Seeing the Gaps: Improving Object Segmentation for Abstract Visual Reasoning in Julia

Grammar Extensions and Structural Ranking for the BEN Agent on the ARC Benchmark

Franciszek Howard

Responsible Professor: Dr. Sebastijan Dumančić
Supervisor: Dekel Zak

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Franciszek Howard
Final project course: CSE3000 Research Project
Thesis committee: Dr. Sebastijan Dumančić, Dekel Zak, Dr. Arie van Deursen

Abstract

The Abstraction and Reasoning Corpus (ARC) is a benchmark designed to measure general-purpose skill acquisition, requiring solvers to infer transformation rules from very few examples. Program synthesis approaches such as the Divide, Align and Conquer (DA&C) framework have shown promise, but their segmentation stage, which decomposes input grids into objects, remains a bottleneck in both computational cost and task coverage. This work presents a reimplementation of the BEN agent in Julia, integrated with the Herb.jl program synthesis ecosystem, alongside two targeted improvements to the segmentation module. First, we extend the segmentation grammar with a `proximity;N` mode that groups same-color pixels within Chebyshev distance N [1], enabling correct decomposition of objects with small internal gaps. Second, we replace the original full-pipeline mode selection with a lightweight structural ranking that scores all candidate modes on their segmentation output alone, using all training examples rather than only the first. Evaluated on the 400-task ARC training set with a 140-second budget, the Julia reimplementation solves 46 tasks, of which 5 are solved via proximity modes absent from the original grammar and are therefore likely attributable to the grammar extension. Analysis of the ranking reveals that the top-ranked mode solves 59% of solvable tasks. A deliberate fallback mechanism compensates for the heuristic's imperfection by guaranteeing that a reliable base mode is always attempted second. Grammar extensions account for some improvement (5 out of 13 tasks likely solved exclusively by Julia).

1 Introduction

Artificial Intelligence systems have made remarkable progress in recent years, transforming many areas of science, from natural language processing and image recognition to healthcare applications such as detecting early-stage cancers. Large language models can generate text, translate between languages, and write code, tasks that once seemed uniquely human. However, their performance drops sharply when faced with problems that require some form of abstract reasoning from principles rather than pattern matching over large training datasets. Their ability to adapt to novel situations and generalize with respect to underlying rules still lags behind human performance. This gap motivates the study of benchmarks specifically designed to measure generalization from limited examples.

Chollet [2] argued in favor of emphasizing breadth of generalization in acquiring skills, rather than raw skill, as a focus of a definition of intelligence. Based on this, Chollet introduced the Abstraction and Reasoning Corpus (ARC), a benchmark designed to measure general intelligence of a system or a human through skill-acquisition leveraging only the priors common to human cognition, referred to as Core Knowledge. Each task in ARC consists of a small number of demonstration examples (~ 3), and typically one test example. Each example consists of an input and output grid. The test-taker must reconstruct "from scratch" the output grid of the test example given only the demonstration examples.

Bratus et al. [3] discuss the taxonomy of the ARC research landscape, showing multiple different approaches and methods for solving the ARC problem. A promising direction is program synthesis [4], where the goal is to infer a program that explains observed input-output pairs.

Recent work has demonstrated that program synthesis techniques can effectively tackle ARC tasks by searching over a space of candidate programs guided by priors. In particular, the Divide, Align, and Conquer (DA&C) [5] approach introduces a structured pipeline that decomposes images into components, aligns corresponding objects between inputs and outputs, and learns transformations at the object level. Its implementation in the BEN agent in Python has shown relatively good performance with very limited cost per task compared to using LLMs which can cost up to $167 per task at high efficiency [6]. However, the object segmentation stage of BEN remains an open challenge, with room to improve both the range of object structures it can express and the efficiency with which a suitable decomposition is selected.

BEN's architecture comprises three tightly coupled stages. First, a **segmentation module** partitions each ARC grid into discrete objects using a grammar of pixel-expansion modes and selects the most suitable mode by scoring candidate decompositions using a decomposition function. Second, an **alignment module** based on the Structure-Mapping Engine (SME) [7] identifies structurally meaningful correspondences between input and output objects, steering synthesis towards component pairs that require minimal transformation programs. Third, a **conquer module** synthesises a transformation program for each aligned object pair using a domain grammar of geometric primitives, together with a Boolean concept definition specifying when the transformation applies. Each stage depends directly on the quality of the previous: errors in segmentation propagate into misalignment, which in turn enlarges the search space for synthesis. Improving segmentation therefore has a multiplicative effect on the pipeline's overall task-solving rate.

This work focuses on improving the object detection stage within the DA&C framework. Specifically, the research investigates how decomposition of ARC grids into meaningful components can be made more computationally efficient and more broadly applicable. Addressing this question is critical, as inefficiencies and incorrectness in the segmentation function propagate throughout the pipeline and decrease overall performance in solving ARC.

The **main contribution** of this project is a reimplementation of the BEN approach in Julia, designed to integrate with the Herb.jl program synthesis ecosystem [8], alongside specific optimizations of the segmentation module. These contributions improve BEN in two concrete ways: (i) Julia's compiled execution model offers the potential to reduce the per-task time of the segmentation phase, which is the computational bottleneck for larger ARC grids; and (ii) an extension to the segmentation grammar and a principled mode-ranking

procedure are introduced, enabling the system to correctly decompose object structures that the original grammar cannot express, and to select a suitable decomposition without running the full synthesis pipeline for every candidate mode.

