Anytime Program Synthesis for the BEN ARC Solver

Exploring heuristically-driven backtracking for the DA&C paradigm

Jakub Florek[1]

Supervisors: Sebastijan Dumančić[1]**, Dekel Zak**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

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 20, 2026

Name of the student: Jakub Florek
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumančić, Dekel Zak, Arie van Deursen

Abstract

Program synthesis for the Abstraction and Reasoning Corpus (ARC) remains challenging due to large search spaces, limited computational budgets, and the propagation of early synthesis decisions throughout the solver pipeline. This paper investigates whether internal solver metrics can guide retriggerable transformation search within the BEN Divide, Align, and Conquer (DA&C) architecture. To address this question, we introduce an anytime transformation-refinement mechanism that resumes synthesis from previously explored search frontiers, along with a threshold-based retriggering strategy driven by transformation overfitting and solver-state metrics. The proposed approach was implemented in a Julia-based reimplementation of BEN using the Herb.jl program synthesis framework and evaluated on the ARC benchmark. The experimental results show that retriggerable synthesis substantially changes the distribution of search effort across correspondences, enabling deeper exploration of highly ranked alignments while preserving exploration of lower-ranked alternatives. Compared to the baseline solver, the proposed method explores significantly fewer candidate transformations. It produces a different set of solved tasks, demonstrating that it redirects synthesis toward different regions of the search space. However, these changes do not translate into a large improvement in overall ARC solve rate, with both approaches achieving comparable aggregate performance. These findings suggest that transformation synthesis is not the primary bottleneck in the current BEN architecture. Instead, overall performance appears to be constrained by earlier stages of the pipeline, including segmentation, correspondence generation, and the expressiveness of the transformation language. While retriggerable anytime synthesis does not increase aggregate benchmark runtime performance, it provides a structured mechanism for adaptive search control. It lays the foundation for future work on stateful, dynamically allocated program synthesis.

1 Introduction

Program synthesis is a technique in which an AI agent attempts to produce a computer program that solves a given problem, usually framed as a search over a Domain-Specific Language (DSL)[1]. As the complexity of target problems increases, the search space explodes, requiring highly sophisticated search strategies. A benchmark that severely tests these limits is the Abstraction and Reasoning Corpus (ARC) [2], which evaluates human-like fluid intelligence and remains largely unsolved by state-of-the-art AI.

To mitigate the state-space explosion problem in program synthesis, solvers often use various algorithmic strategies such as Divide and Conquer (D&C)[1], which break tasks into smaller, independently solvable subproblems. Recently, this has been expanded into a Divide, Align, and Conquer (DA&C) architecture [3]. The additional "Align" step performs a process akin to analogical reasoning [4] to find correspondences between input and output objects, maximizing their shared structure. Program synthesis is then performed for each correspondence and pieced back together in the later Conquer stage. The BEN solver, illustrated by Figure 1, which implements this architecture, has proven highly competitive on the ARC dataset, achieving a 22.50% success rate (90/400 test tasks) [3].

However, a critical limitation remains. While the authors suggest that "missing transformation primitives and object features are the main bottlenecks" [3], the current literature has not explored how suboptimal choices made early in the pipeline propagate through later stages of the solving process or how the solver could correct them. Currently, BEN's pipeline operates mostly in a forward direction, which can lead to suboptimal choices propagating in the pipeline. For example, an incorrect initial segmentation in the Divide step can lead to incorrect correspondences in the Align step, ultimately making it impossible for the Conquer step to synthesize a correct transformation. Alternatively, the solver might only find overfit transformations that fail to generalize to unseen tasks.

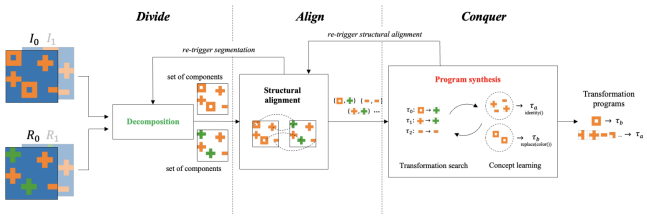

Figure 1: Full BEN solver pipeline. **Divide** segments input examples. **Align** establishes correspondences between segmented objects. **Conquer** synthesizes sub programs and combines them.[3]

BEN already supports retriggering earlier layers, but implements it only in a naive fashion. The solver only revisits earlier layers when critical failures occur, such as failing to find any meaningful correspondences, retriggering segmentation, or failing to find any transformations, retriggering the correspondence search. Both are visualized in Figure 1 by the two back edges between layers.

Expanding on this idea and understanding how the solver fails across the pipeline provides an opportunity to develop more advanced failure-driven backtracking heuristics. *Heuristic backtracking* is the process that would allow BEN to undo suboptimal choices using the knowledge from previous runs of the retriggered layer, as well as guidance from downstream failures. By combining those two sources of information, the solver could make better-informed decisions, correcting suboptimal choices before they propagate and enabling it to solve more complex tasks. This project was motivated by techniques such as Conflict-Driven Clause Learning (CDCL), which have proven highly successful in problems with large-scale search domains.[5][6][7]

This project specifically focuses on identifying and improving weak transformation synthesis outcomes. Therefore, the primary research question of this work is: *To what extent*

can internal solver metrics be used to guide retriggerable transformation search within the BEN pipeline?

To answer this question, the main contributions of this research are:

- An implementation of the BEN solver using Julia and Herb.jl,[8]
- The design and implementation of the anytime transformation search routine,
- The design and implementation of heuristically-driven backtracking to transformation search for BEN,
- An empirical evaluation demonstrating how retriggerable transformation refinement affects search behavior, transformation discovery, runtime characteristics, and task-solving performance on the ARC benchmark.

