border. *Symmetry* tasks complete a partially drawn symmetric figure or repair a grid that is symmetric except for a defect. *Assembly* tasks compose smaller pieces into a larger structure, or conversely crop a region of interest out of a busy scene. Many real tasks combine two or three of these, and a good number rest on a one-off idea that fits no family at all.

Two things about this list matter for the design. First, most of the families are about *objects and their relations* rather than about individual pixels, which is the empirical case for an object-centric representation: a solver that can name the objects, their sizes, their colours, and their spatial and containment relations has the right vocabulary for recolouring, counting, and topological tasks almost for free. Second, the families are just the ones a small set of core-knowledge priors would predict (objects, geometry, number, topology), which is not a coincidence, since Chollet built the corpus to require those priors and little else. The mapping from these families to the parser's node and edge types, and to the DSL's selectors and actions, is the subject of Chapter 4; the point here is that the families are nameable at all, and that their shared object-centric character is what a graph representation is built to exploit.

## 2.3 A timeline of the benchmark

**ARC (2019).** The corpus was released alongside the measure-of-intelligence paper [4] and, in early 2020, as a Kaggle competition, the "Abstraction and Reasoning Challenge". The winning entries were hand-built program search engines over domain-specific operations, not learned models, and they topped out well below human performance. That result set the tone for the next several years.

**ARC-AGI-1 (rebranding).** As the benchmark gained visibility, it was renamed ARC-AGI and given a dedicated home under the ARC Prize foundation [5], a non-profit that now runs the competitions, which began in 2024, and maintains

the benchmark.[3] The tasks did not change; the "-1" suffix was added retroactively once ARC-AGI-2 was in preparation. Most of the literature refers to this first version, and so do the bulk of the experiments in this thesis; Chapter 6 also reports a smaller set of results on ARC-AGI-2.

**ARC-AGI-2 (2025).** The second version was developed to head off saturation on the older dataset. It kept the grid format but rebuilt the task set to resist the strategies that had begun to make headway on version one, in particular brute-force program search and large-model pattern completion. Its tasks lean more heavily on multi-step compositional reasoning and on applying a rule in a way that depends on context, and they were calibrated so that humans still solve them readily while machine solvers drop sharply. The split is also differently shaped. ARC-AGI-2 ships 1 000 training tasks and three separate 120-task evaluation sets: a public one, a semi-private one, and a private one, the last two reserved for competition scoring. Any result in this thesis on "the 120 evaluation tasks" is on the *public* split; the private set has no result here.[4] Tasks in both ARC-AGI-1 and ARC-AGI-2 typically give two to five train pairs. One test input is the norm on ARC-AGI-1 and on the ARC-AGI-2 training split, while the ARC-AGI-2 public evaluation split carries two or more on nearly two tasks in five.

**ARC-AGI-3 (2026).** The third version breaks with the static-puzzle format altogether. Instead of input–output grid pairs, it presents interactive, game-like environments in which an agent must act over time to reach a goal, testing abstraction alongside planning, exploration, dynamic rule discovery, and the use of a learned model of the environment.[5] Because the "map an input graph to an output graph" framing no longer applies, ARC-AGI-3 needs a different solver architecture, which this thesis does not attempt; Chapter 7 sketches what an interactive setting would demand of the representation developed here. The Foundation reports that humans solve 100 % of its environments, while frontier systems scored below 1 % as of March 2026 [11].

**ConceptARC (2023).** Independently of the official versions, Moskvichev, Odouard, and Mitchell built ConceptARC [6], a benchmark organized around specific spatial and semantic concepts ("same," "inside," "count," and so on) with several tasks per concept. Where ARC reports a single solved-count, ConceptARC lets a researcher

[3] `https://arcprize.org/`

[4] ARC-AGI-2, `https://github.com/arcprize/ARC-AGI-2`.

[5] ARC-AGI-3, `https://github.com/arcprize/ARC-AGI-3-Agents`.

see *which* abstractions a system has and which it lacks, which makes it a useful diagnostic companion to the headline benchmark.

## 2.4 The ARC Prize competitions

ARC has been driven forward as much by competitions as by papers. The original 2020 Kaggle challenge established the baseline that dedicated program-search systems could solve a meaningful fraction of the private test set while learned models could barely start. Beginning in 2024, the ARC Prize was run as an annual public competition with a substantial prize pool and an explicit grand-prize threshold: 85 % on the private test set, which at the time of writing remains unclaimed under the competition's efficiency constraints.[6] Two rules shape what the leaderboard means. Entries eligible for monetary prizes must make their code public, and separate prizes are awarded for papers describing an approach in more detail, so the competition produces a literature and not only a ranking. And the rules reward efficiency as well as accuracy: a solver that reaches a score at enormous computational cost is treated differently from one that reaches it cheaply, which is directly relevant to the argument of this thesis and is why the Kaggle-validated figures in Section 2.5 sit so far below the headline ones.

## 2.5 The performance landscape

**Humans.** On ARC-AGI-1, ordinary humans do well, and the figure depends on which split is meant. The H-ARC study put 1 729 people through five tasks each, drawn from all 400 training and all 400 evaluation tasks [12, 13]: average performance is 76.2 % on the training split and 64.2 % on the public evaluation split, with 790 of the 800 tasks solved by at least one participant within three attempts. The training-split figure is the one to compare against this thesis, whose own results are on that split. A 2026 survey quotes the same study as an average worker solving 77 % of the *evaluation* tasks [14], which reads a training-split figure as an evaluation-split one; the primary figures above are used here instead. The same survey reports that each ARC-AGI-2 task is solved by roughly 75 % of those who attempted it, which is not the same as a task-solve rate over the whole split. A separate psychometric validation with a hundred participants finds ARC-AGI correlates substantially with figural fluid intelligence [15]. Averages vary with the protocol and the split, so any single number should be read as a band.

[6] The grand prize requires 85 % on the held-out private set within fixed compute and cost limits.

**Language models.** Off-the-shelf large language models, prompted directly, do poorly. Public trackers put plain LLMs at around 10 % on ARC-AGI-1 and around 1 % on ARC-AGI-2, pooling runs whose split is not always identified, so both are best read as bands.[7] This is not surprising: the tasks are novel by construction, and a model that has learned a distribution has little to fall back on when the test item is outside it. The gap to the human evaluation-split average is roughly fifty-four points on the first benchmark, and wider still on the second, which is the discontinuity Chollet's framework was built to expose.

**Reasoning models and the o3 moment.** The picture shifted with the arrival of "reasoning" models, which spend extra inference-time computation on a chain of intermediate steps [16, 17] and are variously abbreviated LRM, for large reasoning model, or RLM; Table E.2 lists the abbreviations used in this thesis. In late 2024, OpenAI reported that its o3 model reached high accuracy on a subset of ARC-AGI-1, a result widely read as a breakthrough.[8] By 2026 the headline figures are much higher, and every one of them carries a price. A living survey of the benchmark's progress records Gemini 3 Deep Think at 96.0 % on ARC-AGI-1 at US$7.17 per task and Opus 4.6 at 93.0 % at $1.88, with the same two systems reaching 84.6 % (at $13.62 per task) and 68.8 % respectively on ARC-AGI-2 [14]. Two of those four figures need a caveat the survey itself supplies: once it filters for evaluation rigour, the maxima it endorses are 93.0 % on ARC-AGI-1 and 68.8 % on ARC-AGI-2; the 96.0 and the 84.6 belong to a system its filter excludes, so the same document supports two different answers. All four are on the semi-private held-out splits of 100 and 120 tasks, not on the public sets. Quoting the accuracy without the price beside it is what makes these results look like a settled ceiling rather than a purchase.

Two features of those numbers matter more than their size. The first is cost. The o3 result cost about US$4 500 per task; GPT-5.2 Pro reached 90.5 % at $11.64, a 390-fold reduction in little more than a year, which the survey attributes largely to running fewer samples in parallel rather than to a better algorithm [14]. Cheaper, then, but still dollars per puzzle, and still bought with test-time compute.

The second is what happens when that compute is taken away. Under the competition's resource constraints the same benchmark looks entirely different: the best entry on ARC-AGI-2 in 2025, NVARC, scored 24.03 %, and the ARChitects

[7] `https://ai-stats.phaseo.app/benchmarks/arc-agi-1` and `https://ai-stats.phaseo.app/benchmarks/arc-agi-2`.

[8] `https://arcprize.org/blog/oai-o3-pub-breakthrough`. Few details were released to the research community, and the result attracted immediate scrutiny over how much of it rested on massive task-specific sampling and pre-training, and over its cost per task, which was reported to be very high [18].

16.5 %, against the 68.8 % the same survey endorses for the unconstrained leaderboard once it filters for evaluation rigour [14]. The obvious conclusion, which the survey also draws, is that a gap of nearly forty-five points between constrained and unconstrained regimes suggests current progress reflects computational investment as much as architectural insight. That gap is the space this thesis works in, and it is why cost is reported here alongside every count.

These figures and the ones in Chapter 6 are not measured on the same footing, and Section 6.2 states the comparison properly. The short version is that the big-lab numbers come from enormous inference budgets and sometimes task-specific training, while the system described here completes the whole 400-task run in under three hours on a single 30-core workstation, so the quantity worth comparing is cost per task solved rather than accuracy alone.

## 2.6 Published solvers, side by side

Aggregate leaderboard figures hide what kind of system produced them. Table 2.1 collects published ARC-AGI-1 results by approach, which is the comparison Chapter 6 is eventually measured against. The splits differ across its rows: ARGA's 160 are drawn from the ARC-AGI-1 *training* split, the same one this thesis reports on, while the language-model rows and the minimum-description-length row are on the public *evaluation* split, and icecuber's figure is on a 100-task secret set.

**Table 2.1:** Published approaches on ARC-AGI-1, with the present work for reference. "Tasks" records how many tasks the system was evaluated on. Scores are given as a fraction of the tasks attempted and, in the last column, renormalized over 400 by counting unattempted tasks as failures. The rows are not all on the same 400, so the last column is comparable in magnitude and not in provenance; the splits are given in the text below.

| Approach | Type | Tasks | Attempted | Of 400 |
|---|---|---|---|---|
| The ARChitects [19] | Direct grid prediction with LLMs and depth-first search; winner of the 2024 Kaggle competition | 400 | 71.6 % | 71.6 % |
| BARC+TTT [20] | Combination of the two below; the authors report it as matching average human performance | 400 | 61.9 % | 61.9 % |
| BARC [20] | Induced-program library with an LLM proposer; as measured by Akyürek et al., whose own ensemble figure is higher [21] | 400 | 54.4 % | 54.4 % |

*Table 2.1, continued*

| Approach | Type | Tasks | Attempted | Of 400 |
|---|---|---|---|---|
| TTT [20] | Direct grid prediction, LLM fine-tuned at test time | 400 | 53.0 % | 53.0 % |
| icecuber [22] | Hand-built DSL with brute-force search | 100 (secret) | 20.6 % | — |
| ARGA [23] | Program synthesis over a menu of hand-written graph abstractions, chosen adaptively inside a greedy best-first search | 160 | 35.6 % | 14.2 % |
| GPAR [24] | Generalized planning over improved graph abstractions, with a PDDL-style action language; evaluated on ARGA's 160 | 160 | 50.6 % | 20.2 % |
| MDL [25] | Program synthesis with a minimum-description-length objective | 400 | 5.8 % | 5.8 % |
| TransCoder [26] | Neural program synthesis, recurrent networks and variational autoencoders | 400 | 2.0 % | 2.0 % |
| This thesis | Evolved parser, program and projector; best single Python run at two attempts (Chapter 6) | 400 | 34.5 % | 34.5 % |
| This thesis | Same solver, C++ reimplementation at two attempts, on the tasks shared with ARC-AGI-2; the strongest run on record | 391 | 41.7 % | 40.8 % |
| This thesis | Same C++ engine on ARGA's own 160 tasks, at 840 s against their 1 800 s, single attempt to match those rows (Section 6.7) | 160 | 46.9 % | 18.8 % |
| This thesis | Union over all 96 development runs, pooling budgets, seeds and builds (an oracle, not a configuration) | 400 | — | 54.3 % |

Four things in this table matter for what follows. First, the top of it is occupied by large language models. Second, the classical program-synthesis entries sit lower, though how much lower depends on what they were measured on. On ARGA's 160-task subset both ARGA and GPAR clear a third, and GPAR reaches

50.6 percent; measured over a whole split the family runs from 5.8 percent for a minimum-description-length objective down to 2 percent for a neural synthesizer, with a hand-built DSL reaching 20.6 percent on the 100-task secret set. That spread, rather than any single figure, is the gap this thesis is trying to close from the symbolic side.

Third, ARGA is the closest published relative of the present work: it also lifts grids into a graph of objects and relations and searches for a transformation over that structure, and its representation is not simply fixed either. ARGA defines several hand-written abstractions, each a different rule for what counts as an object, and because the graphs they produce share one structure they share one search tree; a tabu list then suspends any abstraction whose branches are producing worsening results. The representation is therefore chosen *during* the search rather than before it. Two differences remain, and they are of degree rather than kind. ARGA selects from a small menu of whole abstractions, where the solver of Chapter 4 evolves a configuration of forty-four parameters over a seventeen-pass pipeline, so the space of available representations is combinatorial rather than enumerable; and that configuration is carried in the same genome as the program, so representation and program are varied by the same operators under one fitness, rather than one being searched over while the other is scored. ARGA's search is also a greedy best-first tree search with constraint acquisition rather than a population-based stochastic one. Its headline 35.6 percent is also measured on 160 tasks its authors selected from the 400, so it is not a whole-benchmark figure. Renormalizing over all 400 gives 14.2 percent, which assumes the 240 unattempted tasks would all have failed; since the selected subset was presumably the one the representation suited, the true whole-benchmark figure lies somewhere between the two, and the honest reading is that a single number cannot be quoted for it. This is the row against which this thesis's own results should be judged most carefully, and Section 6.7 settles it by running the same 160 tasks directly rather than arguing about renormalization. GPAR, which came a year later, reuses ARGA's abstractions and its subset and replaces the search with generalized planning over a PDDL-style action language, reaching 50.6 percent; it is the strongest published figure on that subset and the more demanding comparison of the two.

Fourth, the last two rows deserve their distinction. The 34.5 percent is one run at a fixed seed and budget, directly comparable to the rows above. The 54.3 percent is the union of every development run ever made, 96 of them, pooling different budgets, seeds and builds; it is an upper bound on what the current operator set can reach rather than a result any single configuration delivers. It is reported because the two figures together bracket the approach: a third of the benchmark is reproducible on demand, and just over half is reachable in principle without changing the language. Section 6.2 gives the derivation and the reason the gap between them is informative rather than embarrassing.

## 2.7 The bitter lesson, and its limits

In 2019 Richard Sutton wrote a short, influential essay, "The Bitter Lesson" [27], arguing that the methods which win in the long run are the general-purpose ones that scale with computation, and that human-designed structure tends to be a crutch that scaling eventually discards. Through the early 2020s the trajectory of machine learning seemed to confirm it: more parameters, more data, more compute, better results.

ARC-AGI complicates that story. The very organizations best positioned to scale, among them Anthropic, Google, and OpenAI, have not reached human-level performance on ARC-AGI-2, and the gains they have made come at steep cost with diminishing returns. There is evidence, too, that their apparent reasoning degrades as a problem's compositional depth grows [28]. One reading, and the one this thesis adopts, is that today's large models are not actually the "general-purpose methods" Sutton had in mind. The computational model of a language model is simple, and the "reasoning" bolted onto the reasoning models (chain-of-thought prompting, fixed thinking budgets [29]) is a coarse approximation of deliberation rather than the real thing. Sutton's own advice cuts both ways: he warns against baking in "simple ways to think about ... space, objects, ... or symmetries." The wager here is that a *slightly* more general-purpose method can reach these benchmarks at a small fraction of the cost, and that the right amount of built-in structure is more than zero. Such a method would be built on a bounded, human-curated set of parametric building blocks, and made to scale by search rather than by gradient descent.

## 2.8 A taxonomy of approaches

Solvers proposed for ARC-AGI fall into a few broad families, which Chapter 3 revisits in detail:

- **Program synthesis over a DSL.** A domain-specific language of grid operations is fixed in advance, and a search (brute-force, guided, or learned) looks for a short program that fits the train pairs. The 2020 Kaggle winner [22] and Michael Hodel's widely used `arc-dsl` [30] are canonical examples.
- **Large language models, prompted or sampled.** An LLM is asked to produce the answer grid directly, or to write a program that produces it, often with many samples and a filter. Reasoning models are the recent, higher-scoring members of this family.
- **Test-time training.** A model is fine-tuned on the train pairs of each task at inference time, adapting to the specific puzzle before answering.

- **Graph- and constraint-based methods.** Grids are lifted into a structured representation of objects and relations, and the transformation is found by reasoning over that structure. Xu et al. [23] represent ARC grids as graphs and search a graph-level language over them; the solver in this thesis is in this family, but differs in both its representation and its search.

The approach developed here belongs to the last family and borrows the "search for a short program" objective from the first. Its distinguishing bet is that the *representation* should itself be evolved, task by task, rather than fixed, and that the representation should be a typed multigraph whose vocabulary is chosen to match the core-knowledge priors the benchmark is built on. The theoretical grounding for that bet is the subject of the next chapter.

# Chapter 3

# Foundations

The solver in this thesis sits at the meeting point of three research traditions: cognitive science, which supplies the idea of a small set of innate priors that make abstraction possible; program synthesis and library learning, which supply the idea of searching for a short program and reusing its parts; and genetic programming, which supplies the search itself. Each is reviewed in turn below, and the work is then placed against existing ARC solvers.

## 3.1 Core knowledge and compositionality

*Core knowledge* refers to the set of innate cognitive capacities that humans are born with: estimating quantities, tracking objects through occlusion, predicting how things move. Separating innate from learned ability is hard, and the foundational work, led by Elizabeth Spelke and Susan Carey, proceeds largely by testing infants and diverse populations and looking for what is shared across all of them [7, 8, 31].

Spelke groups core knowledge into a handful of systems: *objects* (bounded, solid, cohesive things that persist and move on continuous paths), *places* (the geometry of the environment humans navigate), *number* (approximate and exact quantity), *forms* (abstract shapes and their relations), *agents* (entities with goals and autonomous motion), and *social* understanding. These systems are useful because they are cheap approximations: intuitive physics gives fast, usually-correct predictions from a coarse, non-quantitative model of the physical world [32]. Chollet's choice to build ARC on objectness and elementary physics, agentness, number, and elementary geometry and topology [4] is a direct borrowing from this literature, and it is why a representation organized along the same lines is a natural fit for the benchmark.

The mapping from these systems to the design of Chapter 4 is close enough to be worth drawing out, because it is the thread that connects the cognitive-science

motivation to the engineering. The *object* system is the parser's reason for existing: its first job is to turn a grid of cells into bounded, cohesive `SubObject`s, the "things that persist" of the object core. The *places* and *forms* systems show up as the parser's geometry: zones, bounding boxes, local coordinates, shape hashes, and the alignment relations that say two objects share a row or lie on a diagonal. *Number* appears wherever the solver counts: the mass of an object, the count guards on a rule, the selectors that pick the most or least frequent colour. The naive-physics fragment of the object and agent systems is the gravity and bridge machinery, "which object would fall onto which," "which gap could be bridged." The two systems the static solver leaves almost untouched are *agents* and *social* understanding, and that is a gap rather than a deliberate exclusion: agentness is on Chollet's own list, and a task turning on an agent's goals is one the parser has nothing to offer for (Chapter 4). Such tasks are a minority of the static corpus, and the interactive setting of Chapter 7, where an action and its consequence have to be represented, is just where the design has to be extended. The point is not that the parser is a faithful model of human cognition, which it is not, but that its vocabulary was chosen from the same short list that the benchmark was built to require, so its gaps and the benchmark's demands line up.

What separates human cognition from that of other animals, which share much of the same core inventory, is *compositionality*: the capacity to build complex meanings out of simple parts according to systematic rules, so that a finite vocabulary yields an unbounded range of thoughts. On one influential account, tracing to Chomsky, the source of this capacity is the language faculty and its generative grammar [33]. A related line formalizes an internal *language of thought*, a mental language whose symbols recombine into abstract programs while grounding out in the older, nonverbal core-knowledge representations [34]. The connection this thesis leans on is a simple one: these hierarchical structures over primitive concepts look a great deal like computer programs, an analogy that recent work in cognitive science has made explicit by modelling human perception of abstract shapes as short programs in a geometric language [35]. If human abstraction is program-like over core-knowledge primitives, then a solver that searches for short programs over a core-knowledge-flavoured representation becomes a plausible model of the competence being tested, and not only a convenient piece of engineering.

## 3.2 Program synthesis

Program synthesis is the task of automatically constructing a program that meets a specification. The specification can be a logical formula, in which case classical *deductive synthesis* derives a program from a constructive proof that a solution exists [36]; or it can be a set of input–output examples, in which case *inductive* or

programming-by-example synthesis must generalize from the examples to a program that also handles unseen inputs. ARC is squarely an inductive-synthesis problem: the train pairs are the specification, and the whole difficulty is generalizing correctly from a handful of them.

Two ideas from this field carry most of the weight here. The first is the *domain-specific language*. Rather than search over a general-purpose language, synthesis fixes a small DSL whose primitives are chosen for the problem domain; this shrinks the search space to something tractable and bakes in useful bias. FlashFill, which synthesizes spreadsheet string transformations from examples, is the canonical practical success [37], and syntax-guided synthesis (SyGuS) later gave a common formal framework for "search over a grammar subject to a specification" [38]. The DSL of Chapter 4 follows this playbook, with the twist that its primitives operate on graph nodes rather than on values or pixels. The second idea is *search with structure*: rather than enumerate programs blindly, systems like Sketch let a human supply a partial program with holes and have the synthesizer fill them [39]. The three-component individual of this thesis, parser plus program plus projector, is a variant of the same intuition, with the parser configuration playing the role of the structural prior that makes the program search cheap.

It helps to see where an evolutionary approach sits among the search strategies the field has tried, because the choice determines what kind of fitness signal is usable at all. Enumerative synthesis searches programs in order of size, smallest first, and is complete (given enough time it will find the shortest solution), but its cost grows exponentially in program length, so it lives or dies by how aggressively the DSL and a cost model prune the frontier. Constraint-based synthesis compiles the specification into a logical formula and hands it to a solver, which is powerful when the specification is crisp but strains when "correct" means matching a handful of grids exactly with no partial-credit structure to exploit. Stochastic search, of which evolutionary computation is one instance, gives up completeness in exchange for two things ARC rewards: it can run under a fixed time budget, stopping with the best candidate found so far rather than running to exhaustion, and it can be steered by a graded fitness signal even where the specification itself is all-or-nothing. That second property is decisive here, because the central difficulty of ARC fitness (Chapter 5) is manufacturing a graded signal from an all-or-nothing target, and only a search that can *use* such a signal benefits from building it.

## 3.3 Library learning

A fixed DSL forces a trade-off: rich primitives make short programs but a large branching factor, while minimal primitives make a small branching factor but long, hard-to-find programs. *Library learning* escapes the trade-off by growing the DSL

over time, mining reusable abstractions out of the programs found so far and adding them back as new primitives.

The reference system is DreamCoder [40], which alternates a *wake* phase that searches for programs solving the current tasks with a *sleep* phase that refactors the solutions, extracts common fragments into named abstractions, and trains a neural recognition model to guide the next wake phase. Over iterations the library climbs from primitives to domain concepts, and problems that were once out of reach become short programs in the enriched language. Later work made the abstraction step both principled and fast: Stitch searches efficiently for the fragments whose extraction most compresses a corpus of solutions, replacing DreamCoder's slower version-space refactoring [41].