The central **research question** motivating this work is: *Can the BEN agent's segmentation stage be faithfully reimplemented in Julia and improved to increase both the range of addressable tasks and the efficiency of mode selection?* The paper first introduces the relevant background and methodology, then details the implementation and proposed improvements, followed by an experimental evaluation demonstrating their impact. Finally, the results are discussed in the context of broader challenges in abstract reasoning systems.

2 Background

2.1 The Abstraction and Reasoning Corpus

As introduced in Section 1, the Abstraction and Reasoning Corpus (ARC) [2] [1]is designed to measure general-purpose skill-acquisition efficiency rather than raw performance on specific tasks. ARC is grounded in Core Knowledge priors, the cognitive primitives shared by all humans, covering objectness and basic physics, elementary goal-directedness, natural numbers and counting, and basic geometry and topology. Tasks require none of the accumulated knowledge gained through education, ensuring that humans and AI systems begin from an equivalent foundation.

2.1.1 Task Specification

Each ARC task presents on average 3 demonstration input-output grid pairs, followed by a test input for which the solver must construct the correct output from scratch. Grids use a palette of 10 colors, with dimensions up to 30×30. Success is binary: the produced grid must exactly match the ground truth, with up to 3 attempts per test example. The benchmark comprises 400 training tasks and 600 evaluation tasks, with training and evaluation sets fully disjoint. Typical humans solve the majority of tasks on their first attempt without practice, while standard machine learning approaches including deep learning have made very limited headway, as the tasks demand genuine abstraction rather than pattern matching over a dense training distribution.

2.1.2 Approaches to ARC

Bratus et al. [3] survey the landscape of methods applied to ARC, organizing them into two broad families. Inductive methods explicitly infer a transformation rule from the demonstration pairs, typically as a program in a domain-specific or general-purpose language, and then apply it to the test input. Transductive methods instead map directly from the support examples and test input to the output, without constructing an explicit rule. The former tend to generalize better within a task and produce interpretable solutions, but face combinatorial search challenges; the latter leverage the pattern-recognition capacity of neural networks but can struggle with compositional or systematic transformations. Despite significant progress, ARC remains difficult: even large language models require substantial test-time compute to approach human-level performance [9], underscoring that genuine abstract reasoning remains an open problem. This work follows the inductive program synthesis direction, building on the Divide, Align and Conquer (DA&C) framework [5], which we introduce in the following section.

[1]https://github.com/fchollet/ARC-AGI

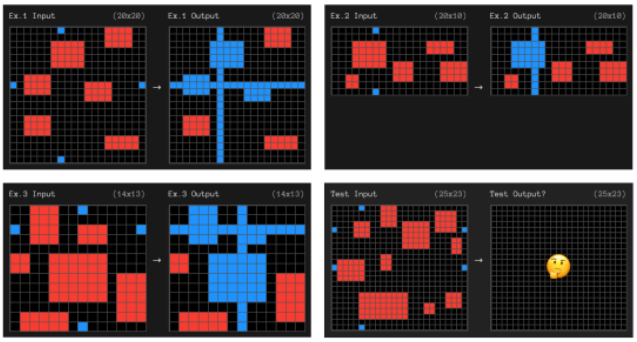

Figure 1: An example ARC task (Difficult). Demonstration pairs reveal an implicit transformation rule; the solver must produce the output for the test input (bottom right, marked with a face).

2.2 Divide Align and Conquer

Divide, Align & Conquer (DA&C) is a program synthesis strategy introduced by Witt et al. [5] that addresses the core bottleneck of search-based synthesis: as target programs grow larger, the search space expands exponentially in both program depth and grammar branching factor. Standard divide-and-conquer approaches mitigate this by partitioning the set of training *examples* into subsets, each defining an independent synthesis task. However, this only marginally simplifies problems where the difficulty lies not in covering multiple examples but in reasoning about the structure *within* a single example. DA&C extends the divide-and-conquer idea to the level of components within individual examples, decomposing a single deep program search into a linear number of much shallower synthesis tasks.

The framework operates in three interdependent phases. **Divide** synthesises a decomposition function δ that segments each input/output example into a set of discrete, mutually exclusive components. For abstract visual reasoning tasks, this corresponds to identifying individual objects within a scene, for instance, grouping equally coloured, directly neighbouring pixels into objects against a background. The decomposition function is itself learned as part of the synthesis process and ranked according to its estimated usefulness for downstream program search, rather than being fixed in advance.