2 Background

This section presents key concepts in program synthesis and anytime algorithms, both of which are important for understanding BEN and the proposed architectural improvements.

ARC Benchmark and Challenges Most contemporary AI systems rely on large datasets to achieve strong performance. In contrast, the Abstraction and Reasoning Corpus (ARC)[2] was designed to evaluate the ability to generalize from a small number of examples and to test aspects of human-like intelligence.

ARC consists of a collection of tasks, each defined by a small set of input-output grid pairs as shown in Figure 2, typically around three examples. These examples specify a transformation that the agent must infer and then apply to unseen test inputs. Each grid is a two-dimensional array with dimensions ranging from 1 to 30 in each direction, where each cell can take one of 10 possible colors. An attempt at solving a task has a binary outcome, and an agent has three attempts to produce the correct output.[2]

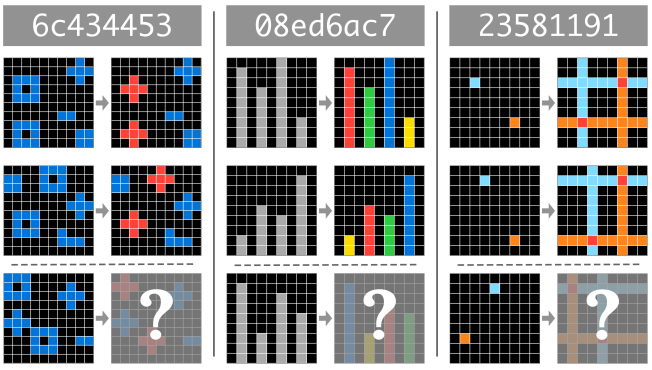

Figure 2: Example tasks from the ARC dataset illustrating object transformations on grids. Each task consists of one or more input-output training pairs from which the solver must infer a generalized transformation rule.

The dataset contains 1000 tasks, partitioned into training, validation, and hidden test sets. Importantly, tasks are constructed so they can be solved independently, without relying on patterns learned from other tasks or on dataset-level generalization.

ARC is specifically designed to be easy for humans but challenging for machine learning systems, as it requires compositional reasoning, abstraction, and the ability to generalize from a minimal number of training examples.[2] This combination of properties makes it a demanding benchmark for program synthesis approaches, which must efficiently search for programs that generalize beyond the provided examples.

Program Synthesis as Search A computer program can be represented as a derivation from a grammar $\mathcal{G}$, where production rules define valid transformations between syntactic constructs.[1] These derivations naturally form a search tree: internal nodes correspond to partial programs containing nonterminal symbols, while leaves correspond to fully specified programs. For expressive grammars with recursive production rules, this search space is infinite.

Program synthesis can therefore be formulated as a search problem over the space of candidate programs.[1] Given a set of input-output examples, the objective is to identify a program whose execution satisfies all provided constraints. Since exhaustive enumeration is infeasible, solvers rely on heuristic guidance to prioritize promising parts of the search space and prune infeasible or less promising branches.

Divide, Align and Conquer and BEN Divide and Conquer (D&C)[1] decomposes complex problems into smaller subproblems that can be solved independently and later combined into a global solution. In program synthesis, this introduces the challenge of determining which input and output components correspond to each other.[3]

The Divide, Align, and Conquer (DA&C) paradigm addresses this issue by introducing an alignment stage between decomposition and synthesis. BEN, the solver used in this work, implements this architecture as shown in Figure 1.[3] The **Divide** stage segments input and output grids into meaningful objects. The **Align** stage establishes correspondences between these objects using principles from Structure Mapping Theory (SMT), favoring structurally consistent mappings.[9] Finally, the **Conquer** stage synthesizes local transformations for each correspondence and combines them into a global program representation.

The alignment stage reduces synthesis complexity by enabling reasoning over smaller object-level transformations rather than entire grids.[3] However, both segmentation and alignment are inherently ambiguous, making the pipeline a sequence of interdependent decisions. Choices made early in the process constrain subsequent synthesis, while failures during synthesis may indicate that earlier decisions were suboptimal. Consequently, BEN supports limited backtracking over decompositions, alignments, and transformations.

Anytime Algorithms Anytime algorithms are computational procedures that progressively improve solution quality as additional computation time becomes available [10]. Unlike traditional algorithms, which only produce useful outputs after termination, anytime algorithms can be interrupted at arbitrary points during execution while still returning a valid intermediate solution.

Several properties characterize well-behaved anytime algorithms, including interruptibility, monotonic quality improvement, and resumability [10]. These properties make anytime methods particularly suitable for combinatorial search problems where exhaustive enumeration is computationally infeasible.

Program synthesis naturally fits this setting because a solver can often generate candidate programs incrementally through enumerative search. Initial synthesis iterations may quickly yield valid but overly specialized programs, whereas discovering more general transformations may require deeper exploration. As a result, synthesis systems could benefit from mechanisms that allow search to resume and progressively refine previously discovered solutions rather than restarting enumeration from scratch.

Our work adopts this perspective by reformulating BEN transformation synthesis as a resumable anytime search process. Instead of performing one-shot BFS synthesis with a fixed depth independently for each correspondence, the solver stores intermediate search states and incrementally refines transformations within dynamically allocated computation budgets. This dynamic budget allocation allows computation to focus on correspondences whose current transformations don't generalize (solve other correspondences), while continuously improving solution quality as additional budget becomes available.

