SOLVING ARC-AGI WITH SYMBOLIC FEW-SHOT LEARNING AND TINY-SCALE ML

Touchchai Chotisorayuth
heartnetkung@gmail.com

ABSTRACT

Few-shot learning is the capability of Large Language Model (LLM) that allows it to perform any task specified by a few examples. It conceptually involves the use of pre-existing knowledge stored in the model and adapt them to the task at hand. Due to its connectionist nature, its safety and explainability cannot be guaranteed. Thus, we propose an alternative symbolic approach with equivalent ability that consists of 3 parts: parameterized functions, program synthesis algorithm, and tiny-scale ML. First, parameterized functions are used to encapsulate the pre-existing knowledge. Second, the program synthesis algorithm assembles a number of functions and their parameters into programs that can perform the task at hand. Last, our novel sample-efficient ML models are trained on the given examples to predict the functions' parameters suitable for the task. To make a supervised learning model operates robustly at a 2-3 sample scale, our approach uses Occam's Razor heuristic to search for the simplest models that fit the data instead of optimizing for loss function. That is, the space of possible models is defined as a Mathematical formula and that formula is optimized to find the simplest solution with the constraints of dataset matching. In addition, thanks to the optimization being primarily performed by Mixed-Integer Linear Programming, the solution has the optimality guarantee regardless of the sample size limitation. To measure the efficacy of our approach, the benchmark is performed on ARC-AGI-1 dataset which contains a number of IQ-test-like puzzles for our system to solve. The experimental result shows that our system can solve 204 of those puzzles which means that it can recombine the predefined knowledge and adapt them in a variety of ways to perform the task. Our contribution in this work is to propose a symbolic alternative to few-show learning as well as a novel sample-efficiency technique for supervised learning. The source code is publicly available at github.com/heartnetkung/symbolic_fsl.

***K*eywords** AI · ML · sample efficiency · program synthesis · Occam's Razor

1 Introduction

Few-shot learning is a machine learning paradigm emerged from Large Language Model (LLM) that enables the generalization from only a small number of training examples [1, 2]. To operate in this way, the model first needs pretraining on a gigantic dataset to acquire a variety of skills and knowledge. Then at inference time, it can apply these existing knowledge to the current task with some degree of adaptation to the given examples.

Even though the capability is flexible and sample efficient, LLMs contain a number of downsides which limit their usefulness in high-risk and mission-critical applications [3]. This includes the lack of interpretability, explainability, robustness, and safety guarantee [4, 5]. Thus, it is useful to have an alternative approach with a different tradeoff that is less capable but is fully controllable and explainable. In practice, controllability can be particularly beneficial if the skills can be added, upgraded, tested, and maintained.

For this reason, this paper proposes a symbolic approach to few-shot learning. The design is similar to a blackboard expert system [6] in that the skills and knowledge are represented by domain-specific functions and in each iteration these functions are called to collaboratively solve the problem at hand. Unlike traditional blackboard systems, our functions may have parameters that enable the modification of their behavior.