Section 5.5 adapts this idea to the present setting. The solver's "sleep" step mines role-typed macros (recurring selector/action skeletons with typed holes) from programs that have solved training tasks, and reinjects them into the evolutionary search as a sampling prior. The design keeps the DreamCoder spirit, but the constraint it has to respect is the one it runs into: in an evolutionary loop a change to the sampling distribution can do more harm through search-trajectory disruption than good through better priors, and Section 6.14 reports that this is what happens.

## 3.4 Genetic programming and evolutionary computation

Genetic programming (GP) evolves programs directly, treating a population of candidate programs as an evolving population under selection, mutation, and recombination. The lineage begins with Koza, who represented programs as expression trees and showed that crossover over subtrees could discover working programs for a range of problems [9, 42]. Because unrestricted tree crossover readily produces type-incorrect or meaningless programs, later work introduced typing: strongly-typed and grammar-guided GP constrain the operators so that only well-formed programs are generated [43, 44], which is essential when the primitives have distinct types, as the graph-node operators of this thesis do. *Byron* and its MicroGP predecessors, developed in the group where this thesis was written, constrain candidate programs to a declared instruction library for the same reason [45, 46].

The mechanics matter for reading Chapter 5, so it is worth fixing the vocabulary. A GP run maintains a *population* of candidate programs; each is scored by a *fitness* function; *selection* picks parents in proportion to fitness (or by a tournament among random subsets); *crossover* recombines two parents into offspring; and *mutation* perturbs an offspring at random. *Elitism* carries the best individuals forward

unchanged so that a good solution is never lost to the randomness of the next generation, and the loop repeats until a budget of generations or wall-clock time runs out. Two design choices in this general recipe recur throughout the present work. The first is *typing*: because a selector that reads cavity nodes matches nothing on a graph with no cavities, the mutation and crossover operators must respect the types of the things they edit, or the population fills with rules that are skipped silently at execution and so carry no information for selection. The second is the *selection pressure*: too much and the population converges prematurely onto one mediocre lineage, too little and the search is a random walk. The lexicase selection adopted in Chapter 5 is a modern answer to that balance, keeping specialists that a simple fitness average would discard [47, 48]; the $\varepsilon$ variant the solver uses relaxes the per-case pass condition so that it works on continuous errors rather than exact hits [49].

One phenomenon from evolutionary computation shapes the experimental methodology of Chapter 6: *neutrality* and *drift*. When many genotypes map to the same or nearly the same fitness, a population wanders across the neutral network without fitness pressure [50]. That literature studies what neutrality does for search, and finds it productive rather than merely noisy: neutral networks widen the set of productive pathways. The inference drawn here is a different one, and it belongs to this thesis rather than to that work. If the wandering is not itself under selection, then small changes to the operators or to the random-number draw order shift where a run ends up without changing the method's true quality. In this solver that shows up as a run-to-run swing in the number of tasks solved that is unrelated to any code change, a noise floor that any honest comparison of two variants has to clear. Recognizing it as neutral drift rather than signal came late: eight development phases had already been reverted on evidence that turned out to be noise (Section 6.10), which is why Chapter 6 measures the floor rather than assuming it.

## 3.5 Neurosymbolic AI

Between the symbolic and sub-symbolic poles that AI has oscillated between for decades sits *neurosymbolic* AI, which tries to combine the pattern-recognition strengths of neural networks with the compositional, interpretable reasoning of symbolic systems [51]. Kautz's taxonomy organizes the many possible couplings of the two [52]. It ranges from loose arrangements where a neural network and a symbolic reasoner call each other as black boxes, through designs where symbolic knowledge is compiled into a neural network's structure or loss, to tight integrations where neural and symbolic computation interleave within a single inference. The distinctions matter because they mark different answers to the same question,

namely which part does the perceiving and which does the reasoning, and different answers carry different costs in data, compute, and interpretability.

The solver in this thesis sits firmly on the symbolic side. The modelling work is done by explicit graph construction and explicit programs, with no learned component in the loop, but the framing matters for two reasons. First, it clarifies what the approach gives up (learned perception) and what it gains (interpretability and cheapness): a solved task yields a human-readable program, and the whole run fits on a workstation, but the parser cannot learn to see a structure its hand-written passes do not already detect. Second, it points to where neural components would help most, and the future-work chapter takes up both: the interactive extension of Chapter 7 would need a learned model of the environment, and the fitness problem of Chapter 5 invites Chollet's suggestion of a learned discriminator in the spirit of AlphaGo [53, 54], which Chapter 7 takes up. Both would move the design a step along Kautz's spectrum, from purely symbolic toward a hybrid, at the two points, perception and evaluation, where a learned prior is most likely to earn its cost.

## 3.6 Related work on ARC

The families sketched in Section 2.8 can now be made concrete. Program-synthesis-over-a-DSL solvers, from the 2020 Kaggle winner [22] onward, fix a library of grid operations and search for a fitting program; Michael Hodel's `arc-dsl` [30] became a common substrate. Ferré's system sits closer to the object family below: it builds object-centric models of each task and selects among them by minimum description length [25]. LLM-based solvers either emit the answer grid [19] or, more effectively, write programs and filter by execution [20]; the recent reasoning models are their higher-scoring descendants. Test-time-training solvers adapt a model to each task's train pairs before answering [20]. Graph- and constraint-based solvers lift grids into objects and relations: Xu et al. [23] represent ARC grids as graphs and search a graph-level domain-specific language over them, using acquired constraints to prune the search rather than to define it.

Each family answers the two questions every ARC solver must answer, how to represent a grid and how to search for a transformation, and it is worth drawing those answers out, because the present work makes an unusual choice on the first. The DSL-synthesis family fixes the representation implicitly in the primitives: `arc-dsl` operates on grids and object-lists with a hand-chosen vocabulary, and the search is an enumeration or a guided walk over programs in that vocabulary. Its strength is that a well-chosen primitive makes the target program short; its weakness is that a task whose natural primitive is missing is unreachable, and the vocabulary is fixed at design time. The LLM family pushes the representation into the weights: a model that has read enough code and enough puzzles carries an

implicit prior over transformations, and "search" becomes sampling, sometimes of answer grids directly and more effectively of programs that are then filtered by execution against the train pairs. Its strength is breadth; its weaknesses are cost and the absence of any guarantee that a sampled program means what it appears to. The test-time-training family sits between the two, adapting a model's weights to each task's handful of pairs before answering, which trades inference cost for a per-task prior.

The graph-and-constraint family, to which this work belongs, makes the representation *explicit*: it lifts the grid into objects and relations and reasons over that structure directly. ARGA (Section 2.6) is the published exemplar: it parses the grid as a graph and poses the transformation as a program search over it, which makes the relational structure first-class in a way a flat DSL does not. The present work shares that commitment to an explicit object-relational representation but differs on two axes. First, the search is evolutionary rather than constraint-solving or enumerative, which trades completeness for the ability to work under a wall-clock budget and to accept a graded fitness signal below the exact-match frontier. Second, the representation is not fixed: the parser that builds the graph is itself part of the individual under selection.

Against this backdrop, the contribution of this thesis is a specific combination. Like the DSL-synthesis family, it searches for a short program and treats exact reproduction of the examples as the objective. Like the graph family, it reasons over objects and relations rather than pixels. Unlike either, it *co-evolves the representation with the program*, so the search can discover, per task, the representation under which a short program exists. The cost of that flexibility is a larger search space, since the individual now spans parser, program, and projector rather than program alone, and much of the design in the next chapter is about keeping that enlarged space tractable.

# Chapter 4

# Representation and Language

The representation and the language are the subject here, starting from the end-to-end architecture and then taking the four moving parts in turn: the parser that turns a grid into a typed multigraph, the intermediate representation itself, the domain-specific language that rewrites one graph into another, and the projector that renders the result. Chapter 5 takes up the search over them, together with the fitness that steers it and the machinery used to audit the parser in isolation.

## 4.1 Overview

A solver for ARC-AGI needs three capabilities: it must *parse* a scene into symbols, *operate* on those symbols to produce a result, and *project* the result back into a grid. It also needs a way to *assess* how close a candidate is to correct, so that a search can prefer better candidates. In this system all four are made explicit and, for the first three, evolved. Nothing in a task prescribes any of it: a task supplies coloured grids, and the decision to see them as objects with typed relations between them is this work's own commitment, which is why the representation is put under search rather than fixed by hand.

Figure 4.1 shows the pipeline. An individual is the triple $I = (\mathcal{I}, \mathcal{P}, \mathcal{O})$: a parser configuration $\mathcal{I}$, a program $\mathcal{P}$, and a projector configuration $\mathcal{O}$. The parser maps the input grid to a graph $G_{\text{in}}$; the program rewrites it into $G_{\text{out}}$; the projector renders $G_{\text{out}}$ into a predicted grid. Every operator in the system acts on the typed graph, never on raw pixels. This constraint is deliberate: it concentrates the modelling in the representation and keeps the program short.

The division of labour is uneven on purpose. The parser spends most of the complexity budget building a structured intermediate representation; the program

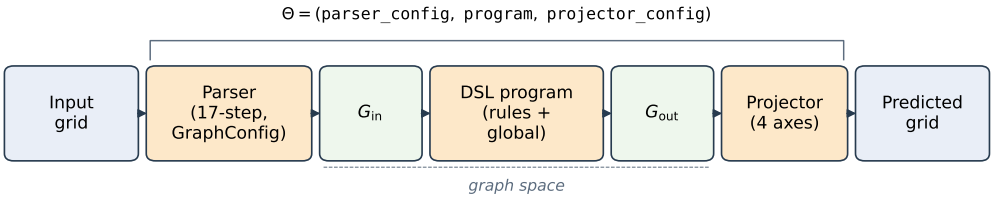



**Figure 4.1:** End-to-end view. The parser turns an input grid into a typed multigraph $G_{\text{in}}$; a DSL program rewrites it into $G_{\text{out}}$; the projector rasterizes the result. The three components are evolved together, per task.

is narrow and typed; the projector is almost trivial. That partition is what makes a per-task search affordable on a single workstation: the hard modelling choices are exposed as a modest number of parser parameters, and the program search then operates over a small, typed space rather than over arbitrary pixel manipulations.

**Why co-evolve, and why it is tractable.** The distinctive choice is to evolve the parser *together* with the program rather than fixing one and searching the other. The motivation is that the two are entangled: a program is short only under the right representation, and the right representation is the one that makes some short program correct, so choosing either in isolation is choosing blind. Co-evolution lets the search discover the pair jointly. The obvious objection is cost, since the joint space is the product of three spaces rather than one, and the answer is the uneven partition just described. The parser space, though the configuration has forty-four fields, is one the search explores gently: only twenty-one fields are put under mutation in the shipped setup, a mutation resamples one of them at a time (occasionally two), and a good representation, once found, is perturbed rarely (the mutation weights of Section 5.1 spend only a twentieth of their mass on parser edits). Most of the search effort goes into the program, whose typed structure keeps its branching factor bounded, and the projector's axes are few enough to sweep exhaustively when it matters. The product space is large in principle, but the search does not treat its three factors symmetrically: it treats the parser as a slowly-moving context within which a fast program search runs, which is what keeps the joint optimization inside a per-task budget. Chapter 6 returns to the empirical evidence that this partition is the right one, namely the finding that the program, not the parser, is where the remaining gains lie.

## 4.2 The parser: from grid to typed multigraph

The parser is a single function that runs a fixed sequence of seventeen detection passes, most of them gated by a flag in a configuration object; background detection and connected-component extraction always run, steered by a categorical parameter rather than switched on and off. The passes are ordered so that later steps can read the results of earlier ones. Background detection comes first, then grid-line "zones," then connected-component extraction of objects, object decomposition and clustering, cavity detection, several families of edge generation, and finally pattern and symmetry detection. Because most passes are gated, the same grid yields different graphs under different configurations, and this is what the evolutionary search exploits.

**Node and edge vocabulary.** The pipeline emits seven node types (`SubObject`, `ClusterObject`, `CavityObject`, `ZoneNode`, `PatternObject`, `ImplicitObject`, and `NoiseObject`) and edges drawn from twenty-two types, organized into five families: hierarchy (containment, part-of, inside-zone), spatial (adjacency, overlap, and the four directional relations), alignment (shared row or column, diagonal, ray projection, shared centroid axis), physics (falls-towards, bridged-to), and logic (same shape, same colour, collinear, symmetric, repeating pattern). A full catalogue is given in Appendix A, in Tables A.1, A.2 and A.3, with the arguments each operator takes in Tables D.2 and D.3.

**The configuration.** The configuration object exposes forty-four fields (object-extraction mode, pixel connectivity, background policy, noise thresholds, and a set of detection toggles), each annotated with metadata declaring its type, range, and legal values; Tables B.1, B.2 and B.3 list them by group. Twenty-one of them are put under mutation in the shipped configuration: the extraction and detection parameters, together with their numeric thresholds. The twenty-three left at their defaults are the edge-computation and node-feature toggles, the five noise-scoring weights, the alignment and symmetry tolerances, the minimum pattern size, and one extraction flag. That halves the searched dimensionality and is a tuning decision rather than a limit of the design. Bringing a new parser behaviour under evolutionary control therefore takes two steps: annotate the field, and add it to the mutable set. Figure 4.2 shows the consequence: the same small input gives rise to quite different graphs under different configurations, and the search discovers, per task, which representation makes the target reachable.

**Core-knowledge grounding.** The five edge families are more than an organizational convenience. They correspond loosely to the *objects*, *places*, *forms*, and naive-physics systems of Spelke's taxonomy (Section 3.1). This has a useful

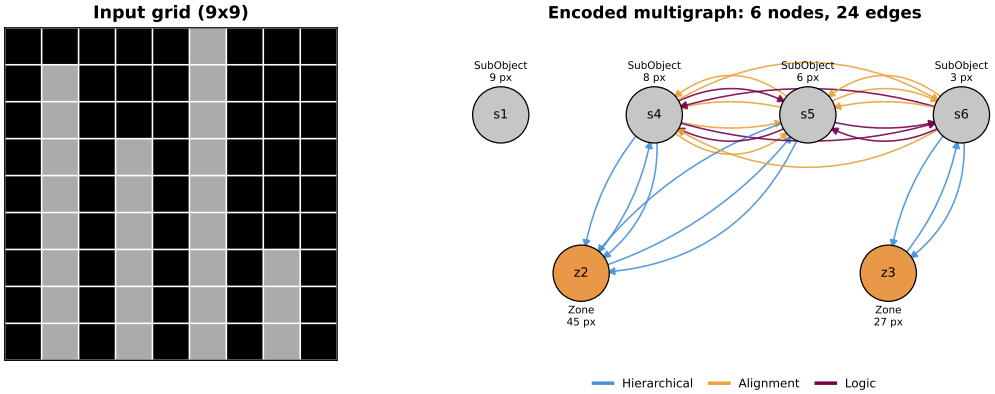



**Figure 4.2:** Task `08ed6ac7`, first training input, under the default configuration. The parser emits four `SubObject`s and two zones, joined by 24 edges of six types: `IS_PART_OF`, `CONTAINS` and `IS_INSIDE_ZONE` in the hierarchy family, `SHARES_ROW` and `PROJECTS_RAY_TO` in alignment, and `SAME_COLOR` in logic. Edges are coloured by family rather than labelled individually. Several pairs carry more than one relation at once, which is what makes the representation a *multi*graph and lets a rule select on any one of them.

failure mode. When a task needs reasoning outside this subset, such as large-scale counting or modelling an agent's goals, the parser has nothing to offer and the search degrades to pixel-level mismatch. That is a clean, visible failure rather than a silent wrong answer, and it is one of the signals used in Chapter 6 to locate the bottleneck.

**The seventeen passes.** The pipeline is best read in four stages, each building on the last; Appendix A.3 lists the passes in order. The first stage *finds the objects*. Background detection (pass 1) decides which colour, if any, is the canvas, so that later passes do not treat the background as an object. Zone detection (pass 2) finds grid lines and the regions they delimit. Connected-component extraction (pass 3) is the heart of the stage: it groups pixels into `SubObject`s under the chosen colour mode and connectivity, after which object decomposition, multi-colour clustering, and merge-by-containment (passes 3b–3d) refine that grouping, either splitting a composite shape into simpler pieces or merging fragments that belong together. Cavity detection (pass 4) finds enclosed empty regions, and cluster detection with split-by-containment (passes 5, 5d) forms and separates higher-level groupings. By the end of the first stage the graph has its nodes.

The second stage *attaches spatial structure*. Spatial-edge computation (pass 6) adds adjacency, overlap, and the four directional relations; noise scoring (pass 7) assigns each node a salience score from five weighted signals; containment edges

(pass 8) record which objects sit inside which; and local coordinates with inside-zone edges (pass 9) place each object in its zone's frame. Zone features and $z$-index (passes 9b, 10) finish the node attributes, layering objects by containment depth. The third stage *adds the relational edges* that a pixel array cannot express: alignment (pass 11, shared rows and columns, diagonals, collinearity), rays (pass 12, directional line-of-sight), gravity (pass 13, which object would fall onto which), and bridges (pass 14, objects joinable across a gap). The fourth stage *adds semantics*: logic edges (pass 15) link objects of the same shape or colour, pattern detection (pass 16) creates `PatternObject`s for repeating structure, and symmetry detection (pass 17) records reflection and rotation relations.

The ordering is a hard constraint, not a convenience: pass 16 creates `PatternObject`s, so any pass that reasoned about patterns before step 16 would see none, and the test suite pins several such ordering dependencies by snapshotting the edge sets a pass produces. Because the detection passes are individually gated, the search can, for a given task, run only the ones that help: zone, cavity, cluster, pattern and symmetry detection are all under its control, so a counting task may switch off symmetry detection entirely while the edge families stay fixed. It is this per-task gating, rather than any single clever detector, that lets one parser serve tasks as different as gravity simulation and symmetry completion.

**A worked parsing.** The graph of Figure 4.2 is a good illustration of what the passes produce. The first training input of task `08ed6ac7` is a $9 \times 9$ grid holding four line-shaped objects and two grid-line zones. Under the default configuration the parser emits six nodes, four `SubObject`s and two `ZoneNode`s, joined by twenty-four edges, and the edge census is worth reading because it shows the families at work on a single small grid. Three `CONTAINS` edges and their three `IS_PART_OF` inverses record which zone each object sits in at the hierarchy level; three `IS_INSIDE_ZONE` edges tie the objects to their zone frames; six `SHARES_ROW` edges and three `PROJECTS_RAY_TO` edges capture the alignment structure that this task turns on; and six `SAME_COLOR` edges record the logic-level fact that several of the objects share a colour. No spatial-adjacency, physics, or symmetry edges appear, because the objects neither touch nor stand in any gravitational or symmetric relation. The parser emits only the relations the grid supports, with no filler. A selector in the program can now ask directly for "the objects that share a row" or "the object a ray points at," relations that a $9 \times 9$ array of colour indices does not expose at all.

The current parser can tune its parameters but cannot yet *compose* detection passes into new ones; adding compositionality to the parser is noted as future work in Chapter 7.

## 4.3 The intermediate representation

Between the parser and the program sits the object that both operate on: a typed directed multigraph. It is worth describing on its own, because it is where the design places its bet. The claim of the thesis is not that graphs are a convenient container but that the *right* intermediate representation makes the program short, and the representation is therefore engineered to carry exactly the information a short rewrite program needs.

**What a node carries.** Every node is one of the seven types (Appendix A.1), but the type is only a label; the content is in the attributes. A `SubObject`, the workhorse type, one per detected connected component, carries its pixel list, its colour, its mass (pixel count), its bounding box in $(y, x)$ coordinates, a shape classification and a shape hash for matching, a centroid, and a noise score. These attributes are what the selectors read: a mass selector reads the mass, a shape selector reads the hash, an extremal selector reads the bounding box. The other six node types specialize this: a `ZoneNode` carries the colour of the region it delimits, a `ClusterObject` a colour histogram over its members, a `CavityObject` the object that encloses it, and a `PatternObject` the kind of repeating structure detected and the nodes that constitute it. A node flagged `is_structural`, a grid line for instance, is excluded from the semantic edge passes, so that the scaffolding of the grid does not masquerade as an object to be transformed.

**What an edge carries.** An edge is a typed, directed relation between two nodes, drawn from the twenty-two types in five families (Appendix A.2). Edges are the part of the representation that a pixel array cannot express, and they are what let a selector reason relationally: "the object that this arrow points at," "the objects that share a row," "the object bridged to that one." The families are not arbitrary. Hierarchy edges encode containment and part-of relations, spatial edges encode adjacency and direction, alignment edges encode the shared-axis relations that so many ARC tasks turn on, physics edges encode a coarse gravity and bridging, and logic edges encode same-shape, same-colour, and symmetry relations. Because the graph is a *multi*graph, two nodes can be joined by several edges of different types at once, so that two objects can be both adjacent and same-coloured, and a selector can ask for either relation independently.

**Why typed, why directed.** The typing is what makes the program search tractable rather than a nicety of implementation. A selector that asks for "nodes with an incoming ray edge" is meaningful only if ray edges exist and are directed; the direction distinguishes the object that casts a ray from the one that receives it, and the type distinguishes a ray from mere adjacency. Because a selector whose

relation the parser did not build matches nothing, the operators that can act on a task reduce to the relations its graph actually contains (Section 4.4), which is the mechanism by which a richer parser configuration can either open up or close off whole regions of the program space. The intermediate representation, in other words, is not a passive data structure between two active components; it is the surface on which the parser's choices become the program's constraints.

## 4.4 The domain-specific language

A `Program` is a finite list of `Rule`s followed by an optional `GlobalTransform`. Each rule has a `Selector`, zero or more `Guard`s, and one `Action`. The executor is deterministic: the same `(scene, program)` pair always yields the same $G_{\text{out}}$. Determinism is not a nicety: it is what makes the program-hash deduplication inside a generation and the fitness cache across generations correct.

**Table 4.1:** The domain-specific language. The operator set is expected to grow; the counts here are those of the version evaluated in Chapter 6.

| Kind | Count | Representative examples |
|---|---|---|
| Selectors | 34 | `MOST_FREQUENT_COLOR_OBJECTS`, `LARGEST_SUBOBJECT`, `HAS_EDGE_TYPE`, `SAME_SHAPE_AS_REFERENCE`, `NEAREST_TO_SELECTOR` |
| Actions | 44 | `RECOLOR_SEQUENCE`, `FILL_CAVITY`, `STAMP_LOCAL_PATTERN`, `COMPLETE_D4_SYMMETRY`, `PROJECT_MARKER` |
| Globals | 26 | `ROT90`, `KRON_SELF_MASK`, `CROP_FOREGROUND`, `FILL_PERIODIC_BACKGROUND` |
| Guards | 3 | `MIN_COUNT`, `MAX_COUNT`, `EXACT_COUNT` |

Table 4.1 summarizes the language. Selectors split by the node property they filter on: type, colour, mass, shape, extremal position, or a relational criterion such as the presence of a given edge type. Actions split by what they do to the selected set: the recolour, delete, and translate families are self-explanatory; a morphology family covers endpoint pruning and dilation; a geometry family covers bounding-box and cavity fills; and a thin higher-level layer packages recurring patterns such as D4-symmetry completion and full-line projection. The optional global-transform slot handles rotations, flips, self-Kronecker products, and a few arithmetic tilings that do not factor through selector–action rewriting.

Because the DSL is graph-typed, a selector that finds nothing in the current graph costs nothing: it returns an empty set and the executor skips the rule. A selector that reads a structure the parser did not build is inert in a task where cavity detection is switched off. The effective language on a given task is therefore not the full $34 \times 44$ matrix of selector–action pairs but the subset whose selector types are populated in that task's graph, which is another way the parser configuration shapes which programs can act, though the sampler draws throughout from the same fixed inventory.