3 Transformation Search Backtracking Strategy

This section describes the implementation of the retriggerable transformation search algorithm and optimization of threshold-based backtracking policies. The experiments were conducted using a new Julia implementation of BEN built on top of the Herb.jl[8] program synthesis framework.

The Julia reimplementation does not reproduce the full performance of the original BEN solver.[3] The reproduced baseline achieves an 8% solve rate, corresponding to 33 solved tasks out of 400, which serves as the reference point for all comparisons in this study.

The first subsection discusses the algorithmic design of the anytime search routine as described in Algorithm 1 and Algorithm 2. The second subsection takes a broader, architectural perspective on how the new search routine fits with the other parts of the solver, as shown in Figure 3.

3.1 Anytime Transformation Refinement

The original BEN implementation performs transformation synthesis independently for each correspondence produced during the alignment phase. Let $C = \{c_1, c_2, \ldots, c_n\}$ denote the set of correspondences ordered according to the Structural Mapping Engine (SME) similarity score.[11] For each correspondence $c \in C$, the solver constructs a correspondence-specific grammar and performs exhaustive enumerative program synthesis using Herb's breadth-first search (BFS)[12] iterator up to a predefined maximum depth k.

Each synthesized candidate program t is subsequently evaluated for generality using the function `evaluate_generality`(t). Rather than validating the program only against the correspondence from which it originated, the transformation is applied to all training examples and matched against all output objects. The procedure returns the set

$$\Gamma(t) = \{c_i \mid t \text{ correctly transforms the input object of } c_i\},$$

which represents all correspondences covered by the transformation t. We define the *coverage*, also referred to as *generality*, of a transformation as the cardinality $|\Gamma(t)|$. The synthesis stage stores transformations along with their coverage sets and prioritizes those with the highest coverage in subsequent stages of the divide-and-conquer pipeline.

This baseline formulation works well when the synthesis depth k remains relatively small. However, as the search depth and number of correspondences increase, exhaustive BFS enumeration becomes computationally expensive. More importantly, the baseline design lacks a mechanism for resumable refinement. Once the synthesis of a correspondence terminates, the search state is permanently discarded. Any future exploration, therefore, requires synthesizing transformations for entirely new correspondences rather than continuing the search for already explored ones.

This behavior is particularly problematic because the correspondences explored first are also those ranked highest by the alignment phase. If an early correspondence receives an overfitting transformation or fails to find a transformation, the downstream concept-learning phase might not be able to find an effective composite program. In practice, transformations with generality equal to one frequently indicate such overfitting behavior. The inability to revisit and further refine

Algorithm 1 Anytime Transformation Refinement

Require: Correspondences C, transformation set T, round robin round budget B_{local}, refinement budget B

$U \leftarrow$ set of uncovered output objects
$Q \leftarrow \{c \in C \mid |\Gamma(c)| \leq 1 \wedge c \text{ is resumable} \wedge o(c) \in U\}$
$spent \leftarrow 0$
while $spent < B$ and $Q \neq \emptyset$ **do**
  if $U = \emptyset$ **then**
    return
  end if
  $n \leftarrow |Q|$
  for $i = 1$ to n **do**
    if $spent \geq B$ **then**
      return
    end if
    $c \leftarrow$ pop_front(Q)
    $b_i \leftarrow B_{\text{local}} \cdot w_i$
    refine correspondence c using budget b_i
    update uncovered output set U
    $spent \leftarrow spent+$ refinement runtime
    update stagnation counter
    if c remains resumable, $|\Gamma(c)| \leq 1$, and stagnation threshold not exceeded **then**
      push c back into Q
    end if
  end for
end while

Algorithm 2 Budgeted Correspondence Refinement

Require: Correspondence c, local budget b
Resume BFS enumeration from stored iterator state
$t_{start} \leftarrow$ current time
while elapsed time $< b$ **do**
    Generate next candidate program p
    if search space exhausted **then**
        mark c as non-resumable
        return
    end if
    if depth$(p) >$ current depth frontier of c **then**
        advance correspondence depth frontier
        store iterator state
        return
    end if
    if p violates primitive reuse constraints **then**
        continue
    end if
    if p correctly solves correspondence c **then**
        if $|\Gamma(p)| > |\Gamma(c_{\text{best}})|$ **then**
            reset stagnation counter
        else
            increment stagnation counter
        end if
        if $|\Gamma(p)| > |\Gamma(c_{\text{best}})|$ **then**
            store p as best transformation
        end if
        store iterator state
        return
    end if
end while
store iterator state

these correspondences motivated the design of a new synthesis strategy.

To address these limitations, we reformulate transformation synthesis as an anytime refinement process. Algorithm 1 summarizes the high-level refinement procedure. The central idea is to selectively retrigger correspondences whose current best transformation shows low generality and continue exploring their search spaces using an additional computation budget measured in seconds.

Instead of treating synthesis as a one-shot BFS procedure, the new approach maintains a global queue

$$Q = \{c_i \in C \mid |\Gamma(c_i)| \leq 1 \wedge c_i \text{ is resumable} \wedge o(c_i) \in U\},$$

where $o(c_i)$ denotes the output object associated with correspondence c_i and U is the set of output objects that are not yet covered by any transformation with coverage of at least two. The queue contains correspondences whose best-discovered transformation still overfits the corresponding example and has not been covered by a high-quality transformation yet. The solver then iteratively refines these correspondences using a weighted round-robin scheduling strategy. During each refinement step, a small local time budget is allocated to a correspondence, and the synthesis procedure resumes exploration from its previously stored search state.