Align then establishes correspondences between the components of the segmented input and output using analogical reasoning, specifically the Structure-Mapping Engine (SME) [7], a computational implementation of Structure-Mapping Theory (SMT) [10]. SME searches for a structural alignment between input and output scenes that maximises their shared relational structure, producing a ranked list of input/output component pairings. Correspondences that contribute most to a deep, structurally consistent alignment are explored first, steering synthesis towards component pairs that require min-

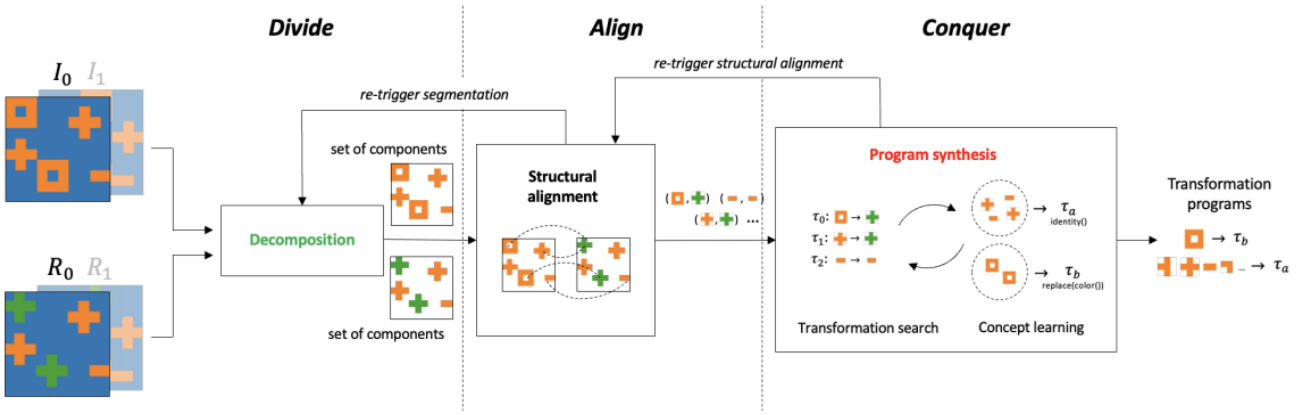

Figure 2: The full DA&C pipeline, illustrating the Divide, Align, and Conquer stages on an example ARC task.

imal transformation programs and avoiding a combinatorial explosion over all possible pairings.

Conquer treats each aligned component pair as an independent synthesis sub-task. For each pair, it learns a transformation program $p'(o_i) = o_j$ using a domain grammar of geometric primitives, together with a concept definition $C_{p'}$ expressed as a Boolean formula over component attributes that specifies the conditions under which the transformation applies. Crucially, information available within an aligned output component is used to prune irrelevant programs during search. For example, any primitive that produces an incorrect colour can be eliminated immediately. These transformation rules are then combined into a global solution program that, when applied to a decomposed test input, reconstructs the correct output.

The agent implementing DA&C is called BEN. On the ARC benchmark, BEN solved 22.5% of the 400 training tasks using only 11 generic geometric primitives, while generating on average just 0.2% as many candidate programs as the top Kaggle competition agent. When the Kaggle agent's primitive library is restricted to those semantically equivalent to BEN's, its performance drops from 47% to below 25%, suggesting that much of the apparent gap between agents reflects differences in hand-crafted prior knowledge rather than search quality.

BEN additionally demonstrates that decomposition reduces effective program complexity. Tasks requiring solution programs with up to eight independent transformation rules are solved by running eight shallow synthesis loops rather than a single search to depth eight, a reduction that makes previously intractable tasks accessible. An ablation analysis further confirms that both segmentation and analogical alignment contribute substantially to performance, with BEN solving roughly 33% more tasks than a version relying on random component correspondences under a constrained time budget.

2.3 Segmentation in BEN

Segmentation is the entry point to the DA&C pipeline: before any transformation can be learned, each input/output example must be decomposed into discrete components that the align and conquer stages can operate on. In BEN, this is implemented as a pixel-expansion procedure guided by a small grammar of segmentation modes.

The core algorithm traverses a bitmap from top to bottom and left to right. At each unassigned non-background pixel, a new object is instantiated and expanded by recursively absorbing neighbouring pixels that satisfy the active segmentation constraints. The result is a set of mutually exclusive objects, each described by its bounding box, colour set, shape, size, and spatial coordinates.

Specifically, expansion proceeds as a **breadth-first search (BFS)**. Starting from the seed pixel, the algorithm initialises a queue and iteratively pops pixels, adding each unvisited neighbour that passes the mode's permissibility check back into the queue. A neighbour is permissible if it (i) is not the background colour, (ii) has not already been assigned to any existing object, and (iii) satisfies the colour and connectivity constraints of the active mode, for instance, the *not_diagonal* mode requires a neighbour to share the seed object's colour and restricts connectivity to the four cardinal directions (no diagonals). When the queue is empty, the object's pixel set, bounding box, and ranked features (size, colour frequency, position, shape) are finalised. Figure 3 illustrates this process on a simple 6×6 grid under the *not_diagonal;0* mode (background colour = 0, four-connected, monochromatic).

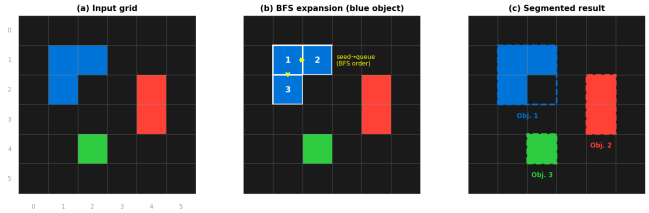

Figure 3: Segmentation of a 6×6 ARC grid under the *not_diagonal;0* mode. (a) The input grid (dark background, three coloured objects). (b) BFS expansion of the blue object: digits show the order in which pixels are visited, with the seed pixel labelled 1; yellow arrows show the expansion frontier. (c) Final segmentation result with each object enclosed in its computed bounding box.