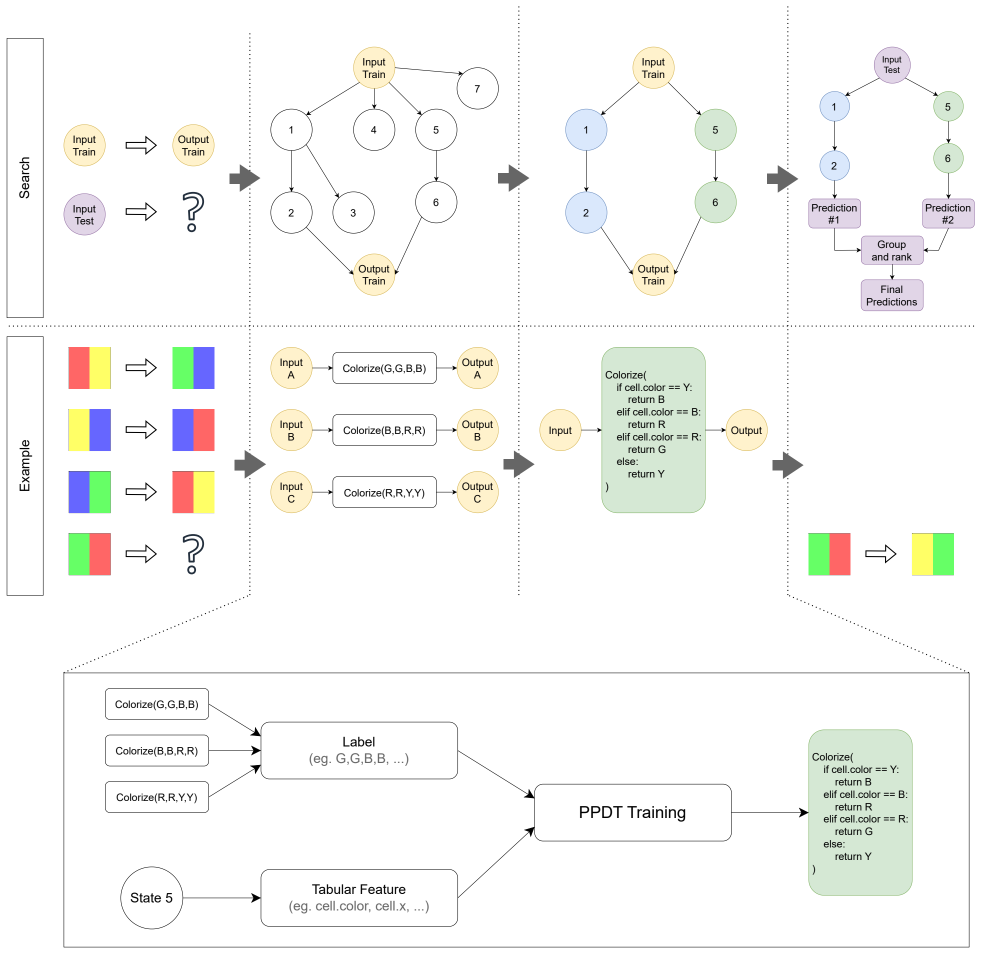

Figure 1: The diagrams show the 3 steps of MAP algorithm. Firstly, the pathfinding step searches for the sequences of expert functions and their parameters that match the input-output examples. Then, the inductive reasoning step trains PPDT models to generalize the parameter generation process. Finally, the model selection step selects the best solutions with minimum cost. The top diagram represents the search perspective that consider multiple candidates in each step whereas the middle shows an example perspective from a particular search path. In addition, the bottom diagram shows how PPDT models are used within the inductive reasoning step.

To solve the given task, the system uses a novel program synthesis algorithm, named Minimum Adjustment Planning (MAP), that can assemble these domain-specific functions into programs that match the input-output specification, then the programs are executed to generate the output predictions of the few-shot learning process. The algorithm contains two distinguishing features: pathfinding and ML-based inductive reasoning. On the one hand, pathfinding describes the procedure that generates the functions' parameters as the programs are continually being built by using the information from the current blackboard state and the terminal blackboard state (which is the example output). By repeatedly inspecting the terminal state, the program synthesis process can eliminate a number of invalid choices during the discrete search and improve its efficiency.

On the other hand, ML-based inductive reasoning step is responsible for converting the generated parameters into ML models. As illustrated in figure 1, these parameters are originally generated from the terminal states but, in order to perform inference on unseen data, they need to be generable from the inputs. Once the process finishes, the ML models are converted into equivalent programming syntax and is included as part of the program.

Unfortunately, the inductive reasoning feature cannot be implemented using the existing ML models in the literature due to the sample efficiency requirement. The crucial insight is that the loss function, which is used by most ML models, does not work at this scale. In this few-shot scenario, the number of features and parameters would likely overwhelm the number of samples. Once the number of parameters overwhelm the number of samples, any optimization would likely yield one of the many overfitted parameters with zero loss.