Crucially, each correspondence stores:

- the current BFS iterator and its state,
- the current search depth frontier,
- the best transformation discovered so far,
- the corresponding coverage set,
- the correspondence-specific grammar and symbol table,

This persistence mechanism allows the Herb iterator[1] to resume search directly from the current frontier rather than restarting enumeration from scratch. Algorithm 2 details the actual refinement operation performed for a single correspondence. During refinement, the solver resumes BFS enumeration, evaluates newly discovered transformations, updates the best coverage achieved so far, and stores the updated iterator state before returning control to the round-robin scheduler.

A key implementation detail is the definition of the local budget b_i. Rather than allocating equal computation time to all correspondences, the scheduler uses an exponentially decaying weighting scheme

$$w_i = \frac{\exp\left(-\frac{i-1}{B_{\text{local}}}\right)}{\sum_{j=1}^{|Q|} \exp\left(-\frac{j-1}{B_{\text{local}}}\right)},$$

and assigns the local refinement budget as

$$b_i = B_{\text{local}} \cdot w_i.$$

Here, i denotes the position of a correspondence in the refinement queue. The weighted budget allocation scheme was chosen as a simple heuristic that prioritizes correspondences appearing earlier in the queue while still allocating a non-zero fraction of computation to lower-ranked ones. Since the initial queue inherits ordering from the SME similarity ranking, correspondences that are considered more structurally similar receive a larger share of the available refinement budget.

During refinement, correspondences that remain eligible for further searches are reinserted at the back of the queue. Consequently, the scheduler behaves as a weighted round-robin policy that balances two objectives: prioritizing high-ranking correspondences and maintaining exploration of lower-ranked alternatives. This policy allows promising correspondences to receive deeper exploration while preventing the refinement process from becoming exclusively focused on a small subset of the search space.

The implementation additionally tracks a stagnation counter for each output object. Whenever refinement fails to improve the best coverage associated with a correspondence, the corresponding counter is incremented. Correspondences whose counters exceed a fixed threshold are removed from the refinement queue. This mechanism prevents the scheduler from repeatedly allocating computation to correspondences that consistently fail to discover more general transformations.

As a consequence, the refinement process effectively transforms synthesis into an anytime algorithm: the solver quickly produces an initial valid transformation set and can continuously improve transformation quality as additional computation budget is used.

[1] https://herb-ai.github.io/Herb.jl/dev/tutorials/TopDown

The refinement process additionally incorporates iterative deepening at the level of individual correspondences. Rather than globally increasing the search depth for all correspondences simultaneously, each correspondence maintains its own current depth frontier. During refinement, BFS enumeration proceeds only until candidates exceed the currently explored depth, or use up the local budget, at which point the iterator state is stored and the refinement returns. Subsequent refinement rounds resume exploration from the stored frontier and progressively advance the depth limit.

This design encourages broad exploration during early refinement stages by preventing the solver from spending too much computation on any single correspondence before other correspondences have been explored. As a result, the scheduler initially favors shallow transformation search across many candidate alignments while progressively skewing towards deeper search in correspondences as the refinement budget is spent.

The refinement process terminates when one of the following conditions is satisfied:

1. the allocated budget for the round is exhausted,
2. the uncovered output set U becomes empty,
3. no further resumable correspondences remain,
4. or the maximum synthesis depth is reached on every correspondence.

Conceptually, the proposed method shifts the objective of synthesis from just finding valid programs toward progressively improving transformation generality. Rather than exhaustively enumerating all correspondences independently, the solver dynamically concentrates computation on transformations most likely to provide an improvement. Such dynamic resource allocation allows deeper exploration of promising regions of the synthesis space while avoiding unnecessary recomputation of previously explored states.

3.2 Threshold-Based Retriggering of Transformation Search

While the anytime refinement strategy enables transformation search to be retriggered an arbitrary number of times, it does not directly address when the system should decide to trigger an additional refinement cycle. In practice, not all refinement cycles are equally useful: some lead to stable, general transformations, while others repeatedly produce overfitting solutions or fail to increase coverage. This observation motivates the introduction of threshold-based retriggering metrics that determine whether additional global refinement iterations should be performed.

The goal of retriggering is to identify when the current transformation set is insufficiently general and therefore likely to benefit from additional synthesis effort. Instead of running refinement for a fixed number of iterations, the solver evaluates a small set of aggregate statistics after each refinement phase and decides whether to continue or finalize synthesis.

As shown in Figure 3, after each call to the anytime refinement procedure (Algorithm 1), the solver constructs a set of generalized transformations and analyzes their quality using two primary signals:

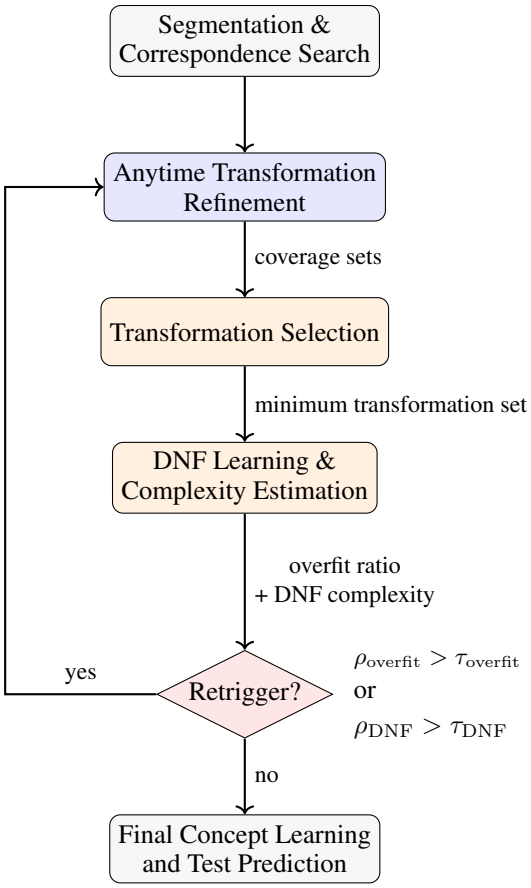

Figure 3: Overview of the retriggerable transformation refinement loop. The solver alternates between anytime transformation refinement, transformation selection, and concept learning. If the resulting transformation set shows excessive overfitting or high DNF complexity, additional refinement is triggered and search resumes from the previously stored synthesis frontiers.

- **Overfit ratio:** fraction of transformations selected during concept learning that only cover a single correspondence.
- **DNF complexity:** structural complexity of the learned concept representation.

These signals indicate whether the current transformation space is sufficiently expressive and general, or whether weak, correspondence-specific programs dominate it.

Overfit ratio Let T_g denote the set of generalized transformations after the refinement and concept discovery stage, and let $\Gamma(t)$ be the coverage set of transformation t. The overfit ratio is defined as

$$\rho_{\text{overfit}} = \frac{|\{t \in T_g \mid |\Gamma(t)| = 1\}|}{|T_g|}.$$

A high value of ρ_{overfit} indicates that most discovered transformations do not generalize beyond a single correspondence, suggesting insufficient exploration.

DNF complexity In addition to coverage-based metrics, the solver constructs a concept representation from the selected transformations using a DNF-based learner. The complexity

of this representation, denoted ρ_{DNF}, measures the structural complexity of the learned clause, i.e., the sum of the lengths of learned clauses. Intuitively, greater complexity suggests that the current transformation set requires a more complex logical structure to explain the observed correspondences, which may indicate weak synthesis outcomes.[12]

Retriggering is activated if either of the following conditions holds:

$$\rho_{\text{overfit}} > \tau_{\text{overfit}} \quad \vee \quad \rho_{\text{DNF}} > \tau_{\text{DNF}},$$

where τ_{overfit} and τ_{DNF} are fixed hyperparameters.

The retriggering logic is implemented at the full task solver level. As Figure 3 shows, after each refinement phase, the solver:

1. computes the current transformation coverage distribution,
2. constructs generalized transformations by resolving overlapping correspondences,
3. computes ρ_{overfit} and ρ_{DNF},
4. decides whether to invoke another refinement iteration or terminate.

If retriggering is enabled, the solver returns to the anytime refinement stage with a reduced remaining budget, allocating additional computation only when the current solution shows signs of weak generalization.

The motivation behind threshold-based retriggering is to distinguish between two kinds of solver behavior. On one hand, if transformations already exhibit sufficient generality, any additional search would likely yield diminishing returns. On the other hand, if the solver produces either overly specific transformations or high-complexity learned concepts, both of which could correlate with poor generalization performance, a refinement round could be helpful. The retriggering mechanism therefore adaptively allocates computation towards cases with high overfitting indicators, while avoiding unnecessary refinement once the solver finds a stable transformation set.

4 Experimental Setup and Results

Having described the design of the anytime transformation refinement and threshold-based retriggering mechanisms, we hypothesize that the proposed approach can improve the quality of transformation search by allocating additional computation to correspondences whose current transformations exhibit signs of overfitting. In particular, we expect the method to facilitate a deeper search for promising correspondences while still providing a shallow overview of the less informative ones.

To evaluate these hypotheses, we compare the proposed approach against the baseline BEN implementation without retriggering. Our experiments intend to investigate three research sub-questions:

Q1 What effect does retriggerable transformation search have on ARC task solve rate?

Q2 How does retriggering affect the depth and distribution of transformation search across correspondences?

Q3 What computational runtime trade-offs does retriggerable transformation search introduce?

The following subsections present a series of ablation studies designed to evaluate the effect of the anytime refinement procedure and threshold-based retriggering mechanism. All experiments were conducted on a machine equipped with an Apple M1 Pro processor and 16 GB of memory. Both solvers have been run with an overall timeout of 140 seconds and a search budget of 90 seconds. The baseline was run with a transformation search depth of 3. The budgeted solver was run with a single-refinement budget B of 30 seconds and a round-robin local budget B_{local} of 15 seconds. ρ_{overfit} and ρ_{DNF} were set to 0.4 and 5, respectively, after preliminary exploratory runs.

4.1 Solve Rate and Basic Statistics

To address research question **Q1**, we analyze the impact of the retriggerable transformation search mechanism on the solve rate and aggregate performance metrics. This evaluation is important for verifying whether our heuristic-driven backtracking loop enables the system to overcome the synthesis of suboptimal transformations during the forward execution pass.