The segmentation grammar in the current BEN in Python

approach defines nine modes along two independent axes: whether connectivity includes diagonal neighbours or only direct ones, and whether objects must be monochromatic or may span multiple colours. An additional mode handles output-primed segmentation, in which the output is a scaled or tiled version of the input and is partitioned accordingly. BEN searches these modes in order, retaining whichever yields the most useful decomposition for the downstream synthesis, estimated greedily on the first output component.

The grammar covers a broad range of ARC tasks but has known limitations. It cannot express segmentation boundaries defined by relational or structural properties, for instance, grouping pixels that form a closed contour, or objects whose identity depends on context.

3 Ranked Segmentation and Extension of Grammar

3.1 Reimplementation of Segmentation in Julia

A core contribution of this work is a faithful reimplementation of BEN's segmentation module in Julia, replacing the original Python codebase as part of a broader effort to integrate the DA&C framework with the Herb.jl program synthesis ecosystem. The reimplementation preserves the full segmentation grammar of the original system while introducing several targeted extensions.

Julia is a compiled language that generates native machine code via LLVM [11], in contrast to CPython which interprets bytecode at runtime. For a segmentation algorithm that repeatedly traverses grids and expands objects through breadth-first search, this eliminates interpreter overhead on the inner loop. Additionally, Julia arrays with a concrete element type store values contiguously in memory without boxing, improving cache locality when scanning ARC grids. These properties make Julia a natural fit for the pixel-level operations that dominate the segmentation stage.

The segmentation algorithm is reproduced in its entirety: bitmaps are traversed pixel by pixel, objects are grown by recursive neighbour expansion, and for each extracted component a bounding box is computed alongside a set of ranked relational features: size, position, color frequency, and shape, that express each object's ordinal standing within the scene rather than its absolute attribute values. These ranked features are computed once after segmentation and are subsequently consumed by the alignment stage, where the Structure-Mapping Engine uses them to establish structurally consistent correspondences between input and output objects. All segmentation modes of the original grammar are supported, covering the two-dimensional space of connectivity (direct neighbours only, or including diagonals) and color constraint (monochromatic or multicolor), combined with three background handling strategies (fixed to zero, absent, or inferred by majority color, the latter treating the most frequently occurring color in the grid as background rather than assuming a fixed value). The primed segmentation mode, which tiles the output according to the shape of a single input component, is also carried over.

3.2 Extension of the Segmentation Grammar

3.2.1 Proximity mode

Analysis of the ARC dataset revealed that many conceptually coherent objects are not strictly connected: their constituent pixels may be separated by small gaps. To handle such cases, we introduce a new segmentation mode, `proximity;N`, in which object expansion considers all pixels lying within a Chebyshev distance of at most N from the current expansion point, rather than only the immediately adjacent pixels [1]. Equivalently, this corresponds to the region reachable in at most N king moves on a chessboard. The color constraint is preserved: only pixels of matching color are admitted during expansion. This mode is illustrated in Figure 5, which contrasts standard 8-connected expansion with `proximity;2`. Including this mode in the segmentation grammar allows tasks in which objects contain small internal gaps to be segmented correctly, potentially increasing the number of solvable tasks.

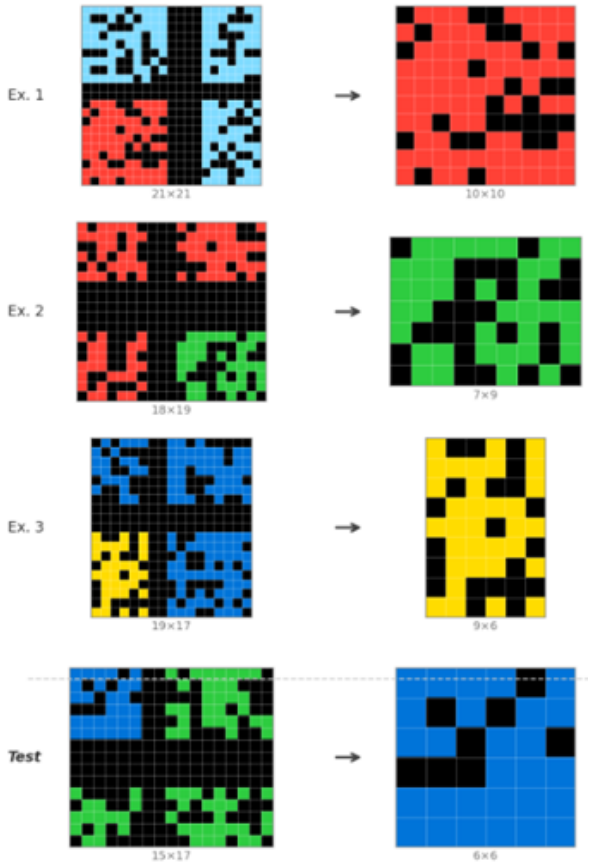

Figure 4: ARC task `0b148d64` (training example 1). The input grid is divided into four quadrants by bands of black pixels (dashed lines); the bottom-left quadrant (red, outlined) is the target object. Its internal black gaps and the fact that its constituent pixels are not all directly connected to one another cause standard segmentation modes to produce many disjoint fragments; `proximity;2` correctly unifies the entire quadrant into one object.