To solve this problem, we propose a novel model named Principle-of-Parsimony Decision Tree (PPDT) that optimizes the Occam's Razor objective rather than loss function. That is, the training algorithmically searches for the simplest model that perfectly fit the given data. This is similar to the concept of Automated Machine Learning (AutoML) that tries multiple types of model to find the most suitable one [7]. However, our approach defines the space of possible models in such a way that is algorithmically constructable and optimizable instead of trying them one by one. This means that our approach is much more efficient but is limited to this search space. In addition, the hard constraint on dataset matching substantially simplifies the training process by turning the non-linear optimization into a matching problem between model's partial prediction and a subset of samples.

In order to measure the efficacy of our approach, the implementation is evaluated on Abstract and Reasoning Corpus (ARC) dataset [8]. It contains a large suite of IQ-test-like puzzles that are designed to be easy for human but hard for computer. In each IQ-test-like puzzle, there are a few input-output examples (mostly 2-3 samples) in which the AI system needs to learn the transformational logic and then apply it to the new inputs. If the prediction is correct, it implies that our system successfully construct the programs from predefined skills as well as adapting them to the examples in a sample efficient way.

To summarize, this work proposes a program synthesis algorithm and a machine learning model that together function as symbolic few-shot learning process. On the one hand, MAP algorithm generates the program that match the input-output specification by assembling pre-existing parameterized functions. Its key feature is that the parameters of each function are generated by pathfinding process and generalized by inductive reasoning. On the other hand, PPDT model is proposed to solve the problem of sample efficiency in inductive reasoning process. It utilizes Occam's Razor heuristic to search for the simplest model that fit the given data by using its unconventional training process. Our contribution in this work is two fold. Our symbolic few-shot learning approach could be used for automation and problem solving as an alternative to LLMs. It can also incorporate LLM or AI agents in some of its function to increase its capability. Also, the technique used to achieve sample efficiency in PPDT model could be applicable to other ML models leading to less data, time, and energy consumption.

2 Minimum Adjustment Planning

Minimum Adjustment Planning is a program synthesis algorithm that is formulated as an extension of state-space planning. In simple terms, the planning is performed first, then the feasible plans are later converted into programs. As illustrated in figure 1, it is composed of 3 steps: pathfinding, inductive reasoning, and model selection. For interested readers, the step-by-step procedures can be found in the appendix.

2.1 Pathfinding

In the pathfinding step, the objective is to find a series of predefined functions that consecutively transform the initial state (the input of the program) to the terminal state (the output of the program). Compared to other planning algorithms, ours is different in that functions are parameterized and these parameters are generated by pathfinding methodology.

To illustrate how it works, let's take an example of navigating to a nearby cafe. The process would involve walking to the nearby train station, take the train to the station closest to the cafe, and then walk to the cafe. In programming terms, this process involves two functions: `walk(destination)` and `take_train(target_station)`. The parameters, `destination` and `target_station`, would be generated by examining the current state and the terminal state in each step of the iteration. In a sense, this is similar to how A* pathfinding algorithm works [9].

From the example, the responsibility of a function is not only to transform the current state to the next, but also to generate its own parameters by examining the current state and the destination. As a consequence, the applicable parameters need to be generable from such process, in other words they are *navigable*. Specifically, a parameter is navigable if all of its possible relevant values are determinable from the current state and the destination. With this definition, all parameters with constant limited possible values (enumerated types) are also navigable.

In addition, to enable a deliberate control over the loop, two additional components are introduced: controller and attention. On the one hand, controller is responsible for selecting the type of functions to be called on each iteration. This is useful for tasks involving heterogeneous functions. For ARC-AGI in particular, the types of function include ones that parse image to objects, ones that manipulate objects, and ones that convert objects back into image. On the other hand, attention selects the relevant part of the states to process on each iteration. This is useful when the task contains heterogeneous parts that should be handled separately.