## 4.5 The executor

The executor is the function that gives a program its meaning: it takes an input grid, a parser configuration, a program, and a projector configuration, and returns a predicted grid. Its full path is grid $\rightarrow$ parse $\rightarrow G_{\text{in}} \rightarrow$ rules $\rightarrow G_{\text{out}} \rightarrow$ project $\rightarrow$ grid, and it is deterministic end to end: the same inputs always produce the same output, the same execution trace, and the same final scene. That determinism is not decorative. It is what makes the program-hash deduplication within a generation sound (two genomes with the same program hash provably compute the same thing) and the fitness cache across generations correct (a cached score can be trusted). A single hidden source of nondeterminism would quietly corrupt both.

**Rule application.** The rules are applied in list order. For each rule the executor evaluates the selector against the current scene to get a set of node identifiers, checks the rule's guards against the size of that set, and, only if every guard passes and the set is non-empty, applies the action, mutating the scene in place. A rule whose guard fails, or whose selector matches nothing, is skipped silently; it is a no-op, not an error. This "skip on empty" semantics is by design: it lets a program carry a rule that fires only on the tasks where its selector matches, so a single evolved program can generalize across a task's train pairs even when they differ in object count.

**The safe-selector fast path.** Naively, every rule would require re-parsing the grid after the previous rule mutated it, because a selector reads graph structure that the mutation may have invalidated. Re-parsing is the dominant cost in the loop, so the executor avoids it whenever it safely can. A frozen set of eighteen selectors is marked *safe*: they read only node attributes that actions maintain in place, such as colour, mass, bounding box, and centroid, and never touch connectivity, edges, or shape hashes. When the next rule's selector is safe, the executor keeps mutating the same in-memory scene and skips the rebuild; only when the next selector needs fresh structure (an edge-type selector, say) does it render the scene to a grid and

re-parse. Because typed mutation frequently produces runs of safe selectors, this fast path removes a large fraction of the re-parses a literal reading of the semantics would demand, and it is a substantial part of why a per-task search fits inside the wall-clock budget.

**Global transforms.** The optional global transform is applied last, and how it is applied depends on its kind. Whole-grid transforms (mirror-concatenations, tilings, run deduplication, block reduction) act on the *rendered grid* as array operations, so that they move the background along with the foreground. Geometric permutations (rotations, flips, transpose) behave differently when a program has rules: they are applied in scene space, to the foreground objects, and only the empty-rule fast path rotates the whole canvas. When a program has no rules, the executor takes a fast path that applies the transform straight to the input, which is how the common "the whole task is a single rotation" family of tasks is solved cheaply.

## 4.6 The projector

The projector inverts the parser, up to a point. A single function walks the pixel lists of the `SubObject`s back onto a blank canvas under four policy axes: which background to use, out of five policies, in what order to paint objects, how to resolve overlaps, and whether to crop the foreground as a post-process. The four axes and their legal values are listed in Table B.4. Four axes suffice because most of the semantic work has already happened in graph space; the projector only has to make consistent rasterization choices. Three of the four are part of the evolved individual, paint order being frozen at its default.

The four axes are small but not negligible, because the same graph can project to different grids depending on how they are set, and the search needs that freedom. The *background* axis decides what fills the cells no object claims: the scene's detected background, a hard zero, the most frequent colour of the input, the most frequent colour of the prediction, or a copy of the input grid. A task that adds objects to an unchanged scene needs the input-copy policy, while one that rebuilds the grid from scratch needs a flat background. The *paint order* axis decides the sequence in which objects are rasterized, which matters only when they overlap: painting largest-first versus smallest-first changes which object's colour survives at a shared cell. The *overlap* axis makes that choice explicit, as either last writer wins or first writer wins, and the *post-process* axis optionally crops the result to its foreground bounding box, which is how a task that extracts a small region from a large input gets its output size right. The projector is the least expressive of the three components by design: giving it more power would let it compensate for a wrong graph, which would blur the signal the fitness function needs to locate errors

in the parser or the program rather than in the rasterizer. Keeping it thin is what makes the bottleneck analysis of Chapter 6 possible at all.

# Chapter 5

# The Evolutionary Search

Chapter 4 fixed what a candidate solution *is*: a parser configuration, a program of typed rewrite rules, and a projector policy, carried together as one individual. Finding one is the subject here: the evolutionary loop that searches the space of individuals under a per-task deadline, the ranking that steers it, the implementation decisions that make the search measurable, and a library-learning extension that tries, so far without success, to give the search a better proposal distribution.

## 5.1 The evolutionary optimizer

The individual $I = (\mathcal{I}, \mathcal{P}, \mathcal{O})$ is optimized, per task, by an evolutionary loop; Table E.1 collects this notation and the symbols used with it. In the current prototype the optimization runs in three phases (Figure 5.1); merging them into a single evolutionary loop is planned future work.

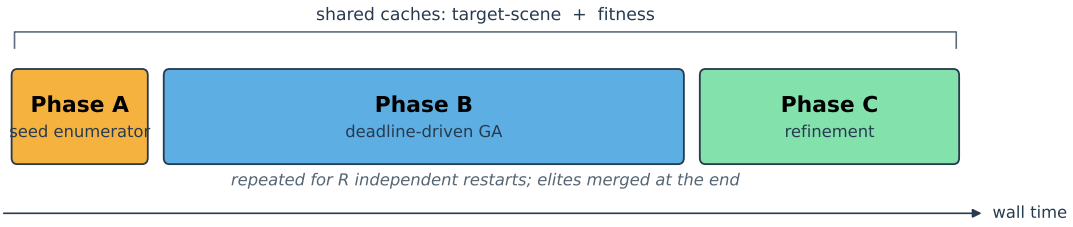



**Figure 5.1:** Per-task search schedule. Phase A supplies seeds, Phase B runs a deadline-driven GP on the combined pool, Phase C refines the best candidates. Two caches span the phases: a target-scene cache keyed by the parser signature, and a fitness cache keyed by the parser signature, the program hash, and the projector fields.

**Phase A: seeding.** A focused seed enumerator, with a budget of a few seconds, covers obvious hypotheses before the GP starts: single global transforms, one-rule programs for common patterns, six hand-written parser configurations spanning sparse to full graphs, and, when the library-learning extension of Section 5.5 is enabled, a set of macro realizations.

**Phase B: deadline-driven GP.** The main loop's binding constraint is a per-task wall-clock budget, not a fixed generation count. Evolution proceeds with typed, per-slot mutation operators, program-hash deduplication, elitism on the top-$K$, and an escalating response to stagnation, and it stops when the deadline fires. Parent selection defaults to $\varepsilon$-lexicase [47, 48, 49]: instead of averaging performance across the train pairs, lexicase considers the cases one at a time in a random order, keeping at each step only the candidates within an $\varepsilon$ band of the best on that case, until one candidate remains or the cases run out. At the default granularity a case is not a whole pair but a sub-objective within one: the object-distance components, the whole-grid pixel error, one case per target colour, and four quadrant cases. The effect is to preserve *specialists*. A program that nails one hard pair but is mediocre on the others survives, because some shuffles put its strong pair first, whereas a mean-based selector would average that program away. This matters on ARC because different train pairs of one task often stress different parts of a program, and a candidate that has solved the hard part is worth keeping even if it has not yet solved the easy one. When the case filter leaves several candidates tied, one of the survivors is taken at random; a tournament over the aggregate rank tuple is the fallback only when the configuration disables lexicase altogether. The stagnation response is layered: a short flat streak injects heavily mutated copies of the elites, and a longer one replaces half the population with fresh random genomes, so the search escapes a converged population without discarding the elite it has found.

**The generation loop.** Within Phase B each generation carries the top $K = \max(2, \lfloor \text{pop}/10 \rfloor)$ elites forward unchanged, then fills the rest of the population by selecting parents, recombining and mutating them, and rejecting any child whose program hash already appears in the generation, a within-generation deduplication that keeps the population from collapsing onto a few genomes. Ninety percent of children are mutated. When the best rank has not improved for a while, two escape mechanisms fire in sequence: after two flat generations the loop injects a batch of double-mutated elites (a local kick), and after four it replaces half the population with fresh random genomes (a mini-restart), resetting the counter. The two thresholds are kept small because the wall-clock budget is the real limit: waiting many generations to detect stagnation would waste budget that is better spent exploring.

**Phase C: refinement.** After the GP deadline, a refinement chain polishes the best candidates in a fixed order. A structural hill-climb applies greedy typed mutations and keeps any that improve fitness. An argument-jitter pass systematically perturbs the numeric arguments of the top few candidates, bounded to a fraction of the remaining budget. For near-miss genomes, those whose worst pair is within a small mismatch of exact, a delta-guided step reads which pixels are wrong and biases mutation toward the colours involved, either swapping an action's colour or appending a targeted recolour rule. A program compressor then drops any rule whose removal does not worsen fitness, which is what keeps solved programs short (Chapter 6). Finally, once all restarts have finished, a cross-restart merge pools their elites, scores each under a leave-one-pair-out generalization penalty, and keeps a small finalist set ranked by that penalty and then by complexity.

**Genetic operators.** The operators are typed to the component they act on, and the mutation operator picks among them by the weights in Table 5.1. The weighting reflects where edits pay off: most of the probability mass edits an existing rule or changes the number of rules, because the program is where a task is usually won or lost, while parser and projector edits are rarer because a good representation, once found, should be perturbed gently. Crossover recombines two parents component by component. Two random cut points splice the rule lists, the parser fields are inherited field by field, and the global transform and the projector configuration are each taken whole from one parent or the other. Because splicing rule lists readily produces programs that no longer type-check, crossover is used alongside, not instead of, mutation, and the invalid offspring it produces are filtered by the same execution guard that scores every candidate.

**Caching.** Two memoization layers make the loop tractable. A target-scene cache, keyed by the parser signature, stores the parsed train-pair outputs so that a program-only mutation does not trigger re-parsing. A fitness cache stores the score of every candidate seen during the run, keyed by the parser signature, the program hash, and the projector fields, the same triple that the determinism invariant guarantees uniquely identifies a candidate's behaviour. Typed mutations collide with existing cache entries often enough that the solver can examine a large candidate pool inside the wall-clock budget. Below these two caches sits a third, a per-pair distance cache keyed by the predicted grid's bytes, so that two different programs that happen to predict the same grid on a pair share the distance computation.

**Table 5.1:** Mutation-operator weights. One operator is chosen per mutation event; the add-rule operator may instead realize a macro when the library-learning extension of Section 5.5 is enabled.

| Operator | Weight | Acts on |
|---|---|---|
| mutate an existing rule | 55% | a random rule's selector, action, or arguments |
| add a rule (or realize a macro) | 15% | appends a rule if under the rule cap |
| remove a rule | 10% | deletes a random rule |
| reorder two rules | 5% | swaps two rule positions |
| swap the global transform | 5% | the optional whole-grid slot |
| mutate the parser | 5% | one evolvable `GraphConfig` field |
| mutate the projector | 5% | one of the three mutable projector policy axes |

## 5.2 The fitness function

Even with a well-chosen DSL, the space of programs is, in Dennett's term, *Vast* [55]: far too large to enumerate. An effective evolutionary search therefore needs a fitness function that can discriminate between candidates and point the way toward better ones. This is hard on ARC, for a specific reason.

**The discrimination problem.** With only input–output examples to go on, a naive correct/incorrect signal per example is nearly useless: two utterly different programs that both fail on every pair are scored identically, and the search gets no gradient. A better signal has to look past raw pixels. A candidate that reproduces every object in the right position and shape but paints the background the wrong colour is, in any reasonable sense, close, even though pixel-wise it may differ everywhere. The right notion of "close" is therefore tied to the representation: differences should be measured over the high-level symbols the parser produces, not over cells.

**Lexicographic ranking.** The solver addresses this with a deliberately non-smooth ranking. Candidates are compared lexicographically on: (i) the strict predicate "all train pairs reproduced exactly"; then (ii) the fraction of train pairs reproduced exactly; then (iii) the worst and (iv) the average normalized mismatch across train pairs; and finally (v) complexity, the rule count, used only to break ties among otherwise-equal candidates. The key empirical finding is that the "all

pairs exact" frontier is a discontinuity the search has to *land on*, and that any attempt to smooth it into a continuous surrogate flattens the landscape and hurts the search. Chapter 6 reports the experiments behind this claim.

**The full rank tuple.** The ranking above is the version that steers the hot loop; the complete tuple has two extra terms that come into play once a candidate is solved, and the ordering encodes a small theory of what a good solution is. Reading a lower value as better, a candidate is ranked by, in order: *solved* (not-solved sorts after solved); a *leave-one-pair-out* generalization penalty; a *specificity* count; *complexity* among solved candidates; the negated exact ratio; the worst-pair mismatch; the mean mismatch; and *complexity* again as a final tie-break, this time unconditionally. The exact-match predicate is gated on true array equality, so a program that reproduces the right *graph* but renders a different grid is never mistaken for a solution: structural coincidence cannot sneak past the frontier. The generalization and specificity terms only separate candidates that have already cleared the frontier: among programs that fit every train pair, the tuple prefers the one that leans least on pair-specific constants (specificity) and that would still fit if any single train pair were withheld (leave-one-out). This is the fitness function's one concession to the fact that fitting the train pairs is necessary but not sufficient, since the real target is the held-out test pair. The leave-one-out penalty is computed only in the cross-restart finalist gather, never in the per-generation hot loop, because doing it for every candidate would be far too expensive; the specificity count is cheap enough to compute on every evaluation, and it is switched off in the shipped configuration, so in the runs reported here it separates nothing.

**A guard against blow-ups.** One practical term sits outside the ranking. Some global transforms multiply the grid size (a 3× nearest-neighbour scale-up produces nine times the cells, a self-Kronecker product squares them), and a program carrying one of these on a large input can ask for an output of millions of cells, which would stall the loop rendering a candidate that is certainly wrong. The fitness therefore refuses to render any candidate whose projected output exceeds a hard cap of a few thousand cells, scoring it as maximally bad without executing it. The cap is checked from the transform and the input shape alone, before any rendering, so the guard is cheap and the blow-up is caught before it costs anything.

**Graph-based distance.** Below the exact-match frontier, the graded comparison is computed in graph space rather than pixel space, over an object-graph distance with five components: node count, node matching, colour, shape, and bounding box. It is deliberately cheaper than the ten-component audit distance of Table B.5 and Section 5.4, which adds node types, extra attributes, and edge structure and is used to measure the parser, not to steer the search. This is the concrete realization

of "measure differences over symbols." An even more ambitious variant, following a suggestion of Chollet's, would project the output graph to pixels and re-parse it with the original parser, comparing $\mathcal{I}(O_t)$ against $\mathcal{I}(\mathcal{O}(\bar{O}))$ to score representational rather than pixel agreement [56, 53]; the trade-offs of that round-trip formulation are discussed with the experiments.

**A worked evaluation.** A small example makes the ranking concrete. Consider a task whose target output, on one train pair, is a red $3 \times 3$ square sitting on a black background, and take three candidate programs. Candidate $A$ produces nothing but background, an empty grid. Candidate $B$ produces a *blue* $3 \times 3$ square in the right place. Candidate $C$ produces a red square of the right colour but one cell too wide, a $3 \times 4$ block. Pixel-wise, $C$ differs from the target on only three cells (the extra column) while $B$ differs on nine (every square cell has the wrong colour) and so does the empty $A$, which simply misses those same nine. A naive pixel count therefore puts $C$ first and cannot separate $B$ from $A$ at all. The lexicographic fitness tells a more useful story. None of the three clears the exact-match frontier, so tier (i) does not separate them and tier (ii), the fraction of pairs reproduced exactly, is zero for all. The separation happens in graph space. The comparison abstracts each grid into single-colour regions of its own, without running the parser: the target becomes one region, a red square of mass nine with a bounding box and a normalized pixel set. Candidate $A$ has no foreground region at all, so it matches nothing and scores worst on every component. Candidate $B$ has one region matching the target on position, mass, and shape but not colour, so it loses only the colour component. Candidate $C$ has one region matching on colour and position but differing in mass, shape and bounding box, so it loses three. The graph-distance therefore ranks $B$ ahead of $C$ ahead of $A$, inverting the pixel count's preference between $B$ and $C$ and separating the $A$ it could not, which is the order a human would call "closest." A search steered by this signal will prefer $B$, which needs only a recolour to become exact, over $C$, which needs a structural edit, even though $C$ looks closer at the pixel level. This is the concrete payoff of measuring differences over symbols rather than cells, and it is why the graph-distance tie-break survived where the alternatives tried against it did not (Section 6.11).

## 5.3 Implementation, determinism, and reproducibility

This work implements the system twice from one design. The Python one, described throughout this thesis, is built over NumPy and NetworkX. Its per-task search runs in a single process; the parallelism is across tasks, one worker per core, which is why

a whole 400-task run fits in hours rather than days. A later C++ reimplementation trades that readability for speed and produces the strongest single run on record (Section 6.2). Two invariants are treated as correctness properties. First, *separate scenes*: the input and output grids of a pair always produce independent graph instances, so that the "after" side can never read intermediate state from the "before" side and turn the fitness into a function of parser side-effects. Second, *determinism*: for a fixed seed and configuration, the population, the program hashes, and the projected outputs must be identical run to run. This second invariant is subtle in Python, where unpinned hash randomization silently perturbs set and dictionary iteration order and, through it, the sequence of mutation draws. That effect masqueraded as method-level improvement or regression during development until the hash seed was pinned. Chapter 6 returns to this point, because it is the difference between a measurable experiment and a misleading one.

## 5.4 Auditing the parser against ground truth

The co-evolutionary loop optimizes the whole individual against raw grid pairs, which answers "does the system solve the task" but not "is the parser any good." Those are different questions, and conflating them is how a project convinces itself the representation is fine when it is not. To keep them separate the project carries a second, standalone evaluation machinery whose only job is to measure how faithfully the parser reconstructs a hand-labelled reference graph. It is not part of the solving loop; it is the instrument that lets Chapter 6 claim, with evidence, that the bottleneck is the DSL rather than the parser.

**Ground truth.** A `GroundTruth` is a hand-annotated graph for one grid: a task identifier, a grid key, a list of nodes, and a list of edges. Each ground-truth node carries the same attributes the parser would produce (a type, colour, pixel list, shape, bounding box, mass, and a bag of extra attributes), and each edge carries a source, a target, and a type. Ground truth is produced by exporting a live scene to the annotation format, hand-correcting it, and saving it as JSON, so that the reference is expressed in the same vocabulary the parser emits and the two can be compared directly.

**A distance over graphs.** Comparing two graphs needs a distance, and a pixel distance will not do, since two graphs can describe the same scene with differently numbered nodes. The evaluation uses a weighted *graph distance* with ten components: node count, node-type distribution, node matching, colour, shape, bounding box, extra attributes, edge count, edge-type distribution, and edge matching. The heart of it is the node-matching component, which pairs predicted

nodes to ground-truth nodes by greedy best-overlap matching, ranking candidate pairs by pixel intersection-over-union, lightly weighted by type agreement, and assigning the best unused pairs in turn, then penalizing both imperfect overlaps and unmatched nodes. The remaining components read off colour, shape, and bounding-box agreement over the matched pairs, and the edge components compare the two edge sets under the node mapping the matching produced. The default weights are not uniform: node matching carries the highest weight and edge matching the next, because they are the most informative about whether the right objects and relations were found, while bounding-box and extra-attribute agreement are down-weighted as secondary (Appendix B.3). Four presets specialize the distance (*full*, *nodes-only*, *edges-only*, and *properties-only*). *Nodes-only* and *edges-only* zero the weights outside their focus, so that node extraction can be tuned without edge noise and vice versa; *properties-only* zeroes the edge terms too but keeps node count and node type at half weight while boosting the colour, shape, bounding-box and extra-attribute terms; *full* leaves every default in place.

**The audit evolver.** On top of this distance sits a second evolver, distinct from the co-evolutionary loop, whose search space is a single `GraphConfig` and whose objective is to minimize the graph distance to a set of reference graphs. It runs in three staged phases that vary different parameter groups in turn: a fifteen-generation *structure* phase that searches the extraction and detection parameters at a mutation rate of 0.25; a thirty-generation *refinement* phase that unfreezes all parameters and fine-tunes at rate 0.15; and a fifteen-generation *edge* phase that fixes the node parameters and searches the edge and tolerance parameters at rate 0.3. It uses tournament selection with two elites, a population of thirty, three independent restarts sharing one fitness cache, and parallel evaluation across worker processes. When the distance preset zeroes the edge weights, as in a nodes-only audit, the third phase is skipped and the edge parameters are frozen throughout, because there is then nothing for them to improve. A companion viewer tool renders a reference graph beside the graph the default configuration produces and prints their distance, which is the manual counterpart to the automated audit.

**Why it matters.** This machinery is why the parser can be cleared rather than merely assumed innocent. If the audit evolver, given a task's reference graphs, can drive the graph distance to near zero with some configuration, then the parser is *capable* of the right representation for that task and any failure to solve it lies downstream. Chapter 6 uses this logic: on the near-miss tasks it finds the parser's output already clean, which is what licenses the conclusion that the missing piece is an operator in the DSL, not a detector in the parser.

## 5.5 Library learning: mining reusable macros

The search described so far starts each task from a nearly blank slate, which is wasteful: many ARC tasks share a strategy even when they share no surface detail. This section describes an extension that mines reusable strategies from previously solved programs and feeds them back as a sampling prior, following DreamCoder's wake–sleep template (Section 3.3). It is reported here as a mechanism and a negative result, not as a contribution to the solved count.

**What is stored.** A *macro* is a parametric multi-rule strategy with three parts. Table D.1 gives example programs in this form. A *role skeleton* is a sequence of (selector-role, action-role) pairs, where roles abstract the concrete operator kinds into eight functional categories on each side, so that recurrence is counted over an $8 \times 8$ space rather than the $34 \times 44$ space of concrete pairs. Per-slot *head distributions* record which concrete operators realized each slot, so realization samples a plausible operator rather than a fixed one. Typed *bindings* record equalities across holes, most usefully "the same colour reused in both rules," each carried with the fraction of instances in which it was observed and applied softly at that rate. The library stores roles and argument holes, never a constant colour or size: every scalar is redrawn through the codec's own samplers at realization time, which is what separates a strategy from a memorized answer.

**How it is mined.** The miner abstracts every solved program to its role sequence, slides a window of length two and three over the rules, and keeps a skeleton only if it recurs across at least three *distinct tasks*. That distinct-task threshold is the anti-memorization gate, and it is the direct analogue of Stitch's compression-times-usage objective: a skeleton appearing many times within one task is a canned solver rather than a strategy. A cross-slot equality holding in at least sixty percent of instances becomes a binding. The miner also records which slots are dominated by a single concrete head, at an eighty percent threshold, but the loader does not read that field back: realization always samples from the full head distribution, so the pin is recorded and unused at the version evaluated here.

**Whether the strategies transfer.** A prior can only lift a task whose strategy it has seen elsewhere, so transfer was measured before lift. Mining on one parity half of the solved tasks and testing on the other, roughly a third of held-out tasks have a multi-rule solved program containing a mined skeleton as a consecutive subsequence. The strategies do recur across unrelated tasks, which is the precondition the whole idea rests on, and it sets the ceiling: a prior cannot lift more than the tasks whose strategies it has already seen.

**The prior does not yet pay for itself.** Measured against the signal-above-noise bar of Section 6.10, the result is a null. On the ARC-AGI-2 training split the prior is worth two to four tasks in distribution, which is inside the drift band and therefore not a result. On held-out data it is flat, moving the count from 0 to 1 of 120 against its own baseline; the 2 of 120 reported in Chapter 6 comes from a different build and is not the comparison. And realizing macros during mutation perturbs the search trajectory enough to churn some thirteen tasks per side, gained and lost, which is more than the signal it delivers. The feature is implemented, off by default, and carries no demonstrated held-out gain.