Task `0b148d64`, shown in Figure 4, illustrates the challenge that motivated this work: the input grid is partitioned

into quadrants by bands of black pixels, and the task requires identifying and extracting the uniquely colored quadrant (highlighted). That quadrant contains many internal black gaps and its constituent pixels are not all directly connected to one another, so standard connectivity-based segmentation fragments it into many disjoint pieces; the `proximity;2` mode bridges these gaps and groups the entire region as a single object.

3.3 Segmentation ranking

3.3.1 Posterior-based Ranking

The original DA&C framework selects a segmentation mode by running every candidate mode through the full downstream pipeline before making any decision. Concretely, the Python implementation iterates over a fixed list of nine modes in declaration order. For each mode it first segments all training grids, then searches for correspondences between the first input and the first output component of the first training example only (capped at three correspondences), then invokes the transformation search to find a program that reconstructs that component, and finally computes a posterior of the form $\log(\text{likelihood}) + \log(\text{prior})$. The likelihood is defined as the fraction of total output pixels that are explained either by a reconstructed object or by a correctly predicted background pixel; the prior penalizes program complexity by counting the number of function applications. Only after this full pipeline has been executed for all nine modes does the system sort by posterior and commit to the top-ranked mode. This approach has two significant drawbacks. First, the cost of correspondence search and transformation synthesis must be paid for every mode, making mode selection a dominant fraction of the overall runtime. Timing measurements on a small sample of tasks confirm this empirically: a single mode evaluation takes between 1 and 10 seconds in practice, with occasional outliers exceeding 40 seconds when the generality estimator runs deep. The BFS pixel expansion itself is negligible by comparison - the bottleneck is the downstream pipeline that must execute in full for every candidate mode before any ranking decision is made. Second, the scoring is anchored exclusively to the first training example and the first output component within it, ignoring whether the chosen mode produces a coherent, consistent decomposition across all training pairs.

3.3.2 A New Ranking Function

Our implementation in Julia replaces this with a lightweight ranking step that executes before any correspondence or transformation search. The function `rank_modes` scores each candidate mode using only the segmentation results themselves, via the function `score_mode`. For a given mode, `score_mode` segments all training input grids and all training output grids, then computes a structural score for each training pair from three components: a count-match term that rewards modes for which the number of input objects equals the number of output objects:

$$\text{count-match} = \begin{cases} +5 & \text{if } n_{\text{in}} = n_{\text{out}} \\ -0.5 \cdot |n_{\text{in}} - n_{\text{out}}| & \text{otherwise} \end{cases} \quad (1)$$

a count penalty that discourages over-fragmentation (applied when more than 30 objects are produced), and a size penalty that flags degenerate single-pixel decompositions (applied when the mean object size falls below 1.5 pixels and more than five objects are produced). The per-pair scores are then averaged across all training examples, and a consistency penalty equal to their standard deviation is subtracted. This final term is a key addition absent from the original: it explicitly rewards modes that decompose every training example in a structurally similar way, rather than modes that happen to score well on a single example but behave erratically on others. Any mode that produces an empty segmentation on any training input or output is immediately disqualified with a score of $-\infty$, short-circuiting further evaluation. The resulting ranked list is produced over eleven candidate modes rather than the original nine, adding the `inferrepeat` background variant (mode that infers the background color as the most frequently occurring color in the grid rather than assuming a fixed value) and the new `proximity;N` mode introduced in this work. The mode with the highest score is tried first by the downstream pipeline. Because the scoring heuristic is imperfect and can rank structurally unsuitable modes highly, the system falls back to successively lower-ranked modes if the top choice fails to produce a passing solution, whether due to segmentation failure, correspondence search timeout, or unsuccessful transformation synthesis. This fallback was introduced after observing that the ranker occasionally assigns anomalously high scores to modes that produce degenerate or over-fragmented decompositions. To further compensate for the heuristic's imperfection, `not_diagonal;0` is always guaranteed to be attempted no later than second: if it does not rank first, it is explicitly inserted into the second position in the candidate list, ensuring that the most reliable base mode is always tried early regardless of the ranker's output.

Evaluating modes on segmentation output alone, before any correspondence or transformation search, avoids paying the full pipeline cost for modes that are unlikely to work. The signal used is also stronger than in the original system: rather than scoring a single training example, all training pairs are considered, and the score reflects structural properties such as object count agreement and decomposition consistency rather than raw pixel coverage. The ranking step itself completes in milliseconds per mode.

4 Evaluation

4.1 Experimental Setup

All experiments are conducted on the 400-task ARC training set. Each task is allocated a wall-clock budget of 140 seconds, consistent with the evaluation protocol of the original BEN implementation [5]. The Julia reimplementation is executed on a single machine running Julia 1.12.6; no parallelism is used across tasks. For each task the system runs `rank_modes` to score all 11 candidate segmentation modes, selects the highest-scoring mode, and falls back to successively lower-ranked modes if the top choice fails to produce a passing solution, whether due to segmentation failure, correspondence search timeout, or unsuccessful trans-