With the mentioned components, the algorithm is as follows. In a search iteration, the process would start with the controller broadcasting the type of function to perform along with the attention. Then, each eligible function would

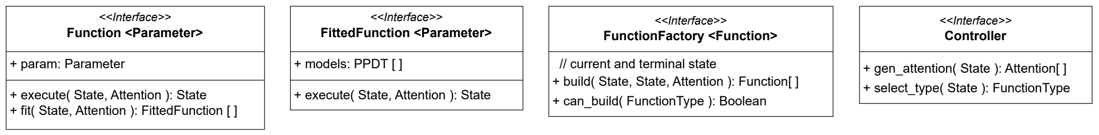
<<Interface>>
Function <Parameter>
+ param: Parameter
+ execute(State, Attention): State
+ fit(State, Attention): FittedFunction [ ]
<<Interface>>
FittedFunction <Parameter>
+ models: PPDT [ ]
+ execute(State, Attention): State
<<Interface>>
FunctionFactory <Function>
// current and terminal state
+ build(State, State, Attention): Function[ ]
+ can_build(FunctionType): Boolean
<<Interface>>
Controller
+ gen_attention(State): Attention[ ]
+ select_type(State): FunctionType

Figure 2: The class diagram representing the programming interface for defining expert functions and loop controller.

generate their own parameters. For each set of generated parameters, functions would be called to generate a future state before proceeding to the next iteration. On the algorithmic level, the states would be explored until the options are depleted. Once the search is finished, the result is a set of feasible plans with each plan consisting of a series of function that connects the initial state to the terminal.

2.2 Inductive reasoning

In inductive reasoning step, feasible plans that can transform all the input examples into the outputs are converted into programs. The difference between them is that a program can be executed on new inputs but a plan cannot. For this purpose, ML is used to turn values used in the plans into decision-making models.

There are two types of value used in planning that needs modeling: attention and parameter. On the one hand, a binary classification model is trained for each attention to extract its logic whether to attend or not. On the other hand, the modeling of parameters depends largely on the parameters' definition which results in a variety of model types as well as data extraction methodologies.

In terms of algorithm, the process, as illustrated in figure 1, is as follows. First, for each function in each feasible plan, retrieve the `current_states` from each example in which that function is executed upon and convert it into a feature matrix. Second, convert the values generated (attention or function's parameters) into label vectors. Third, train PPDT models on the feature and each of the label. Note that unlike traditional ML definition, our training process may return zero or more models at once. If multiple models are found, the program synthesis is branched off into multiple possibilities. On the other hand, if no model is found, the program synthesis for that branch is terminated. Once the process finishes, the outputs are the programs generated from all feasible plans.

2.3 Model selection

For model selection, the task is to select the simplest programs that perfectly fit the data. Since each plan can already transform the input into output and PPDT is optimized with dataset matching constraints, the data fitting is guaranteed for all generated programs. As for simplicity, it is defined as the number of binary operations required to formulate the models. These operations include addition, multiplication, and comparison. With this measurement, the simplest programs can be found simply by ranking the options. In addition, if the interested output is the prediction, the algorithm also perform majority voting as a tie-breaking mechanism.

2.4 Current limitation

At the current version, there are two major limitations in our algorithm. On the one hand, our path-finding method relies on exhaustive tree search which is slow. The reason is that, for the purpose of introducing the algorithm, correctness is more important than speed. The current solution relies on hard-code heuristics implemented in parameter navigation such that the unlikely search is early terminated. For example, the Colorize function used in figure 1 returns the navigable parameter (color) only when the source object and target object have the same shape. On the other hand, the inductive reasoning part can only be performed on plans with the same sequence of functions. This means that loops (which execute a dynamic number of functions) and conditions are not supported at the plan level, however both of the features can be implemented within the expert functions.

3 Principle-of-Parsimony Decision Tree

Principle-of-Parsimony Decision Tree is an ensemble of heterogeneous ML models. Together, they function as a machine learning model that is trained in a parametric fashion to optimize the simplicity objective with the constraints of perfectly fitting the data (achieve zero training loss). Among the supporting model types, the most important one is MILP polynomial regression since it is the only regression model and is the central piece of the algorithm. The rest of the models are binary classifiers included to provide a variety of modeling capabilities.