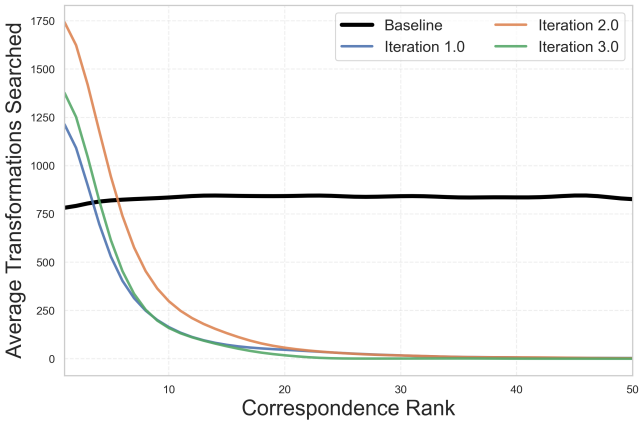

Figure 4: The figure depicts the distribution of synthesis search across correspondence ranks. The black line depicts the one-shot baseline search distribution, which performs an exhaustive, uniform search across all correspondences. In contrast, the three colorful lines depict the search distribution of the anytime transformation synthesis across refinement iterations.

Table 1 presents the comparative statistics aggregated across all 400 ARC training tasks under both the baseline implementation and our budgeted one.

The empirical results reveal that the task solve rate remains nearly unchanged between the baseline and budgeted configurations, with **33** and **35** solved tasks respectively. The distribution of unsuccessful outcomes differs substantially, however, because the two systems handle unsuccessful synthesis attempts differently. In particular, the budgeted solver tends to consume the remaining refinement budget before terminating, causing a larger fraction of unsolved tasks to be reported as timeouts rather than failures. It is also worth noting that average runtimes sometimes exceed the imposed timeout since the timeout mechanism is soft and waits for the respective synthesis sub-component to finish.

Configuration	Outcome	Tasks	Avg. Full Time (s)	Avg. Search Time (s)	Avg. Explored Transforms	Avg. Overfit Ratio(ρ_{overfit})	Avg. DNF Comp (ρ_{DNF})
Baseline BEN	Solved	33	31.15	3.89	7624.70	0.07	0.94
	Timed Out	80	92.95	72.15	115400.61	–	–
	Failed	283	50.71	21.85	34837.81	0.60	1.48
Proposed Solver	Solved	35	41.42	7.17	6553.14	0.04	0.97
	Timed Out	146	161.05	117.21	5608.19	–	–
	Failed	219	153.68	64.03	14036.73	0.54	0.78

Table 1: Aggregate solver performance statistics across the ARC training tasks for the baseline BEN solver and the proposed budgeted solver.

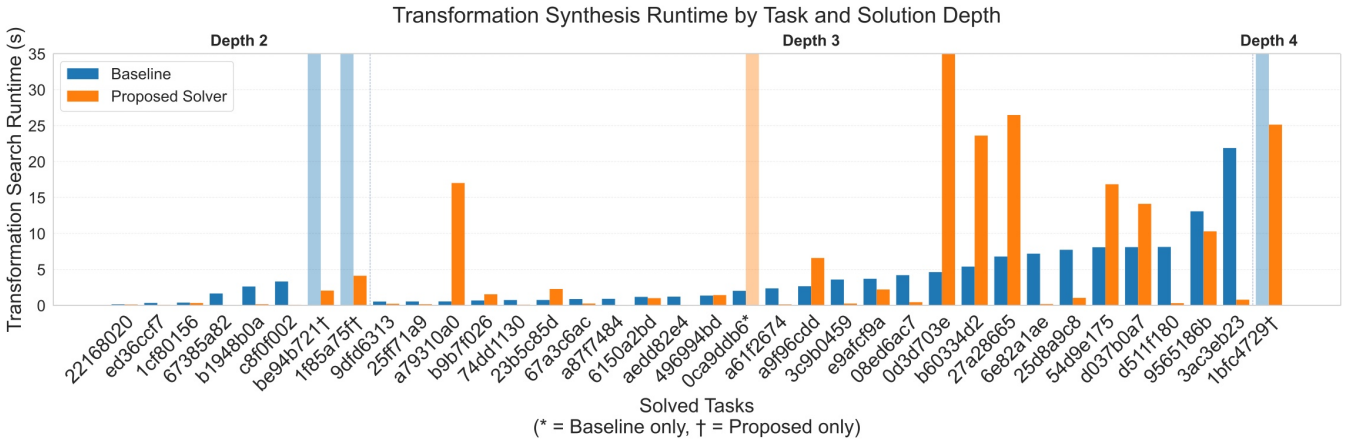

Figure 5: The bar graph shows the differences in synthesis runtime performance for tasks with solutions at depths two through four. The bar chart compares the transformation search execution time of the baseline implementation with that of the proposed anytime solver.

For solved tasks, the proposed approach reduces the average number of explored candidate transformations by approximately **16%**, from **7624** to **6553**, while maintaining a comparative solve rate. A similar trend is observed for timed-out and failed tasks. Going further, the average overfitting ratio (ρ_{overfit}) and the structural DNF complexity (ρ_{DNF}) metrics show a slight decrease, which could be attributed to the increase of timed-out tasks and how the retriggering mechanism runs over budget on correspondences with transformations that could not be improved further.

Despite exploring substantially fewer candidate transformations, the budgeted solver does not reduce transformation-search runtime. This discrepancy suggests that a significant fraction of execution time is spent outside candidate enumeration itself, for example, in refinement management, coverage evaluation, or other bookkeeping operations introduced by the retriggering framework.

The solved-task averages are additionally influenced by task **0d3d703e**, as shown in Figure 5. For this task, most of the correct transformations have only one coverage, prompting the solver to needlessly spend the entire search budget. Excluding this outlier reduces the average transformation search time and average number of explored transformations for solved tasks to **4.73** seconds and **3415.24** transformations, respectively.

These results suggest that, while the retriggering loop focuses synthesis on a smaller subset of transformations, overall performance remains constrained by other pipeline components.