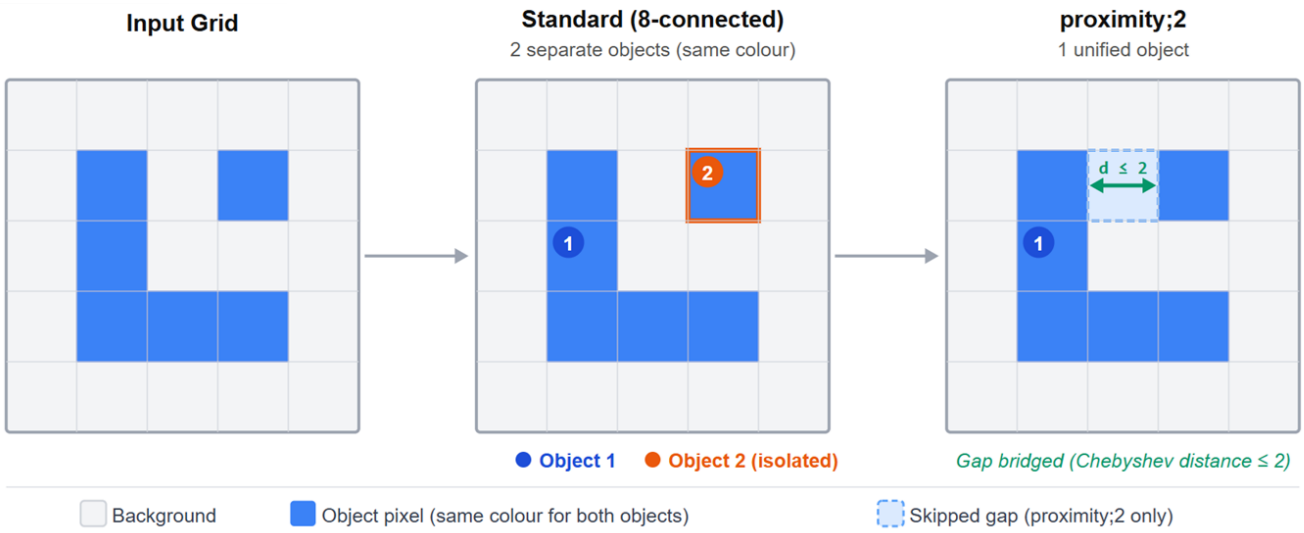

Figure 5: Effect of the `proximity;2` segmentation mode on a same-color input. *Left*: input grid with an L-shaped object and one isolated pixel of the same color. *Centre*: standard 8-connected segmentation yields two separate objects despite identical color. *Right*: `proximity;2` expands each pixel to all same-color neighbours within Chebyshev distance ≤ 2, bridging the single-cell gap and merging both regions into one object.

formation synthesis. The task is marked PASS if the synthesised program correctly reconstructs the test output, FAIL if the pipeline completes without finding a valid program, and TIMEOUT if the 140-second wall-clock limit is reached.

Three questions guide the evaluation. First, how many tasks does the Julia reimplementation solve overall, and how does the outcome distribution break down across PASS, FAIL, and TIMEOUT? Second, do the newly introduced `proximity;N` modes contribute solves that are unreachable by the original segmentation grammar, and if so, how many? Third, how well does the lightweight ranking procedure perform: specifically, what fraction of solvable tasks are recovered by the top-ranked mode alone? This last question is operationalised as a rank-cutoff experiment in which the pipeline is restricted to the top-k ranked modes for increasing values of k.

4.2 Results

Overall performance. Of the 400 training tasks, 46 are solved (PASS), 87 complete without a valid program (FAIL), and 267 exceed the time budget (TIMEOUT). Of the 267 timeouts, 133 are caused by correspondence search, 128 by the overall budget, and 6 by the mode-ranking step itself.

Segmentation mode usage. Table 1 reports the distribution of the mode that ultimately solved each of the 46 passing tasks. `not_diagonal;0` accounts for 31 solves, `proximity;2` for 3, `diagonal;0` for 3, `not_diagonal;epsilon` for 3, `proximity;3` for 2, `multicolor_not_diagonal;epsilon` for 2, and `diagonal;inferrepeat` for 2.

Ranking quality. Of the 46 solved tasks, 27 are solved by the rank-1 mode, 13 by rank-2, 3 by rank-3, 1 by rank-5, and 2 by rank-6. Table 2 shows the cumulative solve count as a function of rank cutoff k. `multicolor_not_diagonal;epsilon` is assigned rank 1 in 18 of the 46 solved tasks but is never itself the solving mode. `not_diagonal;0` appears in the candidate list of 89% of all tasks and is ranked no lower than second whenever it appears.

Selected mode	Tasks solved
`not_diagonal;0`	31
`not_diagonal;epsilon`	3
`diagonal;0`	3
`proximity;2`	3
`proximity;3`	2
`multicolor_not_diagonal;epsilon`	2
`diagonal;inferrepeat`	2
Total	46

Table 1: Distribution of solving segmentation mode across the 46 passing tasks.

Rank cutoff k	Tasks solved
1	27
2	40
3	43
5	44
6	46

Table 2: Cumulative tasks solved when the pipeline is restricted to the top-k ranked modes. All 46 solves are recovered only at $k = 6$.

4.3 Discussion

Baseline comparison. Table 3 partitions the 400 tasks by outcome across both agents. The Python BEN implementation solves 90 tasks; the Julia reimplementation solves 46, of

which 33 are shared. Julia gains 13 tasks that Python does not solve.