The null is worth stating precisely, because the premise survives it. The strategies genuinely recur, so the failure sits in the delivery rather than in the transfer. Either the realization rate is high enough that trajectory churn swamps the benefit, which the thirteen-per-side figure directly supports, or the coupled realizations the prior favours are not the ones the exact-match frontier rewards on these tasks. The seed-only variant has since been run and points at the second: injecting the top-eight macros through the seed lane alone still came in at $-13$, so the churn is not an artefact of the realization rate alone (Section 6.14).

# Chapter 6

# Experimental Evaluation

What follows is the empirical evaluation of the solver on ARC-AGI-1. The figures reflect the state of the prototype at the end of July 2026 and are expected to improve as the domain-specific language and the seed enumerator grow. The evaluation has four aims: to establish how the approach scales with compute, to locate the current bottleneck among the four candidate causes, to say what the failures are made of, and to be honest about the stochastic noise floor that any single-seed comparison has to clear.

## 6.1 Experimental setup

The solver was run over the full 400-task ARC-AGI-1 training set under three compute budgets, on a single machine running one task per core, using the shipped default configuration and seed 47. The reported cost is the elapsed time of the whole run. The worker count was not recorded in the run summaries and is derived as 400 × mean task time/wall, giving 23.4, 30.4 and 29.8 for the three budgets; "30 cores" below is that derivation rounded, and the short configuration sits furthest from it. Except for the sets listed below, the figures in this chapter come from a single pinned build on Python 3.12.3, recorded in Table C.1, with population 40, 60 generations, 5 restarts, at most 5 rules, $\varepsilon$-lexicase selection at `fine2` granularity, the Occam and view continuations enabled, the parser cache on at 1 500 MiB per worker, and the macro library off. The training tasks carry 3.255 train pairs on average. Four sets of figures come from elsewhere and are flagged where they appear: the fourth budget of Section 6.2; the C++ engine run of the same section, which is also the ARC-AGI-2 training row of Section 6.6; the union over the development log, which pools runs of differing builds by construction; and the selection comparison of Section 6.2.

A task counts as *resolved on training* when every train pair is reproduced bit-exact, which is the same discontinuity the lexicographic fitness (Section 5.2) is built to favour. It counts as resolved on test when the same holds for the withheld test pair, which the search never sees. Test-side figures are reported at one attempt and at two, the latter being the protocol the benchmark itself uses.[1] Python hash randomization was pinned for every run reported here, for reasons Section 6.10 explains.

## 6.2 Results on ARC-AGI-1

**Table 6.1:** Resolution on the 400 ARC-AGI-1 training tasks (July 2026, seed 47). $t_{\text{task}}$ is the per-task wall-time budget. "train" and "test" count tasks whose every train (resp. bundled-test) pair is reproduced bit-exact; "@2" allows the two attempts of the benchmark protocol; "both" counts tasks resolved on the train pairs *and* the withheld pair. "avg gen" is the mean number of generations evaluated per task, "avg/task" the mean per-task wall time, "wall" the elapsed time of the run on 30 cores. All three configurations run population 40, 60 generations and 5 restarts. No task failed in any configuration.

| Config. | $t_{\text{task}}$ | train | test | @2 | both | avg gen | avg/task | wall |
|---|---|---|---|---|---|---|---|---|
| Short | 120 s | 104 | 92 | 92 | 85 | 28.2 | 137 s | 0.65 h |
| Medium | 480 s | 142 | 116 | 118 | 112 | 129.1 | 408 s | 1.49 h |
| Long | 960 s | 161 | 133 | 138 | 127 | 247.6 | 752 s | 2.80 h |

As a fraction of the 400 tasks, the training-side counts are 26.0, 35.5 and 40.3 percent, and the two-attempt test-side counts are 23.0, 29.5 and 34.5 percent.

The approach scales with compute, monotonically in every column and without saturating over the range tested. The short 120-second budget already resolves 104 tasks on training; quadrupling it to 480 seconds raises the count to 142, and doubling it again to 960 seconds to 161. Diminishing returns are visible, since the first quadrupling buys 38 tasks and the subsequent doubling 19, but the curve has not flattened, which suggests the reported figures are a property of the budget rather than of the method's ceiling.

[1] This distinction was a point of confusion for readers of the companion paper and is worth stating plainly: "test" means the test pair or pairs bundled with each of the 400 training tasks, used as an unoptimized check of generalization. It is not the 400-task public evaluation split, which is out of scope for this chapter.

The test-side column is the one that matters, and it tracks the training side closely. At the long budget 133 of the 400 tasks, one third of the benchmark, are solved on the withheld test pair, rising to 138 under the two-attempt protocol. The gap between the two columns is the price of programs that reproduce every train pair and still fail to generalize: 34 of the 161 fitted programs miss the withheld pair, while 6 tasks are solved by a program that did not reproduce every train pair, which is what leaves a net difference of 28. That 21 % overfitting rate, rather than the absolute count, is the most informative number in the table.

In absolute terms the long configuration resolves 161 of the 400 training tasks and 133 of the test pairs in 2.8 hours on 30 cores of a single machine, without any task-specific tuning and without a neural network anywhere in the loop.

**A fourth budget, from outside this set of runs.** An earlier build, at the same population, generation and restart settings and the same seed, was run at 3 840 seconds per task over a 1 000-task superset that contains 391 of these 400 tasks.[2] On those 391 it fits 184 and solves 141, against 160 and 132 for the 960-second configuration measured on the same subset. Quadrupling the budget once more therefore buys 24 further fitted tasks and 9 further solved ones. The training side is still climbing while the test side is flattening, which is the same divergence the rest of this chapter reports.

**The best figure on record, from the C++ engine.** The strongest ARC-AGI-1 result in the whole development record comes from neither of the builds above. The C++ reimplementation of the solver, run under the label `c-run-v3`, fits the train pairs of 211 tasks and resolves the withheld pair on 145 at one attempt and 163 at two, over the same 391-task subset as the previous paragraph. Two things about it belong with the numbers, and they are the reason the figure sits here rather than in Table 6.1.

The engine differs. The C++ implementation counts generations by summing across restarts, so its budget is not comparable one-to-one with the Python configurations at an equal wall-clock setting, and no like-for-like comparison between the two engines at a matched budget was run. Against that, it is the most thoroughly checked run in the census: every predicted grid was compared against the released solutions rather than trusted from a boolean flag, and on the ARC-AGI-2 training set, where this run claims the most, all 203 of its first-attempt solves were confirmed and none refuted. It is also free of the submission-writer fault that hit the Python runs, which replicated the first attempt across every test input of a

[2] An earlier build than the one behind the rest of this chapter, as recorded in the companion paper; its run summary carries no commit.

multiple-input task and so could report a solve the grids did not support. It is the best measured figure in the record; Table 6.1 remains the reference for the budget-matched comparison across configurations.

**What every run reaches between them.** One figure from outside these runs belongs beside them, with its status stated carefully. Pooling the 96 runs in the development log, the union over the 400 training tasks reaches 243 fitted and 217 solved on the withheld pair at two attempts, against 161 and 138 for the best single run of Table 6.1. This is an *oracle* over runs of differing engines, budgets, seeds and builds: no configuration achieves it, and nobody can reproduce it by running the solver once. It is reported because of what the comparison implies rather than as a performance claim, and Table 2.1 labels it that way.

What makes it informative is the direction of the gap. The union misses only one of the tasks the single run solves, so per-seed variance compounds across runs instead of cancelling, which is the same effect Section 6.10 measures across five seeds. The 217 is therefore a *floor* under what the current operator set can reach at some budget, not a ceiling, and the floor is soft in both directions. It is soft upward because the census is admittedly incomplete: runs held on a second machine were never folded in, and two classes of run were at first invisible to the aggregator, so the figure has already moved once. It is soft downward because eleven of the solves counted in the pooled totals were refuted when the predicted grids were checked against the released solutions, and a further ninety-nine could not be checked at all, since most runs record only a boolean flag and never save the grid they produced.

The two figures bracket the approach without either being the headline. What one reproducible run delivers on demand is 127 tasks solved on the train pairs and the withheld pair together. What the present language and search reach when their stochasticity is exploited rather than merely suffered is something above half the set, which is why ensembling (Section 6.10) is the cheapest unexploited gain in the system.

**Reading the scaling curve.** The shape of the gain, large at first and then diminishing, is what one expects from a search whose per-task difficulty is heavy-tailed. The easy tasks are found in the first few seconds regardless of budget; the extra minutes buy the harder tasks, and each further doubling reaches a thinner slice of the remaining tail. This has a direct consequence for how the result should be read. A solved-count is meaningless without the budget that produced it, and a fair comparison between two systems has to hold the budget fixed or report the whole curve. It also sets a realistic expectation for the ceiling under pure scaling: throwing ten times the wall-clock at the current system would not close the gap to 100 %, because the tail is populated not by tasks that need more search but by tasks the DSL cannot express at any budget (Section 6.9). Compute scaling and

expressiveness are different axes, and the curve says the system is closer to the limit of the first than of the second.

**The value of not stopping at the first fit.** One secondary counter is worth reporting on its own, because it measures a design decision rather than a budget. The solver records both the first program that fits every train pair and the program finally selected after the Occam and view continuations have spent the remaining budget. Comparing them isolates what the continuations buy: 88 against 92 solved at 120 seconds, 106 against 116 at 480, and 123 against 133 at 960. The continuations are worth 4, 10 and 10 tasks respectively. This is the measured value of refusing to stop at the first program that fits the train pairs, and an indirect argument for the leave-one-out term in the rank tuple (Section 5.2): among programs that all fit the evidence, which one is selected matters by about ten tasks at the budgets that count.

**Deployment metrics.** The figures above are the conservative ones. The submission builder places the view-ensemble alternative in the second attempt and promotes it to the first on a consensus swap, when at least two views agree on an alternative and none back the primary. Those deployed metrics are higher: 93, 117 and 138 at one attempt across the three budgets, and 96, 121 and 146 at two. The second of those exceeds plain pass@2 by construction, since it falls back to it when no distinct view theory exists. This chapter reports plain pass@1 and pass@2 throughout, which understates the deployed configuration by one to five tasks at one attempt and three to eight at two.

**Selection matters more than it looks.** Parent selection is $\varepsilon$-lexicase at `fine2` granularity in the shipped configuration. Measured against a tournament baseline on a paired 200-task ARC-AGI-2 training slice, on an earlier commit than the rest of this chapter, lexicase raised the train-and-test count from 28 to 37, a relative gain of 32 percent. Two caveats keep this out of the headline. The comparison is not on ARC-AGI-1, and the paired runs behind it are no longer archived, so the figure should be re-measured before it is relied on. Taken at face value it is well outside the absolute noise band of Section 6.10, which suggests selection is one of the few non-DSL levers that pays off, but a relative percentage and an absolute band are not directly comparable.

**The per-task budget is soft.** The budget deserves a note, because it is enforced by polling rather than by preemption: it bounds when new work may *start*, not when work already running must stop. A single candidate evaluation on a heavy grid cannot be interrupted, so a task whose very first generation is expensive overruns

its budget entirely. Every overrunning task has zero completed generations, which is the signature of one uninterruptible evaluation. The median task time equals the budget to within a second in all three configurations (120.4, 480.3 and 960.3 seconds), but 52 of the 400 tasks exceed it by more than a quarter at the 120-second setting and 17 do so at 480 seconds. The worst offender, `484b58aa`, costs about 2 000 seconds at either setting and is unsolved both times; its cost is independent of the budget it was given.

This matters for anyone checking the arithmetic. A reader computing $400 \times 120/30 \approx 0.44$ hours against the reported 0.65-hour wall will find a 46 % discrepancy. The overrunning tail explains part of it, since at the observed 137-second mean task time the same arithmetic gives 0.51 hours; the effective worker count explains the rest, 23.4 rather than 30 for this configuration.

**The cost comparison, made explicit.** The reviewers of the companion paper asked for the comparison to the large-model results to be stated on a common footing rather than left implicit, and the honest statement is that the two are not directly comparable and should not be presented as if they were. The reasoning-model results that exceed ninety percent on ARC-AGI-1 are obtained with inference budgets measured in dollars per task and, in some cases, task-specific training; the numbers here are obtained on one 30-core workstation completing the whole 400-task run in under three hours. The right way to read the contrast is not "33 % versus 90 %," a comparison that flatters neither side honestly, but "what does each approach cost per task solved, and how does that cost scale." On that axis the symbolic search is cheap and flat while the large-model route is expensive and, on the evidence so far, rising. Whether the cheap route can be pushed to competitive accuracy is the open question the rest of this chapter, and the thesis, is really about.

## 6.3 A worked example

Figure 6.1 shows one of the resolved tasks, solved on training and on the withheld test pair alike. Task `31aa019c` asks the solver to keep the rarest-colour object, strip everything else, and outline what remains. The grids are $10 \times 10$, with three train pairs and one test input. The winning genome is two rules and a global transform:

| | |
|---|---|
| rule 1 | `least_frequent_color_objects` → `keep_selected` |
| rule 2 | `all_subobjects` → `outline_object(connectivity=8, color=2)` |
| global | `identity` |

The first rule selects the least-frequent-colour objects and keeps them, dropping everything else; the second takes every sub-object among the survivors and outlines

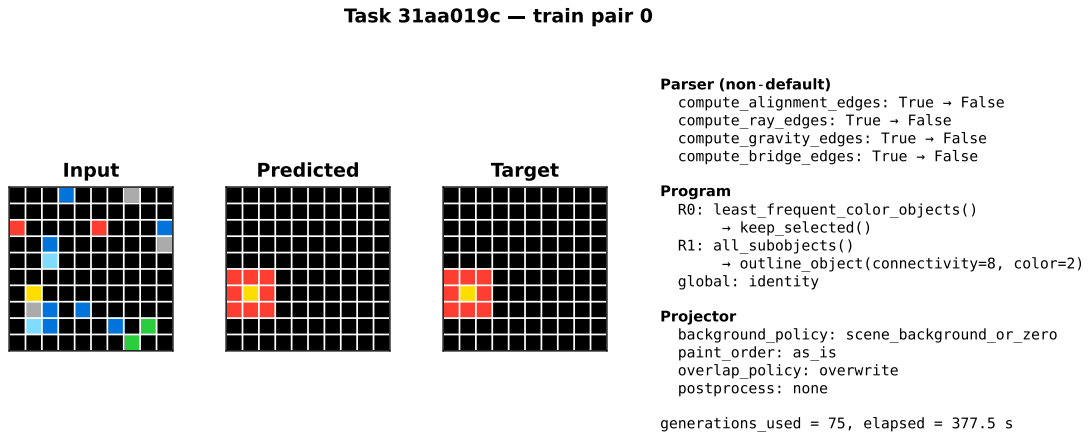



**Figure 6.1:** Task `31aa019c` under the winning genome: the two-rule program, the non-default parser flags, the projector policies, and the bit-exact reconstruction of the first train pair.

it under eight-connectivity, drawing the outline in colour 2. The outline colour is a literal rather than a property of the object: it is the same in all three pairs while the kept object's colour varies. The global-transform slot is the identity, so all of the work is in the two rules. The solver reached this program in 377.5 seconds of the 480 it was allowed. It is not solved at the 120-second budget, which makes it a fair illustration of what the extra compute buys.

The parser configuration inverts the expectation the design invites. The winning genome switches *off* four edge-generation passes, alignment, ray, gravity and bridge, and leaves every other flag at its default. The search did not need a richer graph here; it needed a sparser one. Those four families of edge are noise for this task, and removing them shrinks the set of selectors that type-check, which makes the two-rule program easier to find. Where that configuration comes from is worth being exact about: those four fields sit outside the mutated set (Section 4.2), so the search did not reach them by mutation. It inherited them from one of the six parser configurations the seed enumerator tries in Phase A (Section 5.1), the one that disables the four relational edge families and changes nothing else. The per-task adaptation here is therefore a choice among seeded representations rather than a walk through the parameter space, which is still something a single fixed representation could not do.

## 6.4 Program parsimony

The worked example is two rules long, and a reviewer of the companion paper asked the natural follow-up: is that representative, or a well-chosen illustration? Table 6.2 answers it with the distribution of program lengths over every task solved on both the train pairs and the withheld test pair, at each of the three budgets.

**Table 6.2:** Number of rules in the winning program, over tasks resolved on training *and* test. A zero-rule program is a bare global transform. The median is one rule at every budget.

| Rules | 120 s | 480 s | 960 s |
|---|---|---|---|
| 0 (global transform only) | 24 | 24 | 26 |
| 1 | 45 | 62 | 66 |
| 2 | 10 | 15 | 18 |
| 3 | 6 | 9 | 15 |
| 4 | 0 | 2 | 1 |
| 5 | 0 | 0 | 1 |
| Total | 85 | 112 | 127 |
| At most one rule | 81 % | 77 % | 72 % |
| At most two rules | 93 % | 90 % | 87 % |

Solutions are short. The median program is a single rule at every budget, and roughly nine in ten use at most two. The distribution drifts only slightly toward longer programs as the budget grows by a factor of eight: the three-rule column grows from 6 to 15 while the one-rule column grows from 45 to 66, so additional compute buys mostly *more* solutions rather than *longer* ones.

This is a property of the representation rather than of the tasks. The work that a pixel-level DSL would have to express as a long program is absorbed by the parser, which chooses how to see the grid, and by the global-transform slot. What remains for the program to express is usually a single typed rewrite. It also explains why the complexity term sits so low in the rank tuple: with programs this short there is little for a parsimony pressure to do, and the search does not need to be discouraged from bloat it does not produce.

The result matters for the argument of the thesis. If exact solutions were typically long, the “search for a short program” framing would be wishful, and the space would be Vast in a way no per-task budget could cover. The observed shortness is what makes the budget viable.

## 6.5 Failure analysis

An aggregate count of unsolved tasks says nothing about what is missing. Table 6.3 groups the 400 training tasks by the kind of transformation they ask for. The families overlap, and the last row is the baseline for comparison.

**Table 6.3:** Solve and failure rates at the 960-second budget by the kind of transformation required. "never fitted" counts tasks for which no program reproduced even the train pairs.

| Family | tasks | solved | never fitted |
|---|---|---|---|
| Tiny output, at most 9 cells (counting, selection) | 57 | 22 | 26 |
| Output size not a fixed ratio of the input | 69 | 20 | 42 |
| Output shape varies across the train pairs | 62 | 18 | 37 |
| Grids of 400 cells or more | 31 | 6 | 25 |
| Seven or more colours | 21 | 4 | 17 |
| Input and output the same shape (baseline) | 262 | 88 | 158 |

The pattern is not the one the shape families suggest. Tasks whose output size or shape has to be derived from the content solve at 20 of 69 and 18 of 62, only a few points below the same-shape baseline's 88 of 262, and tasks with a tiny output do better than that baseline, at 22 of 57. What separates clearly is scale and palette: grids of 400 cells or more solve 6 of 31 and tasks with seven or more colours 4 of 21, and four in five of each are never fitted at all, against three in five for the baseline. A search whose per-candidate cost grows with the graph, and whose colour arguments come from a per-task prior that widens with the palette, would fail exactly here.

Four failures are worth walking through. The first three were picked from the near-miss tail using the triage the development tooling already applies to a finished run: tasks are bucketed by the normalized mismatch of the rank-1 program, and anything under 0.10 counts as almost correct. The wrong cells were then read by hand. The bias this introduces is worth stating: a near miss is legible, so these are the failures whose cause can be named, and they are not representative of the 43 tasks above 0.5 mismatch where the search finds no purchase at all. The fourth case is drawn from a different population, the tasks that a longer budget loses.

**A size the language cannot derive.** Task `d631b094`, the top row of Figure 6.2, is a counting task. Each input is a $3 \times 3$ grid holding between one and four coloured cells, and the output is a single row of that many cells in that colour. The evolved program selects the largest subobject and applies a fold-and-tile action with a

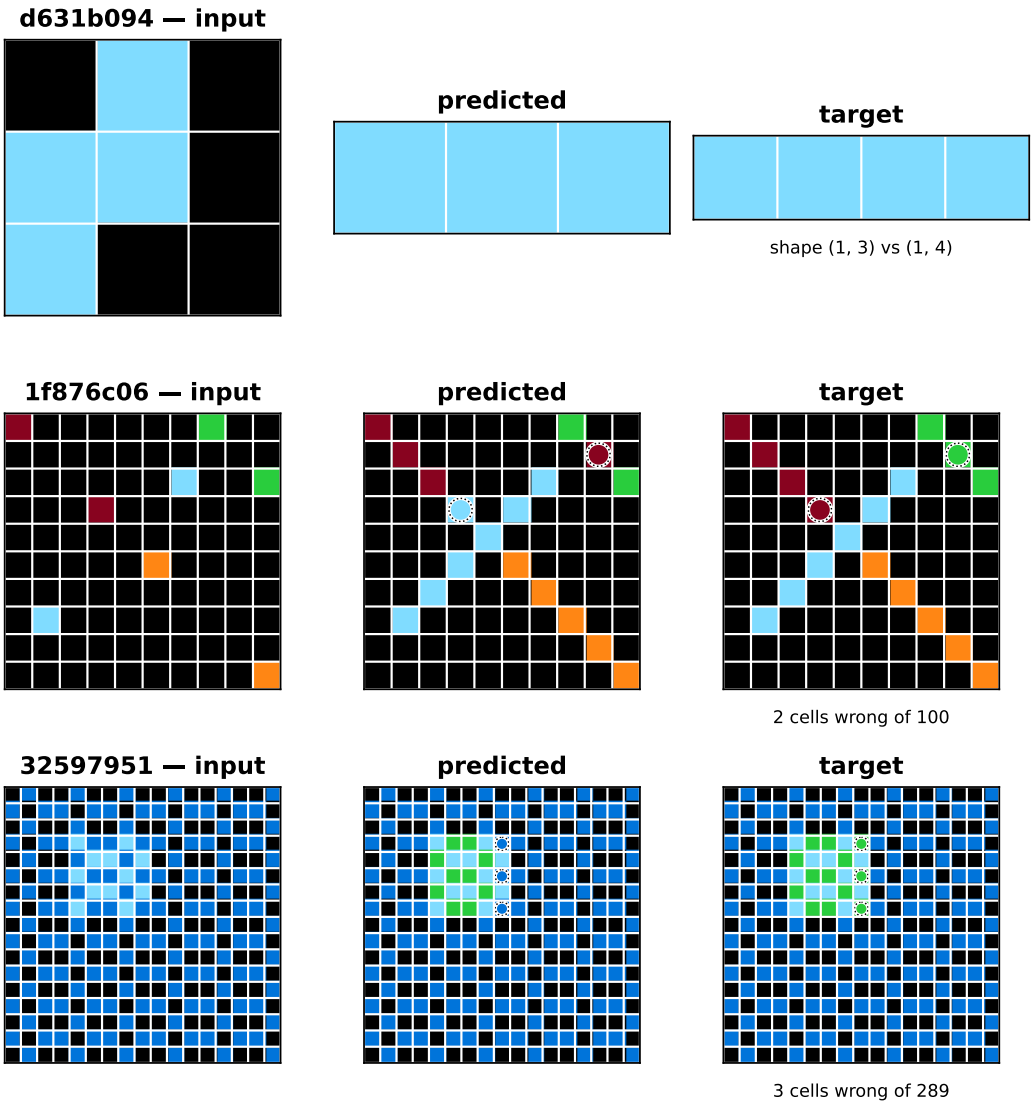



**Figure 6.2:** Three of the four failures, one per row: input, the grid the evolved program produces, and the target, with the disagreeing cells ringed. Top, `d631b094`, where the output should be as long as the number of coloured cells and the program emits a fixed width. Middle, `1f876c06`, where two of a hundred cells are wrong and both are trail collisions. Bottom, `32597951`, where three cells of the periodic background on the shape's boundary are recoloured in the target and not by the program.

horizontal divisor of 3, under a transpose. It reproduces the pairs whose answer is one, two and three cells wide, and produces a three-wide row for the pair that needs four. The divisor is an integer argument that the search fixed by fitting the pairs it saw. The DSL does contain one operator whose output size is data-derived, `SCALE_BY_COUNT`, but the count it binds to is the number of distinct non-background colours or of non-background objects, and it applies that count as a scale or tile factor on the whole grid; every other size-changing operator takes its factor as a literal. Neither form can produce a row whose *length* is the number of coloured cells. Tiny-output tasks as a family are not out of reach, solving at 22 of 57 (Table 6.3); what is out of reach is the subfamily whose output extent is a cell tally, and not because the search is weak but because the one count-derived size operator counts the wrong thing.