4.2 Depth and Search Distribution

To evaluate research question **Q2**, we analyze how transformation search effort is distributed across correspondence ranks and refinement rounds. This analysis reveals whether the scheduler maintains a deep search for highly ranked correspondences while still allocating some search effort to lower-ranked ones.

Figure 4 details the differences in search distributions between the baseline solver and the implementation with anytime backtracking.

The plots illustrate that our framework changes the profile of program synthesis. In the baseline, represented by a single, horizontal line, the solver allocates its entire execution budget uniformly across all correspondences, including the least likely ones positioned at the tail of the list. As a result, this behavior allocates the same search effort to the first and last correspondences, regardless of the SME rank.

In contrast, the proposed anytime synthesis approach reveals a more resilient exploration behavior. While the framework continues to prioritize deep searches for top-ranked correspondences, it extends shallow searches to lower-ranked correspondences. Furthermore, as the queue in the budgeted approach is constantly changing due to the removal of correspondences with sufficiently general synthesized transformations, lower-ranked correspondences are pushed higher in the queue, subsequently receiving more computational budget as the search progresses.

The behavior is a direct consequence of removing correspon-

dences from the refinement queue once they are sufficiently covered, thereby progressively increasing the relative budget allocated to the remaining correspondences. This result supports our core hypothesis, confirming that heuristic-driven retriggering successfully focuses synthesis on the highest-ranking correspondences while preserving search in lower-ranking ones.

4.3 Solution Runtime Analysis

To address research question **Q3**, we evaluate the runtime trade-offs introduced by retriggerable transformation search across different solutions. This analysis allows us to determine whether iterative refinement introduces excessive overhead or improves computation allocation for more complex ARC tasks.

Figure 5 compares transformation synthesis runtimes between the baseline and proposed solver configurations.

The results indicate a substantially different behavior between solvers. Some tasks, such as **3ac3eb23** and **d511f180**, show improvement, while **a9f96cdd** and **27a28665** exhibit regression. The latter case, specifically, is a prominent outlier for which the proposed solver takes approximately 4 times longer to synthesize all transformations. The reason for this behavior appears to be the retriggering overhead that is not incurred by one-shot synthesis.

However, 22 tasks exhibit a significant improvement, which can be traced back to the iterative deepening component of the transformation refinement Algorithm 1. Due to the first, broad search over a large number of correspondences, the algorithm can quickly find the simplest, high-coverage solutions and short-circuit before accumulating costs from deeper, uniform search, as in the baseline.

It is worth noting that the baseline solves 1 task that the proposed solver does not, and the budgeted solver solves 3 tasks that the baseline does not. The budgeted solver struggles with task **0ca9ddb6** because the alignment phase produces high-ranking correspondences that uniquely result in overfit transformations with high coverage. The correct solutions, conversely, have even higher coverage. On the other hand, **be94b721** and **1f85a75f**, both not solved by the baseline implementation, exemplify another strength of the proposed algorithm. Due to the new search effort distribution shown in Figure 4, the solver can synthesize alternative transformations that remain consistent with the training examples while producing the correct test output. Lastly, thanks to a much deeper search at the head of the queue, the proposed solver was able to solve **1bfc4729** at depth 4. Crucially, the baseline, even with a depth limit of 4, failed to synthesize a correct solution within the overall 140-second timeout.

5 Responsible Research

This research explores improvements to the BEN program synthesis solver through heuristic-driven transformation search backtracking inspired by conflict-driven search techniques [5]. Although conducted in a controlled benchmark setting, several scientific responsibility considerations remain relevant.

The work uses the ARC benchmark, a synthetic dataset designed to evaluate abstract reasoning rather than real-world human data. Consequently, the research does not involve personal information, user-generated content, or sensitive datasets, resulting in minimal concerns regarding privacy, consent, or demographic bias. Nevertheless, advances in program synthesis contribute to increasingly capable AI systems, making transparency and interpretability important considerations. Compared to end-to-end learning approaches, the proposed methods provide greater insight into solver behavior.

The study also acknowledges limitations regarding generalization. Improvements observed on ARC may not transfer directly to other synthesis domains or real-world reasoning tasks. Accordingly, the results should be interpreted as architectural improvements within a specific program synthesis setting rather than broader claims about artificial general intelligence.

Artificial intelligence tools were used during the preparation of this paper for language improvement and programming assistance. Specifically, AI-based tools were employed to improve clarity, grammar, and readability of the written text, as well as to assist with code development and debugging. All technical decisions, implementations, experimental designs, analyses, and conclusions were independently developed, verified, and reviewed by the authors. The use of AI tools was limited to supportive tasks and does not alter the authorship of the research contributions presented in this work.

Reproducibility was considered throughout the project. The implementation is built using Julia and the Herb.jl framework [8], and experimental settings, solver configurations, and evaluation metrics are documented to facilitate replication. The exact version of the baseline and the budgeted solver used in this study, together with figure generation scripts, logging infrastructure, and reported experimental data, are available in the `DivideAlignConquerExperiments.jl` repository (`budgeted-solver` branch) under the Herb-AI GitHub organization[2]. Finally, all external ideas, algorithms, and prior research contributions are properly cited in accordance with standard academic integrity practices.

6 Conclusions and Future Work

This paper investigated whether the BEN solver could use internal solver metrics to guide retriggerable transformation search when applied to the ARC benchmark. To address that, an anytime transformation refinement mechanism and a threshold-based retriggering strategy were introduced into the BEN solver architecture.