Group	Tasks
Solved by both	33
Solved only by Python BEN	57
Solved only by Julia	13
Solved by neither	297
Total	400

Table 3: Four-way partition of the 400 ARC training tasks by outcome across the Python BEN baseline and the Julia reimplementation.

Sources of unique Julia solves. The 13 tasks solved exclusively by Julia decompose into two categories based on the mode that produced the solution.

Five tasks are solved via `proximity;2` (three tasks) or `proximity;3` (two tasks) - modes absent from the original Python grammar. These tasks are structurally unreachable by any Python BEN segmentation mode regardless of ranking or downstream behaviour, so the extended grammar is the most plausible explanation for these gains.

The remaining nine tasks are solved via modes that are also present in Python BEN: `not_diagonal;0` (five tasks), `multicolor_not_diagonal;epsilon` [2](two tasks), `not_diagonal;epsilon` (one task), and `diagonal;0` (one task). Because the segmentation grammar is identical for these modes, the difference must originate elsewhere in the pipeline. One plausible factor is the ranking procedure: by placing a suitable mode higher in the candidate list, the Julia ranker may allow the downstream pipeline to find a solution before the time budget is exhausted, whereas Python's fixed order may reach that mode too late. However, differences in the correspondence search or transformation synthesis implementations between the two systems could equally account for these solves, and without a controlled ablation that holds all other factors constant, no firm attribution can be made.

Ranking quality. The rank-cutoff results expose a weakness in the ranking procedure. Committing to the top-ranked mode alone recovers only 59% of solvable tasks, and all 46 observed solves are recovered only when falling back as deep as rank 6. The root cause is a structural bias: `multicolor_not_diagonal;epsilon` is ranked first in 18 of the 46 solved tasks yet never produces the solution itself, always yielding to `not_diagonal;0` at rank 2. This pattern is in part a direct consequence of the deliberate design choice described in Section 3.3: `not_diagonal;0` is explicitly inserted into the second position of the candidate list whenever it does not rank first, guaranteeing that the most reliable base mode is always attempted early. Without this guarantee, tasks in which the ranker assigns rank 1 to `multicolor_not_diagonal;epsilon` would lose their second-chance fallback, and many of the 13 tasks solved at rank 2 would likely time out before reaching a suitable mode. Since `not_diagonal;0` appears in the candidate list of 89% of all tasks and is ranked no lower than second whenever it appears, the ranker effectively degenerates into a near-fixed two-step procedure for the large majority of tasks, but this is by design: the purpose of the explicit insertion is to guarantee that a reliable base mode is always attempted early, regardless of the heuristic's output. Another issue is that the count-match term in `score_mode` assigns high scores to grids with many colors regardless of whether a multicolor segmentation is actually useful, which causes `multicolor_not_diagonal;epsilon` to be systematically ranked too high. A penalty that distinguishes the two would likely correct this bias and reduce reliance on the fallback mechanism.

5 Responsible Research

5.1 Reproducibility

All experiments are run on the publicly available ARC training set [2], which can be downloaded without restriction. The Julia reimplementation and the exact task results are available in the Herb-AI GitHub organisation;[3] the exact commit on which the experiments were run is archived at a permanent link.[4] The repository will be made public upon submission. The scoring constants used in `score_mode` - the count-match reward of $+5$, the per-object penalty of -0.5, the fragmentation threshold of 30 objects, the size threshold of 1.5 pixels, and the primed-mode score of 100 - are documented in the source code. A reader with access to the repository and a Julia installation should be able to reproduce the reported results by running the evaluation script with the same 140-second per-task budget. Note that wall-clock timing results may vary across machines due to differences in hardware and Julia's just-in-time compilation overhead on the first run.

5.2 Benchmark Integrity

All reported results are obtained exclusively on the 400-task ARC *training* set. The 600-task evaluation set is never consulted at any stage of development, parameter selection, or analysis, in accordance with standard benchmark practice. The scoring heuristic constants were set by hand prior to running the evaluation and were not adjusted in response to observed results; however, they were not derived from a held-out validation set either, which limits the strength of generalization claims (see Section 5.3). The binary pass/fail criterion follows the official ARC specification: the produced output grid must exactly match the ground truth.

5.3 Scope of Claims

The results reported in this work characterise the behaviour of the Julia reimplementation on the 400 ARC training tasks only. Several limitations bound the conclusions that can be drawn. First, the scoring constants in `score_mode` were chosen by intuition, trial and error without a formal parameter search; they may be implicitly tuned to the training distribution and could perform differently on the evaluation set

[2]The `epsilon` background strategy denotes an absent background: rather than treating any colour as background, the mode segments all pixels in the grid into objects.

[3]https://github.com/Herb-AI

[4]https://github.com/RP-BEN/BEN-in-Julia/tree/13a18278f8c3d978c4753bd82cf227f6c1fa57b7

or on future ARC variants. Second, the five tasks solved exclusively via proximity modes constitute a small sample; no statistical test can establish that these represent a reliable gain rather than an artefact of the specific tasks in the training set. Third, while a preliminary outcome-level comparison against the Python BEN baseline is presented in Table 3, no controlled ablation holding all other pipeline factors constant has yet been conducted; claims about the precise source of differences between the two systems therefore require further controlled experiments to substantiate. Fourth, the evaluation covers only the segmentation and mode-ranking stages; deficiencies in the correspondence search or transformation synthesis may mask segmentation improvements or failures.