**Right program, wrong write semantics.** Task `1f876c06`, the middle row, fails for the opposite reason: the program is right and the semantics are wrong. Coloured singletons scattered on a $10 \times 10$ grid have to be joined by diagonal trails, which is what the evolved program does, and on the second train pair it places 98 of the 100 cells correctly. Both wrong cells are collisions. At one, two trails of different colours cross and the program keeps the colour painted first where the target keeps the other; at the second, a trail paints over a cell that already held an input singleton. Nothing in the rule language expresses priority at an intersection, and the projector's overlap policy is a single global setting rather than something an individual rule can choose.

**A failure no exact-match count would reveal.** On `32597951`, the bottom row, the program never reproduces a single train pair, yet its mean mismatch is 0.018: over three $17 \times 17$ grids it misplaces 4, 9 and 3 cells out of 289. Every one of those cells is the same error. The task overlays a periodic background of ones and zeros with a small azure shape, and asks for the background cells inside the shape's footprint to be recoloured, the shape itself staying as it is; the program recolours part of that interior and leaves the rest untouched, where the target recolours them too. The rule fires on the right objects with the right colour, and the disagreement is entirely about where the shape ends and the background resumes, which is a question the parser answers when it segments the grid and the program never gets to revisit. Cases like this look like total failures in the headline table and are one segmentation decision away from being solutions, which is an argument for reporting the mismatch distribution alongside the counts rather than instead of them.

**More compute made it worse.** The fourth case is the most damaging to the scaling argument. Task `44f52bb0` is a classifier: six $3 \times 3$ grids of a single colour

on a black field, each answering with one cell, either colour 1 or colour 7. At the 120-second budget the search returns a three-rule program that counts holes and recolours accordingly. It fails one train pair, so the task is not even counted as fitted, and it answers both withheld test pairs correctly. At 960 seconds the search does better by its own measure and worse by the benchmark's: it finds a five-rule program that reproduces all six train pairs exactly, and that program answers colour 1 to both test inputs, so it gets one right by coincidence and the task counts as solved on training and failed on test. The extra budget bought a program that fits the evidence and has stopped modelling the task. This single task is the overfitting gap of Section 6.2 in miniature, and it also shows why a one-cell output flatters any measure of correctness: at that size, with two colours ever answered, a wrong program has an even chance of looking right.

## 6.6 ARC-AGI-2

Everything above is ARC-AGI-1. The harder benchmark was also run, and the result is worth reporting plainly rather than deferring, because it is the clearest available statement of what the current approach cannot yet do.

**Table 6.4:** ARC-AGI-2. "train" counts tasks whose every train pair is reproduced bit-exact; the pass@$k$ columns are the withheld pair under $k$ attempts. The held-out evaluation run is on the same commit as every ARC-AGI-1 figure in this chapter; the training row is the C++ engine run of Section 6.2 and is reported with that caveat.

| Set | tasks | train | pass@1 | pass@2 | pass@5 | build |
|---|---|---|---|---|---|---|
| Held-out evaluation | 120 | 3 | 2 | 2 | 2 | Python |
| Training | 1000 | 287 | 203 | 211 | 216 | C++ |

On the 120-task held-out evaluation split, at a 3 840-second per-task budget, the solver fits the train pairs of 3 tasks and solves 2 on the withheld pair. There were no failures, the run took 4.31 hours, and the average task time was 3 841.9 seconds against a 3 840-second budget, meaning essentially every task consumed its entire allowance. Both predictions were checked against the released solutions and both hold. One qualification belongs with the count: of the two solved tasks, only one is unique to ARC-AGI-2, the other being one of six tasks this evaluation split shares with the ARC-AGI-1 evaluation set. That split is out of scope elsewhere in this chapter, but a task drawn from it cannot count as new here. The honest reading is therefore 2 of 120, only one of them new to ARC-AGI-2, and neither should be quoted as a rate. The fit count is the diagnostic: when only 3 of 120 tasks fit the

train pairs at all, the search is not converging and stopping, it is running out of time everywhere.

Two out of 120 is close to nothing, and it should be read as such. It is also exactly what Section 2.2 and Section 2.3 predict. ARC-AGI-2 was rebuilt specifically to defeat brute-force program search over a fixed operator set, and the failure analysis of Section 6.5 already locates this solver's limit in the expressiveness of its rewrite language. A benchmark designed to demand multi-step compositional reasoning meets a system whose median winning program is one rule. The mechanism of the failure is not mysterious.

The training-split row is more encouraging (287 of 1 000 fitted, 203 solved) but it carries a provenance gap: it comes from the C++ engine run of Section 6.2, on a build that predates every other number in this chapter, so it is not directly comparable to the held-out row above or to the ARC-AGI-1 table. Re-running it costs roughly 35 hours at 30 workers and has not been done. It is reported here because it is real and because the ratio it implies, a fifth of the training split solved, is a more informative estimate of the current ceiling than the held-out figure alone; but the two rows should not be read as one experiment.

The honest summary is that the approach transfers to ARC-AGI-2 poorly, that the reason is the one the ARC-AGI-1 diagnostics already identified, and that compositional parsing (Section 7.1) and a stronger proposal distribution are the changes that would have to land before a rerun is worth the compute.

## 6.7 Head to head on ARGA's own subset

Section 2.6 left the comparison against ARGA unresolved, because its 35.6 % is measured on 160 tasks its authors chose and renormalizing that over 400 assumes the 240 they skipped would all have failed. The comparison can be settled directly instead, by running the same 160 tasks. The subset is published with ARGA as three category files, *recolor* (62), *movement* (31) and *augmentation* (67); all 160 lie inside the ARC-AGI-1 training split this thesis already reports on, so no new data is involved.

The C++ engine was run over the 160 at 840 seconds per task, seed 47, population 40, 5 restarts, at most 5 rules, macro library off. Every predicted grid was scored against the released solutions rather than trusted from a boolean flag, and the five tasks with more than one test input require all of them correct to count. Table 6.5 gives the result beside the three published systems.

Two readings follow, and the second is the useful one. The first is the ranking: this work solves 75 against ARGA's 57 and the Kaggle winner's 64, at under half their time budget, on ground that favours ARGA, since its language was developed on this very subset. GPAR leads by six. Renormalization is no longer needed for

**Table 6.5:** ARGA's 160-task subset. "Fitted" counts tasks whose program reproduces every train pair, "solved" those that also produce the correct test output; both are the definitions used by Lei et al. [24], whose Table 5 supplies the three comparison rows. Single attempt throughout, for comparability; under the benchmark's two-attempt protocol this work reaches 78.

| System | Fitted | Solved | Share | Budget |
|---|---|---|---|---|
| GPAR [24] | 86 | **81** | 50.6 % | 1 800 s |
| This work | **102** | 75 | 46.9 % | **840 s** |
| Kaggle first place [22] | 79 | 64 | 40.0 % | 1 800 s |
| ARGA [23] | 63 | 57 | 35.6 % | 1 800 s |

the ARGA comparison: the subset is simply run.

The second reading is the split between the two counts. This work fits 102 of the 160, more than any of the three, and converts the fewest of what it fits: 26 % of fitted programs fail the test pair, against GPAR's 6 % and ARGA's 10 %. That is the discrimination problem of Section 6.11 measured from outside, on a benchmark this thesis had no hand in designing, and it localizes the remaining gap precisely. The representation search is the strongest component here by the only measure the comparison offers; what follows it is the weakest. A system that fits the most and converts the least does not need a larger operator inventory before it needs a better way of choosing among the programs that already fit.

## 6.8 ConceptARC

ARC-AGI reports a single solved count, so a failure tells you that something is missing but not what. ConceptARC (Section 2.3) is built to answer the second question: its 160 tasks are grouped into sixteen concepts, ten tasks each, with three test inputs per task, so the score decomposes by the abstraction a task demands. That makes it the natural check on the claim of Section 6.9, which localizes the ceiling in the operator set rather than in the parser. If that claim is right, the weakness should not be spread evenly across concepts; it should concentrate where the missing operators live.

The solver was run over the whole corpus at 1 800 seconds per task, seed 47, population 40, 5 restarts, at most 5 rules, macro library off. Following the corpus convention the denominator is the test input rather than the task, so 160 tasks give 480 items. It reproduces every train pair on 135 of the 160 tasks and resolves 139 of the 480 test inputs at one attempt, 29.0 %, rising to 36.9 % at two. A second run at 840 seconds per task differs on five tasks and lands at the same 29.0 % and 37.1 %, which is the saturation the rest of this chapter reports rather than a new

effect.

**Table 6.6:** ConceptARC by concept group, percentage of test inputs resolved at one attempt. The comparison columns are the two strongest entries of the 2020 ARC competition and GPT-4 at temperature 0, as published with the corpus [6].

| Concept group | This work | Kaggle 1st | Kaggle 2nd | GPT-4 | Human |
|---|---|---|---|---|---|
| TopBottom2D | 60.0 | 60.0 | 56.7 | 23.3 | 95.4 |
| ExtractObjects | 53.3 | 43.3 | 43.3 | 3.3 | 86.3 |
| SameDifferent | 53.3 | 53.3 | 16.7 | 16.7 | 88.3 |
| HorizontalVertical | 46.7 | 43.3 | 10.0 | 26.7 | 91.1 |
| Center | 40.0 | 50.0 | 20.0 | 33.3 | 94.3 |
| CleanUp | 36.7 | 50.0 | 20.0 | 20.0 | 97.4 |
| ExtendToBoundary | 33.3 | 76.7 | 46.7 | 6.7 | 93.4 |
| AboveBelow | 26.7 | 70.0 | 33.3 | 23.3 | 90.5 |
| FilledNotFilled | 26.7 | 73.3 | 43.3 | 16.7 | 95.9 |
| InsideOutside | 20.0 | 56.7 | 10.0 | 10.0 | 90.9 |
| TopBottom3D | 20.0 | 60.0 | 3.3 | 20.0 | 92.9 |
| Copy | 16.7 | 23.3 | 26.7 | 23.3 | 93.6 |
| MoveToBoundary | 13.3 | 36.7 | 30.0 | 20.0 | 91.0 |
| CompleteShape | 6.7 | 46.7 | 30.0 | 23.3 | 85.1 |
| Order | 6.7 | 26.7 | 23.3 | 26.7 | 82.8 |
| Count | 3.3 | 60.0 | 40.0 | 13.3 | 88.0 |
| Overall | 29.0 | 51.9 | 28.3 | 19.2 | 91.1 |

Table 6.6 is the point of running the corpus at all. The overall figure places the solver above the second Kaggle entry and both GPT-4 settings and below the first, which is roughly where the ARC-AGI-1 comparison of Section 2.6 would predict. The decomposition is the informative part, and it is not flat.

The groups at the top are the ones the graph representation is built for. *ExtractObjects* at 53.3 and *HorizontalVertical* at 46.7 are the only two groups where this solver beats the brute-force winner, and both turn on identifying objects and their relations rather than on rewriting pixels; *TopBottom2D* at 60.0 and *SameDifferent* at 53.3 draw level with it. Four of sixteen groups is not a claim of superiority, but the four are of one kind, which is the representation earning its keep measured per concept rather than in aggregate.

The bottom of the table is the confirmation. *Count* is the worst group in the corpus at 3.3 %, against 60.0 for the brute-force winner and 88.0 for humans, and it is the single largest gap between this solver and any baseline anywhere in the table. Section 6.9 names one operator whose size argument is bound to the wrong quantity, and Section 6.5 shows a task failing for exactly that reason. A corpus assembled by another group, organized by a taxonomy this thesis had no hand in,

independently isolates counting as the weakest ability of the solver. *Order* at 6.7 and *CompleteShape* at 6.7 are the next two, and both need a rule to depend on a quantity computed from the scene, which is the same missing capability seen from a different angle.

This does not widen the diagnosis; it sharpens it. The claim that the ceiling is in the operator set was argued in Section 6.9 from a parser audit, a budget count, and four hand-read failures, and was called a well-supported hypothesis rather than a proof. ConceptARC supplies the independent test that argument was missing, and the prediction it makes, that the weakness concentrates rather than spreads, is the one the data shows.

## 6.9 Where is the bottleneck?

A recurring question, raised by both reviewers of the companion paper, is which of the four components limits performance: parser, DSL, search, or fitness. Taken together, the diagnostic below and the failure cases above place it in the DSL.

**Not the parser.** A diagnostic built during development parses every training task under the default configuration, dumps per-step node and edge counts, and z-scores each column across tasks. Among the 14 near-miss tasks of a 50-task diagnostic subset (normalized mismatch below 0.10 but not solved), one was flagged on a per-metric extreme and none on cross-pair divergence. On this sample the near-miss tasks parse no less consistently than the solved ones: the parser's output on the failing set is not the messier of the two. Two further observations point the same way. Where a hand-labelled reference graph exists, the audit evolver of Section 5.4 can drive the graph distance to near zero, which shows the parser is *capable* of the right representation and locates any residual failure downstream of it. And 46 % of the tasks that run solves leave the parser at its default configuration, so the evolvable representation is not where most of the work is being done. On this benchmark, at least, the modelling is not the measurable bottleneck, though it is expected to become one on ARC-AGI-2.

**Only marginally the search budget.** At the long setting the budget starves 14 of the 239 unfitted tasks, in the sense that they completed no generation at all before the deadline fired. That is a small enough share to rule the budget out as the principal constraint, and it is consistent with the scaling curve, which is flattening on the test side.

**Partly the fitness.** The fitness cannot separate the 34 programs that fit every train pair and then fail the withheld one. That is a real share of the gap, and

Section 6.11 reports what happened to every attempt at fixing it.

**Mostly the DSL.** The largest share belongs to the operator set, and the failure cases name the gaps precisely: the one operator whose size argument is data-derived binds to the wrong quantity, and write semantics cannot be specified per rule. The third failure case is the exception that qualifies the "not the parser" claim, since a segmentation boundary is a parser decision, and it suggests the parser does become the bottleneck once the program is otherwise correct.

**Three improvements follow, in increasing order of cost.** Extending `SCALE_BY_COUNT`, or adding a sibling, so that the bound count can be a cell tally and the derived number can set an output extent rather than only a scale factor, would open the tasks whose output extent is a cell tally. A local write policy, or a rank-tuple term that counts overwritten cells, would recover a class of near misses that currently look like search failures and are not. Allowing a rule to adjust the segmentation it inherits, rather than only to rewrite the objects the parser produced, would address the third case and is the only one of the three that touches the representation.

## 6.10 Trajectory drift and the noise floor

Any single-seed comparison on this system has to reckon with neutral drift (Section 3.4). Five seeds were run at 120 seconds per task over the full 400 tasks under identical code and configuration, at a total cost of 1.99 hours for the four extra seeds. Table 6.7 gives the per-seed counts.

**Table 6.7:** Five seeds at the 120-second budget over the 400 training tasks, identical configuration. "both" counts tasks resolved on the train pairs and the withheld pair. Seed 47 is the shipped default and the seed behind Table 6.1.

| Seed | train | pass@1 | pass@2 | both |
|---|---:|---:|---:|---:|
| 41 | 106 | 101 | 101 | 95 |
| 42 | 106 | 96 | 96 | 90 |
| 43 | 109 | 97 | 100 | 92 |
| 44 | 106 | 100 | 101 | 94 |
| 47 | 104 | 92 | 92 | 85 |
| mean ± sd | $106.2 \pm 1.8$ | $97.2 \pm 3.6$ | $98.0 \pm 3.9$ | $91.2 \pm 4.0$ |

The training-side count is stable at $106.2 \pm 1.8$ tasks, spanning 104 to 109, while the test-side count is not: $97.2 \pm 3.6$ at one attempt, spanning 92 to 101.

The asymmetry is the informative part. The search reliably finds *a* program that fits the train pairs; which of the fitting programs it happens to find is what varies, and that choice is what determines whether the task generalizes. The stochasticity is therefore noise in the selection among equally training-perfect candidates rather than noise in the search, which is precisely the discrimination problem of Section 5.2 reappearing as variance.

Single-seed comparisons below roughly five tasks are therefore indistinguishable from luck, and eight entries in the development log are reverts of positive-looking deltas that disappeared on re-measurement. Those reverts were decided on single-seed 400-task runs; the multi-seed harness landed late, which is itself one of the record's lessons. Seed 47 is the weakest of the five on every column, so the figures reported throughout this chapter sit below the mean rather than above it. They are conservative rather than cherry-picked, which is worth saying explicitly, because a reader who sees a standard deviation of 4 beside a headline of 85 will otherwise wonder which way the selection went.

**Ensembling is the obvious exploitation.** Across seeds 41 to 44 the union of tasks solved on both sides is 113, against a per-seed mean of 92.75 and an intersection of 75. The vote histogram is the interesting part: 75 tasks are solved by all four seeds, 11 by three, 11 by two, 16 by exactly one, and 287 by none. So 38 of the 113 are found by some seeds and missed by others, and the union beats the average seed by some 22 % for nothing but repeated runs. It is not part of the reported configuration, which is single-seed throughout, but it is the cheapest available gain and it follows directly from the variance being in selection rather than in search.

**The determinism fix underneath everything.** One subtlety compounds all of this, and it is the single most consequential finding in the development record. For a long stretch the solver was not deterministic at a fixed seed: unpinned Python hash randomization reordered the sets and dictionaries on the search's decision path, so the same task found different programs on different runs. This was *the* source of a run-to-run swing that had been attributed to trajectory drift, and it is why eight earlier development phases were reverted on evidence that turned out to be noise. Pinning the hash seed did not improve the method; it made the method *measurable*, and every trustworthy comparison in this chapter postdates that fix. It illustrates the methodological claim of this chapter: an unmeasured noise floor silently corrupts a research program.

The practical consequence is a methodology, not just a caveat. Every candidate change in the development log was held to a signal-above-noise bar: a modification that moved the 50-task or 400-task count by less than the drift band was treated as unproven regardless of the sign of the delta, and either dropped or promoted to

a multi-seed test before being believed. Section 6.12 samples that log and draws out the pattern in it, which is that changes adding *expressiveness* clear the bar while changes that only reshape the fitness surface or re-weight the sampler tend to trade tasks rather than add them.

## 6.11 The fitness problem is real

Section 5.2 argued that discriminating between programs that reproduce every train pair exactly but differ in structure is hard. The experiments bear this out. A minimum-description-length tie-break, applied unconditionally and tested on a 25-task subset where the system produced programs that fit all train pairs yet generalized poorly, gained three tasks on that subset in its unconditional form, then came in at $-3$ on the full 400, inside the drift band, and was reverted. It was then re-landed in restricted form, as a tie-break among *solved* candidates only, at a cost of $-5$ train and no change on test, and in that form it survives as the complexity key of the rank tuple. The distinction matters: complexity is a useful discriminator once the frontier has been cleared and a harmful one before that.

Smoothing the fitness surface below the exact-match frontier has, in every variant tried, either flattened the landscape and collapsed search depth or failed to clear the noise floor. The lexicographic ranking with a graph-distance tie-break remains the best-performing formulation found.

Table 6.8 collects the fitness-formula experiments. The through-line is that the exact-match frontier is not a smoothness defect to be engineered away; it is a structural feature of the problem. The reward is discontinuous, because a grid is either bit-exact or it is not, and every attempt to pretend otherwise either misled the search toward plausible-but-wrong programs or added variance without adding signal. The one term that consistently helped was the graph-distance tie-break *below* the frontier, a graded comparison over symbols rather than pixels, which sharpens the ordering among failing candidates without touching the frontier itself. That is the formulation carried into the reported runs.

## 6.12 Lessons from the development record

The results reported above are the surviving tip of a long development record: dozens of changes proposed, measured, and in most cases reverted. That record is itself a result, and a thesis that reported only the survivors would misrepresent how the system was built and would waste the most transferable thing the project learned, namely which *kinds* of change pay off and which only appear to. The full log runs across roughly thirty-five numbered development phases; Table C.2 in Appendix C collects the representative ones. The asymmetry in it is the single most

**Table 6.8:** Fitness-formula experiments. The graph-distance tie-break below the exact-match frontier is kept; every attempt to smooth or replace the frontier is reverted.

| Formulation | Verdict | Outcome |
|---|---|---|
| Lexicographic rank, graph-distance tie-break | kept | best formulation found; used in all reported runs |
| Object-graph distance below the frontier | kept | graded signal over symbols; steers failing candidates |
| Minimum-description-length tie-break, unconditional | reverted | +3 on subset, −3 to drift on 400 |
| Minimum-description-length tie-break, solved-only | kept | −5 train, 0 test; survives as the complexity key |
| Continuous surrogate for "all pairs exact" | reverted | flattened landscape; collapsed search depth |
| Round-trip parse/project agreement term | open | promising in principle; round-trip cost and stability untested at scale |

useful thing the noise-floor discipline surfaced, and it is what points the roadmap at the operator set rather than at the search or the fitness.

**Additive and immutable beats expressive-but-disruptive.** The changes that landed share a shape: they add capability *without* perturbing the random-number trajectory of the existing search. A new action introduced at the lowest sampling weight, a detector that only ever appends a seed, rule inference moved inside an action handler so that seed hashes stay bit-identical: each of these adds reach while leaving the existing search undisturbed, and each cleared the bar. The changes that failed share the opposite shape. They were more expressive in principle but disruptive in practice. Adding two entries to a sampling table re-normalizes every other operator's probability and flips a handful of noise-floor tasks; adding conditional or loop constructs enlarges the language but at an emission rate the search cannot exploit, so the only measurable effect is the trajectory drift the change induced. The lesson, stated once and confirmed repeatedly, is that on a drift-sensitive search the friction a change imposes on the existing trajectory is often larger than the benefit its new capability provides, and the way to add capability is to add it *beside* the search rather than *inside* it.

**Where the ceiling actually is.** The most important negative result is cumulative rather than any single phase. After the determinism fix, the incremental levers were worked systematically (more operators, sharper selection, anti-overfit

ranking terms, budget reallocation) and past roughly a third of the training set they stopped paying off: single-operator additions became net-neutral through sampling dilution, and post-search selection changes came in at about zero because the remaining failures are *wrong-rule-found*, not mis-selected. The union-of-runs bound of Section 6.2 says the same thing from a different direction: 217 tasks at two attempts is what every run ever made reaches between them, so the operator set can already express far more than any single run finds, 243 fitted against 161, which points at the search rather than the inventory. The conclusion the record forces is that three constraints bind, on three different populations of task. Where no program fits the train pairs at any budget, the limit is what the language can express. Where a fitting program is within reach but no single run finds it, which is what the union's 243 fitted tasks show against the best run's 161, the limit is the search. And where a fitting program is found but does not generalize, which is what the 34 of 161 fitted-but-failing programs show, the limit is a fitness that cannot separate it from the others that fit. Closing the gap therefore needs a stronger proposal distribution as much as it needs more operators or another tie-break. That is exactly what motivates the library-learning extension of Section 5.5, which is the first attempt at such a step-change: a learned proposal prior rather than another hand-added primitive.