3.1 Ensemble training

As illustrated below, a PPDT model consists of multiple binary classification and regression models that form a single-level if-else structure. Specifically, the binary classifiers are used in the if statements whereas the regressors are used in the then statements.

```
def example_ppdt(x1:float, x2:float)->float:
  if binary_linear_classifier1(x1,x2):
    return polynomial_regressor1(x1,x2)
  if binary_linear_classifier2(x1,x2):
    return polynomial_regressor2(x1,x2)
  return polynomial_regressor3(x1,x2)
```

To train this ensemble, the objective is to find the simplest models that match all samples in the dataset. One way to achieve it is to build the ensembles with the least number of models. Conversely, this means that each regressor should match as many samples as possible. Furthermore, to maximize the matching of regressors and samples, the algorithm can be formulated as a greedy matching algorithm.

From this concept, the training is divided into two steps that begin with the greedy matching of regressors and samples. These regressors are designed to jointly maximize both the simplicity and the number of matching samples. Then, each iteration would train the regressors one by one, once a training finishes the matched samples are discarded and the iteration proceeds with the remaining samples. Once the iteration ends, there would be no sample left and all the regressors would be matched. Note that the training process of our regressor returns multiple models at once, thus the result of this process contains multiple combinations of regressors.

In the second half of the training process, each combination of regressor is examined one by one. Starting from the first regressor of the first combination, multiple binary classifiers are trained to find the condition in which the regressor makes the correct prediction. All classifiers that achieve zero training loss are kept for the assembling. Then, the matched samples are removed and the iteration proceeds to the next regressor. The training stops at the last regressor since it needs no pairing classifier. Afterwards, each combination of regressors and associated classifiers are assembled to form decision trees which are ranked by simplicity before returning.

3.2 The approximated cost of MILP optimization

Our complexity measurement of a decision tree is the count of binary operations used to form the tree which includes addition, multiplication, equal, not equal, greater than, and less than. Note that addition by zero and multiplication by one is excluded. One problem with this measurement is that it is not linear according to our formulation. In the formulation of both MILP classifier and MILP regressor, the optimizing variables are the polynomial's coefficient. Thus, in the context of MILP, the approximation form is used and afterwards they are reranked based on the actual measurement. Specifically, the approximating function is the L1 norm of the polynomial coefficient vector weighted by the number of operations used in that polynomial term (its polynomial degree plus one). The example calculation is shown in table 1.

Example decision tree	Actual cost	Approximation
return 3	0	3
return x_1	0	2
return $2 \times x_1$	1	4
return $-3 \times x_1 \times x_1$	2	9
return 0 if $x_1 < 0$ else $x_1 \times x_2$	2	5

Table 1: Example calculation of the actual complexity cost and its approximation.

3.3 MILP polynomial regression

For MILP polynomial regression, there are two objectives for the optimization, both of which require certain optimization techniques. The former uses basis pursuit formulation [10] to optimize L1 vector as mentioned by the approximated cost formula. The latter uses the big M method to greedily match as many samples as possible. To explain how it works, notice the constraints part of the formula below that the boolean variables b_i is zero if and only if the sample matches and the minimization of $\|b\|_1$ is equivalent to the greedy matching. From the mentioned techniques, the formulation for training and inference are as follow.

Let $X \in \mathbb{R}^{m \times n}$ be the dataset with m features and n samples, $P_2(X)$ be the second-degree polynomial terms generated from X , $y \in \mathbb{R}^n$ be the label, $M \in \mathbb{R}$ be a large constant, $\lambda \in \mathbb{R}$ be a hyperparameter for controlling the tradeoff between matching samples and minimizing operators, $\theta_0 \in \mathbb{R}, \theta_1 \in \mathbb{R}^m, \theta_2 \in \mathbb{R}^{(m)(m+1)/2}$ be the polynomial coefficients associated with degree 0,1,2, and $b \in \{0,1\}^n$ be boolean variables for detecting if the samples match.

$$\begin{aligned} \min_{\theta_0,\theta_1,\theta_2,b} \quad & |\theta_0| + 2\,\|\theta_1\|_1 + 3\,\|\theta_2\|_1 + \lambda\,\|b\|_1 \\ \text{s.t.} \quad & |\theta_0\mathbb{1} + X\theta_1 + P_2(X)\theta_2 - y| \leq Mb \end{aligned}$$

$$y_{pred} \;=\; \theta_0\mathbb{1} + X_{test}\theta_1 + P_2(X_{test})\theta_2$$

After the optimization, a postprocess is required to remove the overfitted results. The reason is that a polynomial with sufficient degree can fit any dataset by solving the polynomial equation.