5.4 Energy and Compute

The full evaluation over 400 tasks with a 140-second budget corresponds to a theoretical maximum of approximately 13.3 GPU-free CPU hours on a single machine. In practice, most tasks terminate well before the budget is exhausted, and the actual wall-clock time is substantially lower. No GPU acceleration is used at any stage. Compared to large-language-model-based ARC solvers, which can consume tens to hundreds of dollars of compute per task, the energy footprint of this approach is negligible. Julia's JIT compilation for hot code paths further reduces repeated execution cost relative to the Python baseline.

5.5 Upstream Dependencies

This work builds directly on three upstream artefacts. The DA&C framework and the BEN agent [5] provide the overall pipeline architecture; any limitations in correspondence search or transformation synthesis are inherited unchanged. The Herb.jl program synthesis library [8] provides the Julia infrastructure for program enumeration and evaluation; correctness of the synthesis stage depends on the correctness of that library. The ARC benchmark itself [2] defines the task distribution and evaluation criterion; the benchmark's known biases towards certain geometric transformations are reflected in the task coverage reported here.

6 Conclusions and Future Work

This work investigates whether BEN's segmentation stage could be faithfully reimplemented in Julia and extended with improved modes and ranking to increase both computational efficiency and task coverage on the ARC benchmark.

The reimplementation reproduces the full segmentation grammar of the original system and integrates cleanly with the Herb.jl program synthesis ecosystem. Two concrete extensions are introduced: the `proximity;N` segmentation mode, which handles objects with small internal gaps by expanding to same-color pixels within Chebyshev distance N; and a lightweight structural ranking procedure that scores all candidate modes before any correspondence or transformation search is performed, using all training examples rather than only the first. Evaluated on the 400-task ARC training set, the system solves 46 tasks, of which five are solved via proximity modes absent from the original grammar, suggesting that the extended grammar addresses a gap in the original segmentation vocabulary. A more controlled comparison against the Python BEN baseline, holding all other pipeline factors constant, is a pressing item for follow-up, as the current results do not allow firm attribution of differences to any single component. A more thorough timing comparison of the segmentation stage between the Julia and Python implementations would also be valuable. While a small experiment on a limited number of tasks suggests a speed advantage for the Julia reimplementation, a systematic benchmark across the full task set is needed to quantify this concretely.

The ranking analysis surfaces a clear limitation of the current scoring heuristic: the count-match term systematically overranks multicolor modes, causing the ranker to degenerate into a near-fixed two-step fallback for the majority of tasks. Only 59% of solvable tasks are solved by the top-ranked mode, and recovering all 46 solves requires falling back as deep as rank six. Beyond the ranking heuristic, the segmentation grammar itself cannot express boundaries defined by relational or structural properties - closed contours, symmetry axes, or context-dependent object identity - which limits its applicability to the subset of ARC tasks amenable to pixel-expansion decomposition.

We identify several directions for future work. The most immediate is correcting the multicolor overranking bias, for instance by adding a penalty that distinguishes color diversity in the grid from the utility of a multicolor segmentation mode, and validating this through the rank-cutoff experiment described in Section 4.1. A principled hyperparameter search over the scoring constants, currently set by hand without formal justification, would strengthen the generalization claims, though care must be taken to validate on a held-out set to avoid overfitting to the 400 training tasks used during development. On the grammar side, introducing additional modes could broaden task coverage further. One candidate is a connected-contour mode, which would group pixels forming closed boundaries rather than expanding by adjacency or proximity. Another direction is a symmetry-based mode, which would segment objects by their reflective or rotational structure, a property that appears frequently in ARC tasks but is not captured by any current mode.

References

[1] Michel Marie Deza and Elena Deza. *Encyclopedia of Distances*. Springer Berlin Heidelberg, 2009.

[2] François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

[3] Severin Bratus, David F. Jenny, Andreas Plesner, and Roger Wattenhofer. A survey on the abstraction and reasoning corpus. *preprint*. https://tik-db.ee.ethz.ch/file/cdfe49edf380526841ae8ac64c8a253d/.

[4] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 2017.

[5] Jonas Witt, Sebastijan Dumancic, Tias Guns, and Claus-Christian Carbon. A divide, align & conquer strategy for program synthesis, 2024.

[6] François Chollet. ARC-AGI-1 public leaderboard and o3 evaluation. https://arcprize.org/blog/oai-o3-pub-breakthrough, 2024.

[7] Brian Falkenhainer, Kenneth D. Forbus, and Dedre Gentner. The structure-mapping engine: Algorithm and examples. *Artificial Intelligence*, 41(1):1–63, 1989.

[8] Tilman Hinnerichs, Reuben Gardos Reid, Jaap de Jong, Bart Swinkels, Pamela Wochner, Issa Hanou, Nicolae Filat, Tudor Magurescu, and Sebastijan Dumancic. Extended paper: An introduction to Herb.jl: A unifying program synthesis library. *Journal of Machine Learning Research*, 01:1–19, 2026. arXiv:2510.09726v2.

[9] Gaël Gendron, Qiming Bao, Michael Witbrock, and Gill Dobbie. Large language models are not strong abstract reasoners. *arXiv preprint arXiv:2305.19555*, 2023.

[10] Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2):155–170, 1983.

[11] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.