## 6.13 Limitations and threats to validity

The results above come with caveats that a careful reader should hold in mind, and they fall into four groups.

**Scope of the benchmark.** Most numbers here are on the ARC-AGI-1 *training* split. The headline test-side count uses the test pair or pairs bundled with each of those 400 tasks, which the search never sees, and this is an unoptimized check of generalization rather than the official public-evaluation split, which is out of scope. ARC-AGI-2 is measured in Section 6.6 and the result there is weak, at 2 of 120 on the held-out evaluation split, which is the honest limit of the present approach; the ARC-AGI-2 training row in that section additionally predates the build behind every other figure.

**Verifiability of the counts.** A count is only as good as the artefact behind it, and here the artefacts are uneven. The three runs of Table 6.1 are the strongest case: each saves the predicted grid and the target grid for every train and test pair of all 400 tasks, so every cell of the train and one-attempt columns was re-derived from those grids against the released solutions, with none refuted. What is beyond audit is the pass@5 column of Table 6.4, since only two of the five attempts the

solver generates are persisted. Elsewhere the record is thinner. Across the pooled development log a check against the released solutions refuted eleven solves and left ninety-nine unverifiable, of which four refuted and thirty-one unverifiable fall on ARC-AGI-1 training. A defect in the submission writer compounds this: on tasks carrying more than one test input it replicated the first attempt across all of them, which can turn a partial match into a reported solve, and it affected several runs. The headline table is therefore independently confirmed; it is the pooled union that should be read as an upper estimate of a lower bound.

**Statistical strength.** The headline configurations in Table 6.1 are single-seed runs at seed 47, and Section 6.10 puts the run-to-run spread at ±1.8 tasks on the training side and ±3.6 on the test side at the short budget. The multi-seed measurement was made at 120 seconds only; the 480- and 960-second configurations have not been repeated across seeds, so their counts inherit an unmeasured uncertainty of at least that width. Where this chapter reports a delta between two variants it applies the signal-above-noise bar, but the absolute counts carry the same caveat.

**Hardware and budget dependence.** Because the search is deadline-driven, the solved count is a function of the wall-clock budget and the machine that runs it. The figures here are for one 30-core machine; a faster machine at the same budget would evaluate more candidates and could resolve a different set of tasks. The soft budget compounds this, since 52 tasks overrun the 120-second setting by more than a quarter. This makes the results reproducible only relative to a stated hardware and budget profile (Appendix C), and it means comparisons to systems run on different hardware have to be made on cost-normalized terms rather than raw counts.

**Construct validity of the diagnostics.** The claim that the DSL, not the parser, is the bottleneck rests on a diagnostic over a 50-task near-miss macro, on the 46 % default-parser share among solved tasks, and on four hand-read failures, not on an exhaustive causal analysis; it is a well-supported hypothesis rather than a proof. Three of the four failure cases are drawn from the legible tail and are explicitly unrepresentative of the 43 tasks above 0.5 mismatch; the fourth comes from the tasks a longer budget loses. Likewise, the transfer measurement for the macro library reports coverage of role skeletons in solved programs, which is a proxy for whether the prior lifts unseen tasks rather than a direct measurement of it. The ablation of the next section is what would convert that proxy into a direct result; Section 6.14 reports it, and the outcome is a null.

## 6.14 Macro-library ablation

The library-learning extension of Section 5.5 is evaluated against the same signal-above-noise bar as everything else, and its evaluation is staged so that a null result can be read correctly. The precondition, that mined strategies recur on held-out tasks, was measured first and separately (Section 5.5), covering roughly a third of the held-out tasks that have a multi-rule solution, with the bulk of the utility mass on transferring macros. That number sets the ceiling: a prior can only lift the tasks whose strategies it has seen elsewhere, so a solved-count gain much larger than the coverage predicts would be suspicious, and a gain much smaller would point at a realization or tuning problem rather than at absence of transfer.

The ablation itself compares three configurations at a fixed seed and budget: the baseline with no library, a *macro-seeded* run that emits the top-$N$ macros as Phase A seeds, and a *macro-rate* run that also realizes macros during random-genome construction and mutation. Framing the comparison in terms of transfer rather than raw solved-count is what keeps it honest. The relevant question is not "did the count go up," since a single-seed count can move by the drift band for no reason, but "did the tasks the prior was predicted to reach actually get reached, and did they clear the noise floor."

**The measured result is a null.** On the ARC-AGI-2 training split the prior is worth +2 to +4 tasks in distribution, which is inside the drift band and therefore not a result. On the held-out evaluation split it moves the count from 0 to 1 of 120 against its own baseline, which is not comparable with the 2 of 120 reported earlier in this chapter because that figure comes from a different build. And the realization rate perturbs the mutation trajectory enough to churn some thirteen tasks per side, gained and lost, which is more than the signal it delivers. The library is therefore implemented, off by default, and carries no demonstrated held-out gain.

The null is worth stating precisely, because the transfer measurement above already established the premise. The strategies genuinely do recur across tasks, so the failure is not an absence of transfer; it sits in one of the two places the reachability analysis (Section 5.5) can distinguish. Either the realization rate is high enough that trajectory churn swamps the benefit, which the ±13-per-side figure directly supports, or the coupled realizations the prior favours are not the ones the exact-match frontier rewards on these tasks. The staged protocol was built so that a disappointing count would yield a diagnosis rather than a shrug, and this is that diagnosis: the mechanism is sound, the premise holds, and the delivery is currently too disruptive to pay for itself.

The seed-only variant has since been run and does not rescue the prior: injecting the top-eight macros of Table D.4 through the seed lane alone came in at −13, seven gained against twenty lost, and the near-anchored library was retired from

that lane. The churn is therefore not an artefact of the realization rate alone.

The two outcomes the design anticipated are therefore resolved in the second direction. Had the prior cleared the bar, the interpretation would have been that the search now teaches itself, and the library would have become a self-extending mechanism for growing the reachable program space without hand-adding operators. That remains the goal and the reachability gap says the headroom is real; what has not yet been shown is that the gap can be exploited without paying more in disruption than it returns.

# Chapter 7

# Future Work and Conclusions

This final chapter looks in two directions: forward, to the extensions the current design invites, and back, to what the work has and has not established. It notes on the way why a cheap, symbolic route to these benchmarks would matter beyond the leaderboard.

## 7.1 Future work on the static benchmark

The limits measured in Chapter 6 point at five extensions to the static solver:

- **Compositional parsing.** The parser can currently tune its parameters but not compose its detection passes into new ones (Section 4.2). Making the parser itself compositional, so that it builds new detectors out of existing ones, would let the representation reach tasks whose right abstraction is not among the built-in passes.

- **A single evolutionary loop.** The three-phase schedule (Section 5.1) is a scaffold; merging seeding, search, and refinement into one loop that evolves the full $(\mathcal{I}, \mathcal{P}, \mathcal{O})$ individual is the intended end state.

- **A stronger proposal distribution.** The evaluation locates one of the three limits in the operator set (Section 6.9); the record also shows single-operator additions becoming net-neutral through sampling dilution, so the lever is a stronger proposal distribution, of which the macro library of Section 5.5 is a first and so far unsuccessful attempt.

- **A learned discriminator.** The fitness problem (Section 5.2) invites Chollet's AlphaGo-style proposal [54, 53]: a model trained to judge which of two example-fitting programs is more likely to generalize. This is the one place a neural component would most obviously earn its keep in the static solver.

- **Higher-level program structure.** Evolving programs in a richer, structured representation rather than the current flat rule list would lift the ceiling on expressiveness; Byron, which builds Turing-complete test programs from a graph-structured genome, is one such design [45].

## 7.2 Beyond static puzzles

The solver as built maps one grid to another, so it stops where that format stops. An interactive setting, in which an agent acts and observes the consequence, breaks the framing: there is no single output to produce, only a sequence of actions to choose with feedback in between. What carries over is the representation. A graph of objects, attributes and relations describes a frame of a moving scene as readily as it describes a puzzle input, and the core-knowledge priors that motivated it, object persistence and naive physics above all, matter more where objects have to be tracked through time than where they sit still.

What changes is the job the graph does. In the static setting a program rewrites the graph once and the run ends. In an interactive setting the graph becomes a state to be predicted forward: what the scene will look like after an action. Three capabilities the present system lacks would be needed. The first is *temporal* structure, edges relating an object to its own earlier and later positions, so that motion is first-class rather than re-inferred frame by frame. The second is *action* conditioning, since a purely observational parser has no way to express how the scene depends on what the agent did. The third is *planning under an imperfect model*, which pushes towards the sampling-based lookahead reinforcement learning uses rather than the exact rollout a perfect model would allow. None of the three contradicts the design of this thesis; each is an addition to it. Whether they compose has not been tested, and nothing interactive was run here.

## 7.3 Broader impact

The unconstrained leaderboard is dominated by a few large companies able to spend heavily on compute, and the gap of nearly forty-five points to the resource-constrained competition (Section 2.5) is the measure of that advantage [14]. The scale of that spending is easy to underappreciate: Amazon, Microsoft, Meta and Google together are reported to have sustained capital expenditure in the

neighbourhood of US$400 billion across 2025 in connection with data-centre and AI development, on a public collation of their financial statements rather than on the filings themselves.[1] A symbolic route needs none of that infrastructure: concepts, symbols and a tractable search on hardware anyone can rent by the hour. That does not level the field, but it keeps progress available to groups who cannot buy compute at this scale, and it suits settings where a decision has to be inspected or where compute is scarce. On Harari's telling, that same capacity for concepts and symbols is what gave *Homo sapiens* its edge in the first place [57].

## 7.4 Conclusions

Despite the name, solving ARC-AGI-1 and 2 would not be solving artificial general intelligence, and this thesis claims neither. What it does claim is that the more modest, measurable target of ARC-AGI can be attacked directly, and that doing so is worthwhile in its own right. The route taken here is a graph-native co-evolutionary solver: a grid is parsed into a typed multigraph whose vocabulary mirrors core-knowledge priors, a short typed program rewrites that graph, a simple projector renders the result, and an evolutionary loop evolves all three together under a per-task budget, steered by a fitness that treats exact reproduction as a frontier to land on.

The prototype solves a third of the ARC-AGI-1 training set on the withheld test pair, the whole 400-task run taking under three hours on a single 30-core machine; the C++ reimplementation does better still, fitting 211 of the 391 tasks shared by the two editions and reproducing the withheld pair on 163 of them within two attempts. It scales with compute on the training side, though the test-side curve is already flattening. Across the 96 runs in the development log the union reaches 217 of the 400 training tasks at two attempts, over half that split, though no single configuration reaches it and the audit behind the record is uneven (Section 6.13). That is a floor under what the present rule language can reach rather than a ceiling, and the gap to any one run is a failure of choice more than of search: fitting programs are found reliably, but the fitness cannot always tell which of them will generalize. It also makes the ensembling of Section 6.10 the cheapest gain left on the table. The solutions it finds are short: the median winning program is a single rewrite rule, and nine in ten use at most two. Careful measurement separates three limits, on three different populations of task: what the language can express, what the search finds inside the budget, and what the fitness can tell apart among

[1] Estimated from the companies' released financial statements, as collated at `https://www.ben-evans.com/presentations`; a slide index rather than a primary filing, and part of the total is not data-centre spending.

programs that all fit. The first of the three is the domain-specific language rather than the parser. The same measurement pins down the stochastic noise floor that makes many apparent improvements illusory, which comparisons based on single runs cannot see. Mined role skeletons recur in roughly a third of held-out tasks with a multi-rule solution, which is the precondition for a search that teaches itself, but the prior built on them delivers no held-out gain and churns more tasks than it wins.

The limits are equally clear. On ARC-AGI-2 the same solver fits the train pairs of only 3 of the 120 public evaluation tasks and solves 2, only one of them new to ARC-AGI-2 (Section 6.6), which is close to nothing, and the reason is the one the ARC-AGI-1 diagnostics had already named: a benchmark built to demand multi-step compositional reasoning meets a system whose median winning program is one rule. It is the clearest statement of where this approach stops.

None of this is finished. The parser is not yet compositional, the search runs in three phases rather than one, the DSL has room to grow, and the fitness function remains the hardest open problem. But a symbol-based method still reaches a third of ARC-AGI-1 on hardware anyone can rent by the hour, though no like-for-like cost comparison was attempted and ARC-AGI-2 remains largely out of reach. What that route yields is not the score but the form of the answer: a short program that can be read and checked, rather than a set of weights.

# Appendix A

# Domain-Specific Language Reference

This appendix catalogues the operator inventory of the domain-specific language and the node and edge vocabulary of the parser, at the version evaluated in Chapter 6. The authoritative source is the solver's abstract-syntax definition; the counts here are a snapshot and are expected to grow.

## A.1 Node types

The parser emits seven node types: `SubObject` (a connected component or pixel group), `ClusterObject` (a group of related objects), `CavityObject` (an enclosed empty region), `ZoneNode` (a grid-line-delimited region), `PatternObject` (a detected repeating structure), `ImplicitObject` (an inferred object not directly present), and `NoiseObject` (a low-salience fragment). Nodes flagged `is_structural` (grid lines and the like) are excluded from the semantic edge passes.

## A.2 Edge families

The twenty-two edge types are grouped into five families:

- **Hierarchy:** `CONTAINS`, `IS_PART_OF`, `IS_INSIDE_ZONE`.
- **Spatial:** `TOUCHES`, `OVERLAPS`, and the four directional relations `IS_IMMEDIATELY_ABOVE`, `IS_IMMEDIATELY_BELOW`, `IS_IMMEDIATELY_LEFT_OF`, and `IS_IMMEDIATELY_RIGHT_OF`.
- **Alignment:** `SHARES_ROW`, `SHARES_COLUMN`, `DIAGONAL_TO`, `PROJECTS_RAY_TO`, `SAME_CENTROID_ROW`, `SAME_CENTROID_COLUMN`.

- **Physics:** `FALLS_TOWARDS`, `BRIDGED_TO`.
- **Logic:** `SAME_SHAPE`, `SAME_COLOR`, `IS_SYMMETRIC_TO`, `FORMS_REPEATING_PATTERN`, `COLLINEAR_TO`.

## A.3 The parser pipeline

The pipeline has seventeen numbered stages, three of which carry lettered sub-steps, for twenty-three operations in all, twenty-one of them gated. Most are controlled by a configuration flag, while background detection and connected-component extraction always run under a categorical parameter rather than a switch: (1) background detection, (2) zone detection, (3) connected-component extraction, (3b) object decomposition, (3c) multi-colour clustering, (3d) merge-by-containment, (4) cavity detection, (5) cluster detection, (5c) cluster colour lists, (5d) split-by-containment, (6) spatial edges, (7) noise scoring, (8) containment edges, (9) local coordinates and inside-zone edges, (9b) zone features, (10) z-index, (11) alignment edges, (12) ray edges, (13) gravity edges, (14) bridge edges, (15) logic edges, (16) pattern detection, (17) symmetry detection. The ordering guarantees that a step only reads structure produced by earlier steps. "Seventeen passes" elsewhere in this thesis refers to the numbered stages.

## A.4 Selectors

A `Selector` maps the current graph to a subset of its nodes. The thirty-four selector kinds, in the order they appear in the abstract-syntax source, are listed in Table A.1. Selectors split into three broad groups: filters that read a single node attribute (colour, mass, shape, boundary contact), extremal selectors that pick one node by position or size, and relational selectors that read the graph's edge structure or refer to the result of a second selector. Eighteen of the thirty-four are marked *safe* in the source, meaning they read only attributes that actions maintain in place, which lets the executor skip a scene rebuild between consecutive safe rules; that set is a performance detail rather than a semantic one, so it is not called out per row below.

**Table A.1:** The thirty-four selector kinds. Several take arguments (a colour, a mass threshold, an edge type, a reference selector); the argument is drawn per task by the codec's samplers at genome-construction time.

| Selector | Meaning |
|---|---|
| `ALL_SUBOBJECTS` | Every `SubObject` in the scene. |

| Selector | Meaning |
|---|---|
| `BY_COLOR` | Subobjects of a chosen colour. |
| `NON_BACKGROUND` | All non-background subobjects. |
| `BY_SHAPE` | Subobjects matching a chosen shape hash. |
| `BY_MASS_AT_LEAST` | Subobjects with pixel mass $\geq$ a threshold. |
| `BY_MASS_AT_MOST` | Subobjects with pixel mass $\leq$ a threshold. |
| `BY_MASS_EXACT` | Subobjects with a specified exact mass. |
| `BY_BOUNDARY_TOUCH` | Subobjects touching the grid boundary. |
| `BY_ZONE_POSITION` | Subobjects lying in a specified zone. |
| `MOST_FREQUENT_COLOR_OBJECTS` | Objects whose colour is the most frequent. |
| `LEAST_FREQUENT_COLOR_OBJECTS` | Objects whose colour is the least frequent. |
| `TOUCHING_COLOR` | Subobjects adjacent to a given colour. |
| `EDGE_TO_COLOR` | Subobjects with an edge to a given colour. |
| `HAS_EDGE_TYPE` | Subobjects incident to a given edge type. |
| `HAS_NO_EDGE_TYPE` | Subobjects with no edge of a given type. |
| `EDGE_COUNT_AT_LEAST` | Subobjects with at least $k$ incident edges. |
| `LARGEST_SUBOBJECT` | The single largest subobject. |
| `SMALLEST_SUBOBJECT` | The single smallest subobject. |
| `LEFTMOST_SUBOBJECT` | The leftmost subobject. |
| `RIGHTMOST_SUBOBJECT` | The rightmost subobject. |
| `TOPMOST_SUBOBJECT` | The topmost subobject. |
| `BOTTOMMOST_SUBOBJECT` | The bottommost subobject. |
| `TOP_LEFTMOST_SUBOBJECT` | The top-left-most subobject. |
| `TOP_LEFTMOST_NON_MOST_FREQUENT` | Top-left subobject, excluding the most-frequent colour. |
| `LEFTMOST_NON_MOST_FREQUENT` | Leftmost subobject, excluding the most-frequent colour. |
| `INSIDE_LARGEST_BBOX` | Subobjects inside the largest bounding box. |
| `SAME_SHAPE_AS_REFERENCE` | Subobjects whose shape matches a reference. |
| `REFERENCE_SINGLETONS` | Single-cell reference objects. |
| `BY_NOISE_SCORE` | Subobjects ranked by their noise/irregularity score. |
| `NEAREST_TO_SELECTOR` | The subobject nearest to another selector's result. |
| `FARTHEST_FROM_SELECTOR` | The subobject farthest from another selector's result. |
| `ADJACENT_TO_COLOR` | Subobjects adjacent to a specified colour. |
| `SELECTOR_SET_OP` | Union, intersection, or difference of two selectors. |
| `BY_HOLE_COUNT` | Objects with exactly $n$ enclosed background regions. |

## A.5 Actions

An `Action` rewrites the selected node set. The forty-four action kinds (Table A.2) fall into families: recolouring, deletion and translation, morphology (endpoint pruning, erosion, dilation, hollowing), geometry (bounding-box and cavity fills, cropping), and a thin higher-level layer that packages recurring composite moves such as D4-symmetry completion, ring stamping, and marker projection. Three of the actions (`SET_DIFFERENCE_KEEP`, `NORMALIZE_TO_ORIGIN`, and `SPLIT_AND_FOLD`) are excluded from uniform GA sampling, so as not to distort the mutation distribution over the older primitives; they are flagged with a dagger in the table. Five further actions were once excluded on the same grounds and have since been promoted into GA sampling, which is worth recording because the exclusion is a tuning decision rather than a property of the language. One caveat belongs with `SPLIT_AND_FOLD` in particular: the seed lane that used to reach it was retired, and the function that emitted its seeds now returns before doing so, so at the evaluated version the operator is reachable by neither path and is effectively dead.

**Table A.2:** The forty-four action kinds. A dagger (†) marks an action excluded from uniform GA sampling.

| Action | Effect |
|---|---|
| `RECOLOR` | Recolour the selection to a chosen colour. |
| `RECOLOR_TO_BACKGROUND` | Recolour the selection to the background colour. |
| `RECOLOR_TO_MOST_FREQUENT` | Recolour to the most frequent colour. |
| `RECOLOR_TO_LEAST_FREQUENT` | Recolour to the least frequent colour. |
| `RECOLOR_FROM_EDGE_MAJORITY` | Recolour from the majority colour on the boundary. |
| `RECOLOR_FROM_REFERENCE` | Recolour using a reference object's colour. |
| `RECOLOR_SEQUENCE` | Recolour the selection through an ordered colour list. |
| `SWAP_COLORS` | Swap two colours within the selection. |
| `PRUNE_ENDPOINTS` | Remove degree-one cells from the selection. |
| `SHRINK_TO_CORE` | Erode to a connectivity-preserving core. |
| `DILATE_SELECTED` | Grow the selection by one cell in all directions. |
| `HOLLOW` | Keep the boundary, clear the interior. |
| `FILL_BBOX` | Fill the bounding box of the selection. |
| `OUTLINE_OBJECT` | Replace each object with its outline. |
| `TRANSLATE_BY_OWN_EXTENT` | Translate by the object's own width or height. |

| Action | Effect |
|---|---|
| `CROP_TO_OBJECT` | Crop the grid to the selection's bounding box. |
| `PANEL_SPLIT_BOOLEAN` | Split a panel by a Boolean test over colours. |
| `SCALE_BY_COUNT` | Scale objects by a count-derived factor. |
| `COMPLETE_D4_SYMMETRY` | Complete the dihedral-4 symmetry group. |
| `PROJECT_INNER_QUAD_TO_OUTER_CORNERS` | Project an inner quadrant to the outer corners. |
| `PROJECT_SINGLETONS_TO_FULL_LINES` | Extend singleton cells into full rows/columns. |
| `STAMP_LOCAL_PATTERN` | Stamp a local pattern at each selected node. |
| `SYMMETRIZE_OBJECT` | Mirror-clean each selected figure against its own axis. |
| `SET_DIFFERENCE_KEEP` † | Keep the set difference of two selections. |
| `EXTEND_LINE_UNTIL` | Extend a line until it meets a given colour. |
| `PROJECT_MARKER` | Project marker objects to a target location. |
| `ATTACH_SAME_COLOR_SINGLETONS` | Attach singleton cells of the same colour. |
| `DELETE_REFERENCE_AFTER_TRANSFER` | Delete the reference once its pattern is copied. |
| `DELETE` | Delete the selection. |
| `TRANSLATE` | Translate the selection by an offset. |
| `TRANSLATE_WRAP` | Translate with wraparound at the grid edges. |
| `NORMALIZE_TO_ORIGIN` † | Move the selection to the grid origin. |
| `MIRROR_H` | Mirror the selection horizontally. |
| `MIRROR_V` | Mirror the selection vertically. |
| `COPY_TRANSLATE` | Copy the selection and translate the copy. |
| `KEEP_SELECTED` | Keep only the selection; clear the rest. |
| `KEEP_SELECTED_NORMALIZED` | Keep the selection, normalized to the origin. |
| `FILL_CAVITY` | Fill enclosed empty regions of the selection. |
| `FILL_BETWEEN_COLLINEAR_SAME_COLOR_PAIRS` | Fill the gap between aligned same-colour pairs. |
| `GRAVITY_TO_EDGE` | Slide every non-background cell toward a grid edge. |
| `FILL_ENCLOSED_BG` | Fill background fully enclosed by objects. |
| `SPLIT_AND_FOLD` † | Split along an axis and fold one half onto the other. |
| `FILL_DIAGONAL_PERIODIC` | Fill diagonals with a periodic pattern. |

| Action | Effect |
|---|---|
| TILE_FOLD_OR_NONBG | Tile and fold, favouring non-background cells. |

## A.6 Global transforms

A GlobalTransform is an optional whole-grid operation applied after the selector–action rewriting, for transformations that do not factor cleanly through per-object rules: rotations, reflections, tilings, and a few arithmetic resizings. The twenty-six transforms are listed in Table A.3. Where a transform acts depends on its kind. The mirror-concatenations, tilings, run deduplication, block reduction and the symmetry and periodic repairs act on the rendered grid, background and foreground alike. The geometric permutations act in scene space, on the foreground objects, whenever the program has rules, and on the whole canvas only on the empty-rule fast path.