3.4 MILP binary linear classification

In addition to the MILP regression, PPDT also includes a custom binary classifier optimized by Mixed-Integer Linear Programming (MILP) by using the approximated cost. Note that the zero-degree term is removed since it's the decision boundary of the classification.

Let $X \in \mathbb{R}^{m \times n}$ be the dataset with m features and n samples, $y \in \{0,1\}^n$ be the label, $\theta_0 \in \mathbb{R}, \theta_1 \in \{-1,0,1\}^m$ be the linear coefficients associated with degree 0 and 1, and $l \in \{-\infty, 0\}, u \in \{-1, \infty\}$ be the lower bound and upper bound which the values depending on y.

$$\begin{aligned} \min_{\theta_1} \quad & \|\theta_1\|_1 \\ \text{s.t.} \quad & l \leq \theta_0 + X\theta_1 \leq u \end{aligned}$$

$$y_{pred} \;=\; \mathbb{1}_{\theta_0 + X\theta_1 > 0}$$

3.5 Complimentary features

Even with the two MILP models, there are still three missing features that are commonly used in programming. First, nested if-else feature is implemented by adding decision tree classifier to the list of binary classifiers. Second, missing comparison operators (equal, not equal) is included as a straightforward comparison function. Last, there are cases where greedy matching cannot provide the best logic. When it happens, the best logic is usually pure association. Thus, a dictionary model is added to complement the greedy matching.

3.6 Vertical integration with MAP algorithm

By design, PPDT model and MAP algorithm use the same complexity measurement (the count of binary operation usage) from the model level to the program level. Greedy search and MILP are used at the model level to find the simplest models whereas exhaustive tree search is used at the program level to find paths composing of simplest models. Together, they form a nested search for the simplest programs given the examples.

4 Result

The goal of our experiment is to show that our few-shot learning algorithm is capable and adaptable by using it to solve ARC puzzles. However, the current approach is not scalable as it requires hard-code implementation of controller and expert functions as denoted in figure 2 and 3. In particular, MAP algorithm relies on a variety of expert functions for composing the solution programs. Future work is required to improve the scalability before it can solve a larger dataset like ARC-AGI-2. As a result, the scope of this experiment is limited to implement a sufficient number of functions for demonstrating its capability and adaptability.

In our implementation, there are 26 expert functions that are written in a generic way which means that adaptation is required for problem solving. There are 2 families of function that showcase this adaptability. On the

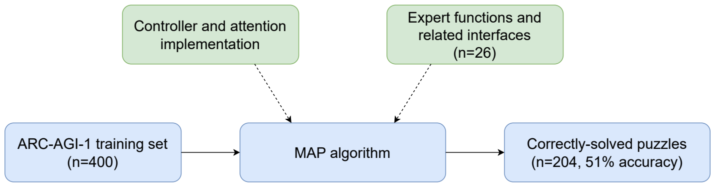
Controller and attention implementation
Expert functions and related interfaces (n=26)
ARC-AGI-1 training set (n=400)
MAP algorithm
Correctly-solved puzzles (n=204, 51% accuracy)

Figure 3: The diagram shows implementation settings and results on ARC-AGI-1 dataset. As a prerequisite, the algorithm needs the implementation of controller, attention, state, and expert functions. The experiment shows that our approach can solve 51% of the problems by adaptively apply some of the 26 predefined expert functions. In terms of machine learning, this result is equivalent to achieving 100% accuracy under 2 attempts on 204 datasets out of 400 since each puzzle has its own train/test set. The list of expert functions can be found in the appendix.

one hand, certain functions allow PPDT models to freely decide the color of each pixel in the output image by training the model directly on the given examples. This includes `fill_in_the_blank(object,pixels)` function that can redraw any object and `free_draw(pixels)` function that can redraw the whole image; they are used extensively in 61 and 23 of the solutions. On the other hand, certain functions perform its task by modifying the objects' properties with the values decided by PPDT. In programming terms, these are basic constructors and setter functions with obvious navigable parameters. The examples include `colorize(object,new_color)`, `move(object,x,y)`, `create_rectangle(x,y,width,height,color)` functions which are used in 38, 19, and 18 of the solutions. The full list of these functions along with their definitions can be found in the appendix. Note that these methods are designed according to the description of prior knowledge provided by the ARC dataset [8].