The main conclusion of this work is that retriggerable transformation search substantially changes the profile of transformation synthesis within the BEN pipeline. While the overall ARC solve rate remains similar to the baseline implementation, the proposed refinement strategy alters how computation is distributed across correspondences, resulting in a different set of solved tasks. 3 tasks that the baseline could not solve become solvable under the proposed search strategy, while 1 task solved by the baseline is no longer solved. This difference suggests that the proposed method changes the regions of the synthesis space explored by the solver rather than simply increasing search depth or computational effort.

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

The experimental results further show that the proposed method reduces the number of explored candidate transformations while achieving a comparable solve rate. This evidence indicates that the refinement framework is successful at concentrating search effort on selected correspondences. However, the limited improvement in aggregate performance suggests that transformation synthesis may not be the dominant bottleneck of the current BEN architecture. Instead, overall performance appears to be constrained by earlier stages of the pipeline, including segmentation, correspondence generation, and the expressiveness of the underlying domain-specific language.

Several directions for future work naturally follow from these findings. First, extending the threshold-based backtracking to segmentation could directly address one of the main suspected failures, namely incorrect object decomposition. Heuristically retriggerable segmentation would allow the solver to revise not only transformations but also objects themselves. Second, introducing more expressive DSL primitives or clever pruning may reduce the search space and allow the solver to find better transformations deeper within the same budget. Third, due to time constraints, this work did not investigate the best possible thresholds for backtracking. Given the way the problem is formulated, one could employ hyperparameter tuning strategies such as grid or random search to find the optimal backtracking strategy.

An additional direction for future work is the development of hybrid search policies that combine the strengths of both synthesis strategies. The experimental evaluation revealed that the baseline and budgeted solvers solve partially overlapping sets of tasks, suggesting that the two search procedures explore different regions of the transformation space. Future work could investigate adaptive mechanisms for switching between search strategies or combining their outputs within a single solver.

In summary, this work demonstrates that transformation synthesis can be reformulated as a resumable, stateful, anytime search process with dynamically allocated computation. The proposed refinement framework maintains a comparable ARC solve rate while reducing the number of explored candidate programs and enabling the discovery of solutions missed by the baseline search procedure. The reduced number of candidate transformations, however, does not improve runtime performance. These findings suggest that adaptive transformation refinement is a viable mechanism for controlling search effort and may become more effective when combined with improvements to other components of the BEN pipeline.

References

[1] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling Enumerative Program Synthesis via Divide and Conquer". In: May 2017, pp. 319–336. ISBN: 978-3-662-54576-8. DOI: 10.1007/978-3-662-54577-5{_}18.

[2] François Chollet. "On the Measure of Intelligence". In: (Nov. 2019). URL: http://arxiv.org/abs/1911.01547.

[3] Jonas Witt et al. "A Divide, Align and Conquer Strategy For Program Synthesis". In: *Journal of Artificial Intelligence Research* 82 (2025), pp. 1961–1997. ISSN: 1076-9757; 1076-9757. DOI: 10.1613/jair.1.16847. URL: http://dx.doi.org/10.1613/jair.1.16847.

[4] Thomas G Evans. "A heuristic program to solve geometric-analogy problems". In: *Proceedings of the April 21-23, 1964, Spring Joint Computer Conference*. AFIPS '64 (Spring). New York, NY, USA: Association for Computing Machinery, 1964, pp. 327–338. ISBN: 9781450378901. DOI: 10.1145/1464122.1464156. URL: https://doi.org/10.1145/1464122.1464156.

[5] Gilles Audemard et al. "Integrating conflict driven clause learning to local search". In: *Electronic Proceedings in Theoretical Computer Science, EPTCS*. Vol. 5. Open Publishing Association, Oct. 2009, pp. 55–68. DOI: 10.4204/EPTCS.5.5.

[6] Martin Brain et al. "Deciding floating-point logic with abstract conflict driven clause learning". In: *Formal Methods in System Design* 45.2 (Oct. 2014), pp. 213–245. ISSN: 15728102. DOI: 10.1007/s10703-013-0203-7.

[7] John Slaney and Bruno Woltzenlogel Paleo. "Conflict Resolution: A First-Order Resolution Calculus with Decision Literals and Conflict-Driven Clause Learning". In: *Journal of Automated Reasoning* 60.2 (Feb. 2018), pp. 133–156. ISSN: 15730670. DOI: 10.1007/s10817-017-9408-6.

[8] Tilman Hinnerichs et al. "Herb.jl: A Unifying Program Synthesis Library". In: (Jan. 2026). URL: http://arxiv.org/abs/2510.09726.

[9] Dedre Gentner. "Structure-mapping: A theoretical framework for analogy". In: *Cognitive Science* 7.2 (1983), pp. 155–170. ISSN: 0364-0213. DOI: https://doi.org/10.1016/S0364-0213(83)80009-3. URL: https://www.sciencedirect.com/science/article/pii/S0364021383800093.

[10] Shlomo Zilberstein. "Using Anytime Algorithms in Intelligent Systems". In: *AI Magazine* 17.3 (1996), pp. 73–83.

[11] Brian Falkenhainer, Kenneth D Forbus, and Dedre Gentner. "The structure-mapping engine: Algorithm and examples". In: *Artificial Intelligence* 41.1 (1989), pp. 1–63. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(89)90077-5. URL: https://www.sciencedirect.com/science/article/pii/0004370289900775.

[12] Sumit. Gulwani, Oleksandr. Polozov, and Rishabh. Singh. *Program synthesis*. Now Publishers, 2017, p. 119. ISBN: 9781680832921.