**Table A.3:** The twenty-six global transforms.

| Transform | Effect |
|---|---|
| IDENTITY | Return the grid unchanged. |
| ROT90 | Rotate ninety degrees counter-clockwise. |
| ROT180 | Rotate one hundred eighty degrees. |
| ROT270 | Rotate ninety degrees clockwise. |
| FLIP_H | Reflect left–right. |
| FLIP_V | Reflect top–bottom. |
| TRANSPOSE | Reflect across the main diagonal. |
| ANTI_TRANSPOSE | Reflect across the anti-diagonal. |
| KRON_SELF_MASK | Kronecker product of the grid with itself as a mask. |
| KRON_MASK_ANCHOR_COLOR | Kronecker product masked by an anchor colour. |
| CROP_FOREGROUND | Crop to the bounding box of non-background cells. |
| SCALE2_NEAREST | Upscale 2× by nearest neighbour. |
| SCALE3_NEAREST | Upscale 3× by nearest neighbour. |
| FILL_PERIODIC_BACKGROUND | Fill the background by periodic tiling. |
| MIRROR_CONCAT_H | Concatenate the grid with its horizontal mirror. |
| MIRROR_CONCAT_H_REV | As above, mirror on the opposite side. |
| MIRROR_CONCAT_V | Concatenate the grid with its vertical mirror. |
| MIRROR_CONCAT_V_REV | As above, mirror on the opposite side. |
| MIRROR_QUAD_TL | $2 \times 2$ mirror tiling anchored top-left. |
| MIRROR_QUAD_TR | $2 \times 2$ mirror tiling anchored top-right. |

| Transform | Effect |
|---|---|
| MIRROR_QUAD_BL | $2 \times 2$ mirror tiling anchored bottom-left. |
| MIRROR_QUAD_BR | $2 \times 2$ mirror tiling anchored bottom-right. |
| TILE_H | Tile the grid twice horizontally. |
| TILE_V | Tile the grid twice vertically. |
| DEDUP_RUNS | Collapse consecutive identical rows and columns. |
| BLOCK_REDUCE2 | Downsample by $2\times$ block reduction. |

## A.7 Guards

A Guard is a cardinality precondition attached to a rule: the rule fires only if the number of selected nodes satisfies the guard. The three guard kinds are MIN_COUNT (fire if at least $k$ nodes are selected), MAX_COUNT (fire if at most $k$), and EXACT_COUNT (fire if exactly $k$). Guards let a rule express "recolour the objects only if there are exactly three of them," which is common in counting-flavoured tasks and which no selector–action pair can express on its own.

## A.8 Composition

A Program is a finite list of Rules followed by an optional GlobalTransform, and each Rule is a Selector, zero or more Guards, and one Action. The executor applies the rules in order, each rule selecting, checking its guards, and rewriting, and then applies the global transform, in scene space or to the rendered grid depending on its kind (Appendix A.6). The effective operator inventory on a given task is smaller than the full $34 \times 44$ selector–action product, because a selector that reads an edge type or node type absent from the current graph is inapplicable; the parser configuration therefore shapes the reachable program space as well as the representation.

# Appendix B

# Parser, Projector, and Distance Configuration

This appendix tabulates the two evolvable configurations, the parser's `GraphConfig` and the projector's `DecoderConfig`, together with the `DistanceConfig` of the parser audit, which is a fixed objective rather than an evolved one. Each field carries metadata (its kind, legal range or choice set, and default), and it is that metadata, rather than any hand-written mutation code, that makes a field reachable by the evolutionary search: adding a field to the dataclass and its metadata is the first of the two steps that bring a new behaviour under evolutionary control, the second being to add it to the mutable set.

## B.1 Parser parameters

The parser exposes forty-four configuration fields, grouped here by function. Twenty-one of them are evolved in the shipped configuration; the rest are frozen at their defaults to keep the search dimensionality down. Boolean toggles gate whole pipeline steps; categorical fields choose among detection modes; numeric fields set thresholds and tolerances. Where a toggle gates a specific pipeline step, the step number from Appendix A.3 is given.

## B.2 Projector policy axes

The projector rasterizes the output graph back into a grid. Because most semantic work has already happened in graph space, the projector is governed by only four categorical policy axes (Table B.4); three of the four are part of the evolved individual, paint order being frozen at `as-is`.

**Table B.1:** Object-extraction parameters: how pixels become `SubObjects`.

| Field | Kind | Default | Range / choices |
|---|---|---|---|
| `object_mode` | choice | `conn. comp.` | connected-component, pixel, colour-global |
| `color_mode` | choice | mono | mono, multi |
| `connectivity` | choice | 8 | 4, 8 |
| `background_mode` | choice | auto | auto, none, bg-as-object |
| `background_color_override` | int | none | $-1 \ldots 9$ |
| `min_object_size` | int | 0 | $0 \ldots 20$ |
| `merge_distance` | int | 0 | $0 \ldots 5$ |
| `noise_threshold` | float | 1.0 | $0.0 \ldots 1.0$ |
| `grid_line_threshold` | float | 1.0 | $0.5 \ldots 1.0$ |
| `noise_filter_order` | choice | size-first | size-first, noise-first |
| `multi_color_as_cluster` | bool | false | — |
| `cluster_decompose_two_color` | bool | false | — |
| `decompose_objects` | bool | false | — |
| `decomposition_mode` | choice | lines | lines, rectangles, both |
| `split_by_containment` | bool | false | — |
| `decompose_grid_lines` | bool | false | — |
| `merge_by_containment` | bool | false | — |

## B.3 Distance components

The parser audit of Section 5.4 compares a parsed graph against a hand-labelled reference under a weighted object-graph distance. That distance has ten components with independently tunable weights; setting a weight to zero disables that component, and the weights are normalized internally, so they need not sum to one. Table B.5 lists the components and their default weights. Four named presets are provided: *full* (all defaults), *nodes-only* (edge weights zeroed), *edges-only* (node property and node-matching weights zeroed), and *properties-only* (colour, shape and bbox emphasized, edge terms zeroed, with the node count and node type terms retained at half weight). The node-matching and edge-match terms carry the highest default weights because they are the most informative about whether the right objects and relations were produced.

**Table B.2:** Detection and edge toggles. The gated pipeline step (Appendix A.3) is given where applicable. All are booleans defaulting to *true*, except the touch connectivity, which selects four- or eight-neighbour adjacency and defaults to four.

| Field | Step | Gates |
|---|---|---|
| `detect_zones` | 2 | grid-line zone partitioning |
| `detect_cavities` | 4 | enclosed-region (hole) detection |
| `detect_clusters` | 5 | multi-object cluster grouping |
| `detect_patterns` | 16 | repeating-pattern nodes |
| `detect_symmetries` | 17 | symmetry-axis detection |
| `compute_spatial_edges` | 6 | `TOUCHES`/`OVERLAPS`/directional |
| `spatial_touch_connectivity` | 6 | 4- or 8-adjacency for `TOUCHES` |
| `compute_containment_edges` | 8 | `CONTAINS`/`IS_PART_OF` |
| `compute_alignment_edges` | 11 | shared-row/column, diagonal, collinear |
| `compute_ray_edges` | 12 | `PROJECTS_RAY_TO` |
| `compute_gravity_edges` | 13 | `FALLS_TOWARDS` |
| `compute_bridge_edges` | 14 | `BRIDGED_TO` |
| `compute_logic_edges` | 15 | same-shape, same-colour, symmetric |
| `compute_shape_classification` | — | node shape labels |
| `compute_shape_hash` | — | shape hashes for matching |
| `compute_local_coordinates` | 9 | zone-relative coordinates |
| `compute_z_index` | 10 | containment-derived layering |
| `compute_noise_scoring` | 7 | five-signal noise heuristic |
| `compute_zone_features` | 9b | per-zone summary features |

**Table B.3:** Numeric thresholds. The five `noise_weight_*` fields weight the signals of the noise heuristic (they need not sum to one).

| Field | Kind | Default | Range |
|---|---|---|---|
| `noise_weight_isolation` | float | 0.30 | 0.0 . . . 1.0 |
| `noise_weight_singleton_color` | float | 0.20 | 0.0 . . . 1.0 |
| `noise_weight_size_outlier` | float | 0.20 | 0.0 . . . 1.0 |
| `noise_weight_shape_outlier` | float | 0.15 | 0.0 . . . 1.0 |
| `noise_weight_position_outlier` | float | 0.15 | 0.0 . . . 1.0 |
| `alignment_tolerance` | float | 0.50 | 0.1 . . . 2.0 |
| `symmetry_tolerance` | float | 0.50 | 0.1 . . . 2.0 |
| `pattern_min_size` | int | 3 | 2 . . . 10 |

**Table B.4:** The four projector policy axes and their legal values.

| Axis | Default | Legal values |
|---|---|---|
| `background_policy` | scene-bg-or-zero | scene-bg-or-zero, zero, most-frequent-input, most-frequent-predicted, input-copy |
| `paint_order` | as-is | as-is, mass-desc, mass-asc, $y$-then-$x$, $z$-asc, $z$-desc |
| `overlap_policy` | overwrite | overwrite, keep-first |
| `postprocess` | none | none, crop-foreground |

**Table B.5:** The ten weighted components of the parser-audit graph distance.

| Component | Weight | Measures |
|---|---|---|
| node count | 1.0 | normalized difference in node count |
| node types | 1.0 | type-distribution distance (soft type-similarity) |
| node matching | 2.0 | greedy IoU matching of predicted to ground-truth nodes |
| colour match | 1.0 | fraction of matched pairs sharing colour |
| shape match | 1.0 | fraction of matched pairs sharing shape |
| bbox match | 0.5 | normalized bounding-box distance for matched pairs |
| extra match | 0.5 | fraction of ground-truth extra attributes reproduced |
| edge count | 1.0 | normalized difference in edge count |
| edge types | 1.0 | edge-type histogram distance |
| edge match | 1.5 | fraction of ground-truth edges reproduced under the node map |

# Appendix C

# Reproducibility

A run of the solver is fully specified by a handful of quantities. This appendix lists them and describes the configuration mechanism.

## C.1 What pins a run

Given the same code, a run is reproduced exactly by fixing: the random *seed*; the state of the `PYTHONHASHSEED` pin; the *worker count* (because work is partitioned across processes and the partition can affect tie-breaking); the *per-task wall-clock budget*; the population, generation, and restart counts; the dataset split; and the git commit of the solver. The wall-clock budget matters most: because the search is deadline-driven rather than generation-bounded, two machines of different speed given the same budget will evaluate different numbers of candidates and can resolve different task sets. A budget-bounded result is therefore reproducible only relative to a stated hardware profile, and the reported runs come from a single machine at seed 47, one task per core, with the effective core count derived per run rather than recorded (Section 6.1).

The determinism invariant (Section 5.3) is what makes the seed meaningful. Python's hash randomization, left unpinned, perturbs the iteration order of the sets and dictionaries on the decision path, and through that the order in which mutation draws are consumed, so the same seed produced different programs from run to run until `PYTHONHASHSEED` was pinned. Any reported single-seed comparison assumes the pin is in place; a run made without it should be treated as one sample from the noise band of Section 6.10, not as a point measurement.

## C.2 Configuration layering

Configuration is resolved from four layers, each overriding the one before: the dataclass defaults in the source; a configuration file (`arc_config.conf`); legacy environment variables; and command-line flags. This lets a default be changed in one place without touching source, while still allowing a single run to override any field from the command line. The macro-library controls (library path, realization rate, seed top-$N$) follow the same layering and all default to off, so a checkout with no library configured reproduces the pre-library behaviour byte-for-byte.

## C.3 The reported configurations

The three configurations of Table 6.1 share their population, generation and restart settings and differ only in the per-task wall-clock cap: population 40, 60 generations, and 5 restarts throughout, with a cap of at most 5 rules per program, and caps of 120, 480 and 960 seconds for the *short*, *medium* and *long* configurations respectively. All three run at seed 47, the shipped default, on a single machine with one task per core, at a worker count derived rather than recorded and rounding to thirty. They target training-side resolution (all train pairs bit-exact), deduplicate candidates by program hash within a generation, and use $\varepsilon$-lexicase parent selection at `fine2` case granularity with a tournament fallback. The Occam and view continuations are enabled, the view alternatives being transpose, left-right flip and up-down flip. The parser cache is on at 1 500 MiB per worker. The macro-library fields are off for all three (`macro_rate = 0.0`).

**Table C.1:** Provenance of the reported runs.

| Item | Value |
|---|---|
| commit | `1d4c06d` (branch `develop/luca`) |
| toolchain | Python 3.12.3, as recorded for the runs of Chapter 6 |
| dataset | ARC-AGI-1 training, 400 tasks, 3.255 train pairs per task on average |
| search | pop 40, 60 generations, 5 restarts, max 5 rules, seed 47 |
| selection | $\varepsilon$-lexicase, `fine2` case granularity |
| objective | `object_graph_distance`, target `train` only |
| continuations | Occam on; view on (transpose, fliplr, flipud) |
| parser cache | on, 1 500 MiB per worker |
| macros | off (`macro_rate = 0.0`) |
| workers | $\approx 30$, derived per run as $400 \times$ avg task time/wall |
| failures | 0 tasks failed in all three runs |

A fourth data point in Section 6.2 comes from an earlier build whose run summary records no commit, spanning 7 to 9 July 2026, run at 3 840 seconds per task at the same population, generation and restart settings and the same seed, over a 1 000-task superset of which 391 tasks overlap the 400 reported here. The ARC-AGI-2 held-out evaluation of Section 6.6 is on the same build as the ARC-AGI-1 table; the ARC-AGI-2 training row is from an older build and is reported with that caveat in place. The pin sets `PYTHONHASHSEED` to `0`, applied by the run driver, which re-executes the interpreter if the variable is unset.

## C.4 The development record

The signal-above-noise methodology of Section 6.10 is easier to judge against the record it was applied to. Table C.2 lists the representative changes from the roughly thirty-five numbered development phases, with the verdict each received.

**Table C.2:** The development record, classified against the signal-above-noise bar. "Kept" changes cleared the drift band on repeated measurement; "reverted" ones did not. Deltas are single-seed on the 400-task training set unless noted, and the bar is $\geq \pm 5$ (Section 6.10), so sub-bar deltas are read as drift regardless of sign. Fitness formulations are treated separately in Table 6.8.

| Change | Verdict | Effect and mechanism |
|---|---|---|
| Pin `PYTHONHASHSEED` | kept | removed a hidden variance source; made A/B valid |
| Object-graph distance fitness (no re-parse) | kept | +11 train; tripled GA depth at equal wall-clock |
| Dead-code removal + codec split | kept | +18% generations; throughput, not algorithm |
| Add whole-grid transforms (rotations, tilings, Kronecker) | kept | new exact solutions previously unreachable |
| Add `HOLLOW` / `FILL_BBOX` actions | kept | delivered by the GA with no seeding |
| `FILL_CAVITY` action at weight 1 | kept | +2 both; five clean cavity-family solves |
| `TILE_FOLD` delta-only detector | kept | +3 test, zero losses; narrow additive seed |
| Parameterized `STAMP`/`EXTEND` seeds | kept | +1 both; additive, trajectory-preserving |
| Runtime rule inference inside the action | kept | +2 both; keeps seed hashes bit-identical |
| Permutation-order bug fix in the codec | kept | a genuine +1, banked |
| Per-generation graph-edit-distance floor | reverted | −10 test; per-gen cost ate 24% of GA depth |
| Conditional / loop AST operators | reverted | −7 test; expressivity without matching search bias |
| Rule-freeze + mutation bandit | reverted | −13 train; freezing elites blocks parser walk |
| `GRAVITY_TO_EDGE` at weight 1 | reverted | −5 test; one real solve, renormalization flips eight |
| Tail-append two edge types to the codec | reverted | −2 test; shifted every sampling probability |
| Aggressive stagnation thresholds | reverted | +10 wins but −15 losses; right lever, overshot |
| Diverse post-search guess pool (pass@$k$) | reverted | pass@1 invariant; within-noise on the metric that counts |
| Same seed injected as a seed-arg change | reverted | −7 train; changed seed hash, reranked elites |

# Appendix D

# Example Programs and Operator Arguments

This appendix grounds the operator inventory of Appendix A in concrete material: a gallery of real example programs, a reference for the arguments the most-used operators take, and a table of the highest-support macros mined by the library-learning extension. Everything here is drawn from the solver's own seed enumerator, mutation codec, and mined libraries, not constructed for exposition.

## D.1 A gallery of programs

The clearest way to read the DSL is to see whole programs. Table D.1 lists example programs the seed enumerator constructs before the evolutionary search begins, the "obvious hypotheses" of Phase A (Section 5.1). Each is a short sequence of rules, written as `selector` →`action`, optionally followed by a global transform. They illustrate the range of the language in miniature: single-selector filters, two-rule compositions, and the higher-level stamp and projection actions.

**The same language, on real tasks.** The table above lists what the enumerator proposes; Figure D.1 shows six programs that actually won, each beside the withheld pair it reproduces. The six were picked to span the language rather than to flatter it: one task needs no rewrite rule at all and falls to a single global transform, three fall to one rule built on a topological, a physical, and a panel-level operator respectively, one needs a two-rule composition, and the last is the longest in this gallery at four rules; the parsimony table records a solved five-rule program. Reading a program against the grids it transforms is the quickest way to see what the operator names in Appendix A actually do. Each program in the figure was re-executed to produce the grid shown and checked against the target, so the figure does not rest on the

**Table D.1:** Example programs from the seed enumerator. Each rule is written `selector` →`action`; a two-rule program applies its rules in order. These are the canned hypotheses Phase A tries before the GP starts.

| Program | Targets |
|---|---|
| `LARGEST_SUBOBJECT` →`KEEP_SELECTED` | keep only the dominant object |
| `NON_BACKGROUND` →`RECOLOR`($c$) | flat recolour of the foreground |
| `LEAST_FREQUENT_COLOR_OBJECTS` →`KEEP_SELECTED` | isolate the rare-colour object(s) |
| `BY_MASS_EXACT(mass=1)` →`DELETE` | remove single-cell specks |
| `ALL_SUBOBJECTS` →`PRUNE_ENDPOINTS` | thin lines by one cell |
| `LARGEST_SUBOBJECT` →`KEEP_SELECTED`; `ALL_SUBOBJECTS` →`PRUNE_ENDPOINTS` | keep the big object, then thin it |
| `MOST_FREQUENT_COLOR_OBJECTS` →`RECOLOR_TO_LEAST_FREQUENT`; `ALL_SUBOBJECTS` →`DELETE_REFERENCE_AFTER_TRANSFER` | swap dominant colour, clean up |
| `BY_COLOR`($c$) →`STAMP_LOCAL_PATTERN(ring)` | stamp a ring on each coloured object |
| `SAME_SHAPE_AS_REFERENCE(largest)` →`KEEP_SELECTED` | keep everything shaped like the biggest |
| `(no rules)` + `ROT90` | the whole task is a rotation |
| `(no rules)` + `MIRROR_CONCAT_H` | mirror-tile the grid horizontally |

run record; none of the six uses a symmetry- or periodic-repair global, so none depends on that setting either.

**A resolved task, in full.** Task `31aa019c` (Section 6.3) is a useful worked example because it exercises a parser toggle, a colour-based selector, and a geometric action on a readable $10 \times 10$ grid. The program the search discovered is two rules and an identity global transform:

```
Rule 0:  LEAST_FREQUENT_COLOR_OBJECTS() →KEEP_SELECTED()
Rule 1:  ALL_SUBOBJECTS() →OUTLINE_OBJECT(
       connectivity=8, color=2)
Global:  IDENTITY
```

The first rule keeps the objects of the rarest colour and drops the rest; the second outlines every surviving sub-object under eight-connectivity, in a colour the program carries as a constant. Only the colour and connectivity are fixed: which objects are

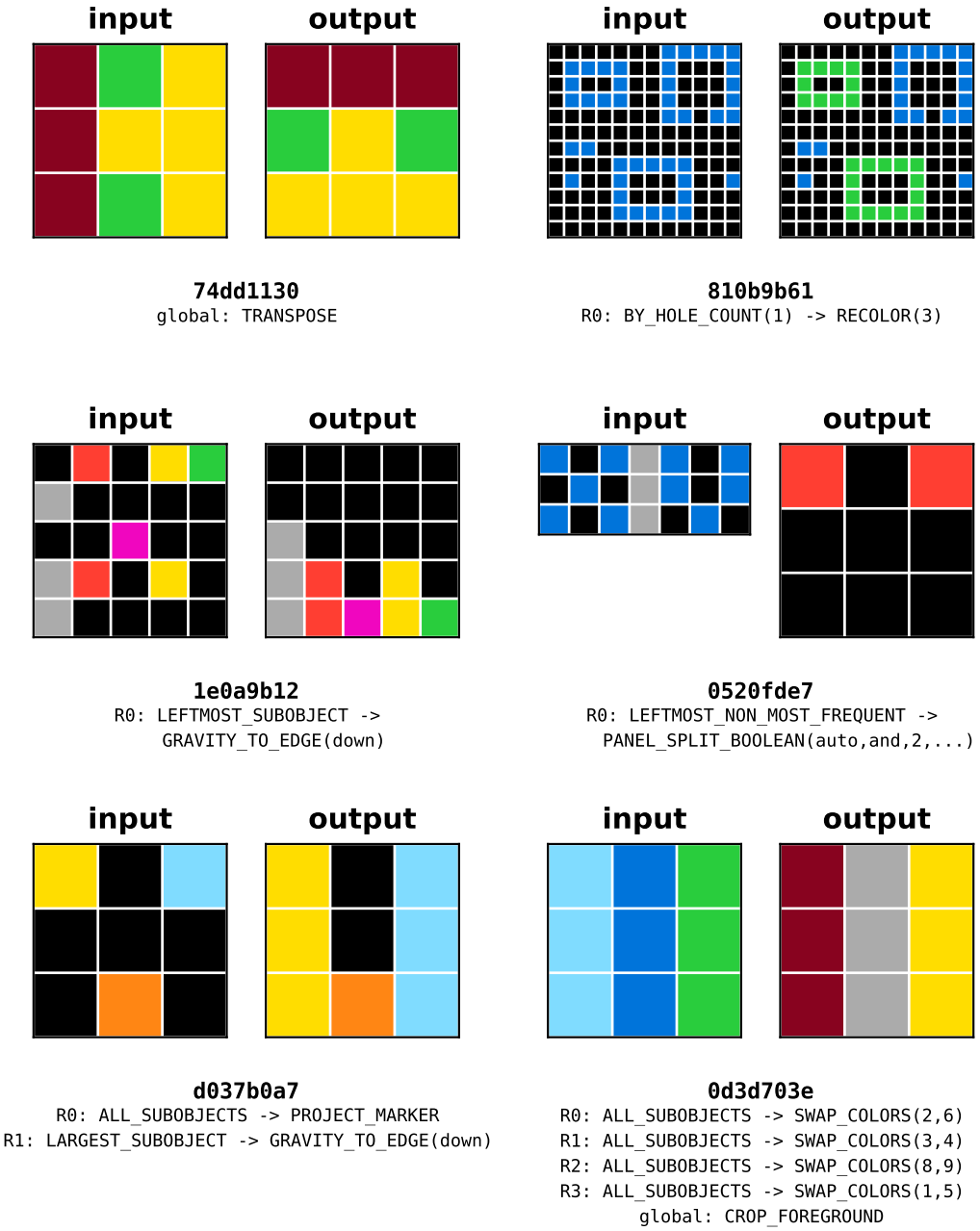



**Figure D.1:** Six solved ARC-AGI-1 training tasks with the programs that solve them, one panel each: the withheld test pair, input left and output right, under the task identifier and the winning program. Arguments are abbreviated to their values; Table D.2 and Table D.3 give the names and ranges.

kept is resolved against each grid, since the "rarest colour" differs from pair to pair, which is why the same two rules generalize across the task's train and test pairs.

## D.2 Operator arguments

Most selectors and actions take arguments, drawn per task by the codec's samplers at genome-construction time. Table D.2 and Table D.3 give the argument names and their legal ranges for the most frequently used operators; the full set follows the same pattern. Numeric ranges are inclusive. Colours are grid colours 0–9, and a colour argument is drawn from the task's output-colour prior most of the time and uniformly otherwise, so the search stays global while favouring plausible colours.

**Table D.2:** Arguments of representative selectors.

| Selector | Arguments |
|---|---|
| `BY_COLOR` | `color` ∈ [0,9] |
| `BY_MASS_AT_LEAST` | `mass` ∈ [1,9] |
| `BY_MASS_EXACT` | `mass` ∈ [1,9] |
| `BY_HOLE_COUNT` | `holes` ∈ {0,1,2,3,4} |
| `HAS_EDGE_TYPE` | `edge_type` (one of the sixteen edge types the codec samples, a subset of the twenty-two of Appendix A.2); `direction` ∈ {outgoing, incoming, both} |
| `NEAREST_TO_SELECTOR` | `distance` ∈ {centroid, bbox_gap}; `top_k` ∈ [1,4]; `include_references` ∈ {true, false} |
| `BY_NOISE_SCORE` | `min_score`, `max_score` ∈ [0,1]; `order` ∈ {none, highest, lowest}; `top_k` ∈ [1,4] |
| `SELECTOR_SET_OP` | `op` ∈ {intersection, difference, union}; nested left/right selector kinds and their arguments |

## D.3 The highest-support macros

Table D.4 lists the ten highest-support macros of the deployment library (116 macros mined from 335 solved tasks and 1 069 programs; Section 5.5). Each row gives the role skeleton, the dominant concrete head per slot, and the support, meaning the number of distinct tasks the skeleton was mined from. The list reads as a census of the recurring two-rule strategies in the solved set: paired colour-class moves, paired size-keyed recolourings, relational recolour chains, and projection pairs. Most of the top macros are role-level with no cross-slot binding; the coupled "same colour in both rules" macros, which the reachability analysis (Section 5.5)

**Table D.3:** Arguments of representative actions.

| Action | Arguments |
|---|---|
| `RECOLOR` | `color` $\in$ [0,9] (output-colour prior) |
| `TRANSLATE` | `dy`, `dx` $\in [-10,10]$ |
| `TRANSLATE_WRAP` | `dy`, `dx` $\in [-10,10]$, wrapping at the edges |
| `COPY_TRANSLATE` | `dy`, `dx` $\in [-10,10]$ |
| `DILATE_SELECTED` | `radius` $\in$ {1,2}; `connectivity` $\in$ {4,8}; `mask` $\in$ {auto, cross, diamond, square} |
| `RECOLOR_SEQUENCE` | `order` $\in$ {y_then_x, x_then_y, mass_desc, mass_asc}; `start_color` $\in$ [0,9]; `step` $\in$ {1,2,3} |
| `SWAP_COLORS` | `color_a`, `color_b` $\in$ [0,9], `color_a` $\neq$ `color_b` |
| `STAMP_LOCAL_PATTERN` | `pattern` $\in$ {dot, cross, x, ring, square}; `anchor` $\in$ {centroid, bbox_center, min_corner}; `color_source` $\in$ {self, constant, background}; `color` $\in$ [0,9] |
| `EXTEND_LINE_UNTIL` | `direction` (one of eleven); `stop_condition` $\in$ {boundary, touch_object, non-background}; `color`, `intersection_color` $\in$ [0,9] |
| `RECOLOR_FROM_REFERENCE` | `reference_selector_kind`; `color_policy` $\in$ {nearest, majority, only}; `match_policy` $\in$ {same_shape, same_shape_strict, any} |

identifies as the ones a plain sampler rarely reaches, appear a little lower in the ranking, the strongest carrying a colour binding at fraction 0.75.

**Table D.4:** The ten highest-support macros of the deployment library. "Head" is the top concrete `(selector, action)` pair for each slot; support is the number of distinct source tasks.

| Skeleton | Dominant heads (slot 1; slot 2) | Support |
|---|---|---|
| COLORCLASS, MOVE ×2 | `by_color`→`translate_wrap`; same | 18 |
| RELATIONAL, RECOLOR ×2 | `nearest_to_selector`→`recolor_sequence`; same | 17 |
| SIZE, RECOLOR ×2 | `by_mass_exact`→`recolor`; same | 17 |
| COLORCLASS, RECOLOR ×2 | `least_frequent_color_objects`→`recolor`; same | 15 |
| COLORCLASS, PROJECT ×2 | `by_color`→`stamp_local_pattern`; same | 14 |
| PICK1, RECOLOR; COLORCLASS, RECOLOR | `rightmost_subobject`→`recolor_sequence`; `most_frequent_color_objects`→`recolor` | 13 |
| PICK1, RECOLOR ×2 | `smallest_subobject`→`recolor`; `largest_subobject`→`recolor` | 12 |
| ALL, PROJECT ×2 | `all_subobjects`→`stamp_local_pattern`; same | 11 |
| COLORCLASS, RECOLOR; RELATIONAL, RECOLOR | `by_color`→`recolor`; `farthest_from_selector`→`recolor` | 11 |
| COLORCLASS, RECOLOR; PICK1, RECOLOR | `most_frequent_color_objects`→`recolor`; `smallest_subobject`→`recolor` | 9 |

# Appendix E

# Notation and Abbreviations

This appendix collects the symbols and abbreviations used throughout the thesis, for reference. Table E.1 lists the mathematical notation and Table E.2 the abbreviations.

**Table E.1:** Notation.

| Symbol | Meaning |
|---|---|
| $I$ | an individual, the triple $(\mathcal{I}, \mathcal{P}, \mathcal{O})$ |
| $\mathcal{I}$ | a parser configuration (a `GraphConfig`) |
| $\mathcal{P}$ | a program (a list of rules plus an optional global transform) |
| $\mathcal{O}$ | a projector configuration |
| $G_{\text{in}}$ | the input graph, the parsing of an input grid |
| $G_{\text{out}}$ | the output graph after the program rewrites $G_{\text{in}}$ |
| $(y, x)$ | the coordinate convention throughout: row then column |
| $O_t$ | a target (ground-truth) output grid |
| $\bar{O}$ | a predicted output grid |
| $c$ | a grid colour, an integer in $[0, 9]$ |
| $K$ | the elite count carried between generations |
| $N$ | the seed top-$N$ macro count |
| $\varepsilon$ | the tolerance band of $\varepsilon$-lexicase selection |

**Table E.2:** Abbreviations.

| Abbreviation | Expansion |
|---|---|
| AGI | artificial general intelligence |
| AI | artificial intelligence |
| ARC | Abstraction and Reasoning Corpus |
| ARC-AGI | the renamed ARC benchmark and its versioned successors |
| AST | abstract syntax tree |
| DSL | domain-specific language |
| GA | genetic algorithm |
| GP | genetic programming |
| IoU | intersection over union |
| LLM | large language model |
| LRM | large reasoning model |
| MDL | minimum description length |
| SyGuS | syntax-guided synthesis |
| TTT | test-time training |

# Bibliography


[1] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020. ISBN: 9780134610993. URL: http://aima.cs.berkeley.edu/ (cit. on p. 1).

[2] Melanie Mitchell. *Artificial intelligence: A guide for thinking humans*. Penguin UK, 2019 (cit. on p. 1).

[3] Ben Goertzel and Cassio Pennachin. *Artificial general intelligence*. Vol. 2. Springer, 2007 (cit. on p. 1).

[4] François Chollet. *On the Measure of Intelligence*. 2019. arXiv: 1911.01547 [cs.AI]. URL: https://arxiv.org/abs/1911.01547 (cit. on pp. 1, 2, 5, 9, 18).

[5] Francois Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. *ARC Prize 2024: Technical Report*. 2025. arXiv: 2412.04604 [cs.AI]. URL: https://arxiv.org/abs/2412.04604 (cit. on pp. 2, 9).

[6] Arseny Moskvichev, Victor Vikram Odouard, and Melanie Mitchell. *The ConceptARC Benchmark: Evaluating Understanding and Generalization in the ARC Domain*. 2023. arXiv: 2305.07141 [cs.LG]. URL: https://arxiv.org/abs/2305.07141 (cit. on pp. 2, 10, 59).

[7] Susan Carey and Elizabeth Spelke. «Science and Core Knowledge». In: *Philosophy of Science* 63.4 (Dec. 1996), pp. 515–533. ISSN: 1539-767X. DOI: 10.1086/289971. URL: http://dx.doi.org/10.1086/289971 (cit. on pp. 3, 18).

[8] Elizabeth S. Spelke and Katherine D. Kinzler. «Core knowledge». In: *Developmental Science* 10.1 (Jan. 2007), pp. 89–96. ISSN: 1467-7687. DOI: 10.1111/j.1467-7687.2007.00569.x. URL: http://dx.doi.org/10.1111/j.1467-7687.2007.00569.x (cit. on pp. 3, 6, 18).

[9] John R. Koza. «Hierarchical genetic algorithms operating on populations of computer programs.» In: *IJCAI*. Vol. 89. 1989, pp. 768–774 (cit. on pp. 3, 21).

[10] Luca Favole, Guido Meak, Giorgio Mongardi, Giovanni Squillero, and Alberto Tonda. «GP and the Quest for AGI: When Goodhart's Law Turned Out to Be Useful». In: *Genetic Programming Theory and Practice*. Submitted. Springer, 2026 (cit. on p. 4).

[11] ARC Prize Foundation. *ARC-AGI-3: A New Challenge for Frontier Agentic Intelligence*. 2026. arXiv: 2603.24621 [cs.AI]. URL: https://arxiv.org/abs/2603.24621 (cit. on p. 10).

[12] Solim LeGris, Wai Keen Vong, Brenden M. Lake, and Todd M. Gureckis. *H-ARC: A Robust Estimate of Human Performance on the Abstraction and Reasoning Corpus Benchmark*. 2024. arXiv: 2409.01374 [cs.AI]. URL: https://arxiv.org/abs/2409.01374 (cit. on p. 11).

[13] Solim LeGris, Wai Keen Vong, Brenden M. Lake, and Todd M. Gureckis. «A Comprehensive Behavioral Dataset for the Abstraction and Reasoning Corpus». In: *Scientific Data* 12.1 (Aug. 2025). ISSN: 2052-4463. DOI: 10.1038/s41597-025-05687-1. URL: http://dx.doi.org/10.1038/s41597-025-05687-1 (cit. on p. 11).

[14] Sahar Vahdati, Andrei Aioanei, Haridhra Suresh, and Jens Lehmann. *The ARC of Progress towards AGI: A Living Survey of Abstraction and Reasoning*. Living survey, revision of 17 March 2026. 2026. DOI: 10.48550/arXiv.2603.13372. arXiv: 2603.13372 [cs.AI]. URL: https://arxiv.org/abs/2603.13372 (cit. on pp. 11–13, 70).

[15] Jasmin Thelen and Oliver Wilhelm. *Bringing Back Rule Induction to Fluid Intelligence Research? An Initial Validation of the ARC-AGI Benchmark in Humans*. 2026. arXiv: 2607.11263 [cs.AI]. URL: https://arxiv.org/abs/2607.11263 (cit. on p. 11).

[16] Aske Plaat, Annie Wong, Suzan Verberne, Joost Broekens, Niki van Stein, and Thomas Back. *Multi-Step Reasoning with Large Language Models, a Survey*. 2025. arXiv: 2407.11511 [cs.AI]. URL: https://arxiv.org/abs/2407.11511 (cit. on p. 12).

[17] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. «Chain-of-Thought Prompting Elicits Reasoning in Large Language Models». In: *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. Ed. by Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh. 2022. URL: http://papers.nips.cc/paper%5C_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html (cit. on p. 12).

[18] Gary Marcus. *o3 AGI: The Art of the Demo and What It Means*. https://garymarcus.substack.com/p/o3-agi-the-art-of-the-demo-and-what. Accessed: 2025-10-26. 2025 (cit. on p. 12).

[19] Daniel Franzen, Jan Disselhoff, and David Hartmann. «Product of Experts for LLMs: Boosting Performance on ARC is a Matter of Perspective». In: *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, Canada*. OpenReview.net, 2025. URL: https://openreview.net/forum?id=dsBjxI6l8W (cit. on pp. 13, 23).

[20] Ekin Akyürek, Mehul Damani, Adam Zweiger, Linlu Qiu, Han Guo, Jyothish Pari, Yoon Kim, and Jacob Andreas. «The Surprising Effectiveness of Test-Time Training for Few-Shot Learning». In: *Proceedings of the 42nd International Conference on Machine Learning*. Ed. by Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu. Vol. 267. Proceedings of Machine Learning Research. PMLR, July 2025, pp. 942–963. URL: https://proceedings.mlr.press/v267/akyurek25a.html (cit. on pp. 13, 14, 23).

[21] Wen-Ding Li et al. «Combining Induction and Transduction for Abstract Reasoning». In: *The Thirteenth International Conference on Learning Representations*. 2025. URL: https://openreview.net/forum?id=UmdotAAVDe (cit. on p. 13).

[22] Johan Socrates Wind. *1st place solution + code and official documentation*. 2020. URL: https://www.kaggle.com/competitions/abstraction-and-reasoning-challenge/writeups/icecuber-1st-place-solution-code-and-official-docu (cit. on pp. 14, 16, 23, 58).

[23] Yudong Xu, Elias B. Khalil, and Scott Sanner. «Graphs, Constraints, and Search for the Abstraction and Reasoning Corpus». In: *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*. Ed. by Brian Williams, Yiling Chen, and Jennifer Neville. AAAI Press, 2023, pp. 4115–4122. DOI: 10.1609/AAAI.V37I4.25527. URL: https://doi.org/10.1609/aaai.v37i4.25527 (cit. on pp. 14, 17, 23, 58).

[24] Chao Lei, Nir Lipovetzky, and Krista A. Ehinger. «Generalized Planning for the Abstraction and Reasoning Corpus». In: *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Vancouver, Canada, February 20-27, 2024*. AAAI Press, 2024. arXiv: 2401.07426 [cs.AI] (cit. on pp. 14, 58).

[25] Sébastien Ferré. «Tackling the Abstraction and Reasoning Corpus (ARC) with Object-Centric Models and the MDL Principle». In: *Advances in Intelligent Data Analysis XXII - 22nd International Symposium on Intelligent Data Analysis, IDA 2024, Stockholm, Sweden, April 24-26, 2024, Proceedings, Part I*. Ed. by Ioanna Miliou, Nico Piatkowski, and Panagiotis Papapetrou. Vol. 14641. Lecture Notes in Computer Science. Springer, 2024, pp. 3–15. DOI: 10.1007/978-3-031-58547-0_1. URL: https://doi.org/10.1007/978-3-031-58547-0%5C_1 (cit. on pp. 14, 23).

[26] Jakub Bednarek and Krzysztof Krawiec. «Solving Abstract Reasoning Problems With Neurosymbolic Program Synthesis and Task Generation». In: *Neurosymbolic Artificial Intelligence* 2 (Jan. 2026). ISSN: 2949-8732. DOI: 10.1177/29498732261419309. URL: http://dx.doi.org/10.1177/29498732261419309 (cit. on p. 14).

[27] Richard S. Sutton. *The Bitter Lesson*. Accessed: 2025-10-19. 2019. URL: http://www.incompleteideas.net/IncIdeas/BitterLesson.html (cit. on p. 16).

[28] Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. *The Illusion of Thinking: Understanding the Strengths and Limitations of Reasoning Models via the Lens of Problem Complexity*. 2025. arXiv: 2506.06941 [cs.AI]. URL: https://arxiv.org/abs/2506.06941 (cit. on p. 16).

[29] Junyan Li, Wenshuo Zhao, Yang Zhang, and Chuang Gan. «Steering LLM Thinking with Budget Guidance». In: *CoRR* abs/2506.13752 (2025). DOI: 10.48550/ARXIV.2506.13752. arXiv: 2506.13752. URL: https://doi.org/10.48550/arXiv.2506.13752 (cit. on p. 16).

[30] Michael Hodel. *arc-dsl: a domain-specific language for ARC-AGI*. https://github.com/michaelhodel/arc-dsl. The same author's procedural example generator is described in arXiv:2404.07353. 2024 (cit. on pp. 16, 23).

[31] Elizabeth S Spelke. *What babies know*. en. Oxford Series in Cognitive Development. New York, NY: Oxford University Press, Oct. 2022 (cit. on p. 18).

[32] Patrick J. Hayes. «A Naive Physics Manifesto». In: *Expert Systems in the Micro-electronic Age*. Ed. by Donald Michie. Edinburgh University Press, 1979 (cit. on p. 18).

[33] Noam Chomsky. *The minimalist program*. MIT press, 2014 (cit. on p. 19).

[34] Noah D. Goodman, Joshua B. Tenenbaum, and Tobias Gerstenberg. «Concepts in a Probabilistic Language of Thought». In: *The Conceptual Mind: New Directions in the Study of Concepts*. Ed. by Eric Margolis and Stephen Laurence. MIT Press, 2015 (cit. on p. 19).

[35] Mathias Sablé-Meyer, Kevin Ellis, Josh Tenenbaum, and Stanislas Dehaene. «A Language of Thought for the Mental Representation of Geometric Shapes». In: *Cognitive Psychology* 139 (Dec. 2022), p. 101527. ISSN: 00100285. DOI: 10.1016/j.cogpsych.2022.101527. (Visited on 09/10/2024) (cit. on p. 19).

[36] Zohar Manna and Richard Waldinger. «A Deductive Approach to Program Synthesis». In: *ACM Trans. Program. Lang. Syst.* 2.1 (Jan. 1980), pp. 90–121. ISSN: 0164-0925. DOI: 10.1145/357084.357090. URL: https://doi.org/10.1145/357084.357090 (cit. on p. 19).

[37] Sumit Gulwani. «Automating string processing in spreadsheets using input-output examples». In: *SIGPLAN Not.* 46.1 (Jan. 2011), pp. 317–330. ISSN: 0362-1340. DOI: 10.1145/1925844.1926423. URL: https://doi.org/10.1145/1925844.1926423 (cit. on p. 20).

[38] Rajeev Alur et al. «Syntax-Guided Synthesis». In: *Formal Methods in Computer-Aided Design (FMCAD)*. 2013, pp. 1–8 (cit. on p. 20).

[39] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. «Combinatorial sketching for finite programs». In: *SIGARCH Comput. Archit. News* 34.5 (Oct. 2006), pp. 404–415. ISSN: 0163-5964. DOI: 10.1145/1168919.1168907. URL: https://doi.org/10.1145/1168919.1168907 (cit. on p. 20).

[40] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke B. Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. «DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning». In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (2021). URL: https://api.semanticscholar.org/CorpusID:235474001 (cit. on p. 21).

[41] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. «Top-Down Synthesis for Library Learning». In: *Proceedings of the ACM on Programming Languages* 7.POPL (2023). Stitch (cit. on p. 21).

[42] John Koza. *Genetic Programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Tech. rep. STAN-CS-90-1314. Dept. of Computer Science, Stanford University, 1990 (cit. on p. 21).

[43] David J. Montana. «Strongly Typed Genetic Programming». In: *Evolutionary Computation* 3.2 (1995), pp. 199–230 (cit. on p. 21).

[44] Peter A. Whigham. «Grammatically-based Genetic Programming». In: *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*. 1995, pp. 33–41 (cit. on p. 21).

[45] Giovanni Squillero, Alberto Tonda, Dimitri Masetta, and Marco Sacchet. «Byron: A Fuzzer for Turing-complete Test Programs». In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '24 Companion. ACM, July 2024, pp. 1691–1694. DOI: 10.1145/3638530.3664136. URL: http://dx.doi.org/10.1145/3638530.3664136 (cit. on pp. 21, 70).

[46] Giovanni Squillero. «Microgp—an evolutionary assembly program generator». In: *Genetic programming and evolvable machines* 6.3 (2005), pp. 247–263 (cit. on p. 21).

[47] Lee Spector. «Assessment of Problem Modality by Differential Performance of Lexicase Selection in Genetic Programming: A Preliminary Report». In: *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO Companion '12)*. ACM, 2012 (cit. on pp. 22, 36).

[48] Thomas Helmuth, Lee Spector, and James Matheson. «Solving Uncompromising Problems with Lexicase Selection». In: *IEEE Transactions on Evolutionary Computation* 19.5 (2015), pp. 630–643 (cit. on pp. 22, 36).

[49] William La Cava, Lee Spector, and Kourosh Danai. «Epsilon-Lexicase Selection for Regression». In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '16)*. ACM, 2016. arXiv: 1905.13266 [cs.NE]. URL: https://arxiv.org/abs/1905.13266 (cit. on pp. 22, 36).

[50] Ting Hu, Wolfgang Banzhaf, and Gabriela Ochoa. «How Neutrality Shapes Evolution: Simplicity Bias and Search». In: *Proceedings of the Genetic and Evolutionary Computation Conference*. New York, NY, USA: Association for Computing Machinery, 2025, pp. 1008–1016. ISBN: 9798400714658. URL: https://doi.org/10.1145/3712256.3726454 (cit. on p. 22).

[51] Brandon C. Colelough and William Regli. *Neuro-Symbolic AI in 2024: A Systematic Review*. 2025. arXiv: 2501.05435 [cs.AI]. URL: https://arxiv.org/abs/2501.05435 (cit. on p. 22).

[52] Henry A. Kautz. «The Third AI Summer: AAAI Robert S. Engelmore Memorial Lecture». In: *AI Magazine* 43.1 (Mar. 2022), pp. 105–125. ISSN: 23719621. DOI: 10.1002/aaai.12036 (cit. on p. 22).

[53] François Chollet. «Improving program search with deep learning». ARC 2024 Hackathon. Lab42, Villa Fontana, Obere Strasse 22B, 7270 Davos, Switzerland, 2024. URL: https://lab42.global/updates/arcathon-2024/ (cit. on pp. 23, 40, 70).

[54] David Silver et al. «Mastering the Game of Go with Deep Neural Networks and Tree Search». In: *Nature* 529.7587 (2016), pp. 484–489. ISSN: 14764687. DOI: 10.1038/nature16961 (cit. on pp. 23, 70).

[55] Daniel C Dennett. *From bacteria to Bach and back*. en. New York, NY: WW Norton, Feb. 2017 (cit. on p. 38).

[56] François Chollet. «Abstraction & Reasoning: What Deep Learning can do, what it can't, and what we can try next». AAAI Fall Symposium. 2020. URL: `https://fchollet.com/%5C#talks` (cit. on p. 40).

[57] Yuval Noah Harari. *Sapiens: A Brief History of Humankind*. New York: Harper, 2015. ISBN: 9780062316097 (cit. on p. 71).