In addition to expert functions, the implementation includes controller and attention module. Controller is implemented in a straightforward manner which is basically parse the image into objects, keep transforming them, and then convert all objects back into image afterward. On the other hand, attention uses ARC-specific cluster-based algorithm designed to emulate how humans group similar objects into categories and apply homogeneous transformation rules upon them. The step-by-step algorithms of both controller and attention can be found in the appendix.

The evaluation method follows the competition rules of ARC competition [11]. Specifically, the system can make two attempts at predicting the correct output. The puzzle is solved if either of them exactly match the label. The platform used to run this experiment is Google Colab. The associated notebook is publicly available for reproducibility.

From the described settings, MAP algorithm can solve 204 puzzles in ARC-AGI-1 training set by generating solution programs composing of expert functions. The successful solutions contain an average of 3.46 expert functions with the maximum of 7. Note that among them, two functions are mandatorily used in all solutions. To generate each successful solution, the algorithm finds an average of 45.85 search paths in the pathfinding step but produces only 1.63 unique predictions on average. The main reason most search paths fail to generate the solutions is that the PPDT training automatically discard non-sensible high-cost overfitting models. For example, `free_draw(pixels)` function, which in theory can solve any puzzle, is found in all search paths but only a fraction of them succeed because the PPDT training return zero result. The mean execution time of the algorithm is 121.86 seconds per puzzle.

These results suggest that the algorithm can adapt the generic functions as well as has the capability to solve various puzzles from a small number of functions. In our opinion, both qualities are attributed to the features of PPDT that is trainable on small datasets, produce programming-like logic, and adapt any navigable function. However, the current disadvantages of our approach are the scalability on hard-coding the expert functions and combinatorial explosion if the number of expert functions becomes too large.

5 Discussion

In this work, we propose a few-shot learning algorithm with the sufficient capability and adaptability to solve ARC puzzles while having the benefit of the symbolic approaches including controllability, interpretability, and explainability. It first defines how the prior skills can be represented, and then provide a method to assemble and reason over them to solve the task. Its key features are the pathfinding methodology and tiny-scale ML models that enable the flexible adaptation of the existing skills. The reason the algorithm is designed this way is because we believe this is similar to how human brains perform few-shot learning.

There are multiple neuroscientific evidences and theories that mention similar mechanisms. First, Global Workspace Theory [12, 13] states that the brain includes centralized controller, attention, state-based shared memory, expert functions, and planning mechanism. Second, the theory of crystallized and fluid intelligence [14] distinguishes two types of human intelligence; one is capable of learning and adaptation and the other can retain and reuse the acquired skills. In terms of functionality, this implies that the brain can execute learned functions as well as train ML models. Third, Qiu et al. [15] shows that fluid intelligence is associated with certain regions of the brain, whereas crystallized regions can be found almost anywhere in the cortex. Notably, Supramarginal gyrus is associated with the working memory and Temporal gyrus/Occipital sulcus are adjacent areas supporting reasoning and adaptation. In terms of information flow in a planning algorithm, the data flows from memory, to adaptation, to function execution, and then return. For this reason, our design similarly uses ML to adapt the parameters of each function prior to its execution. Last, dual process theory states that humans have two modes of thinking; the system-one is fast and automatic whereas the system-two is logical, adaptable, slow, and effortful [16]. This is why our algorithm deliberately separates the planning part from the inductive reasoning part. In addition, the simplicity is only measured on the amount of reasoning effort (count of binary operations) since the intuitive planning is primarily system-one.

Aside from few-shot learning, an additional sample-efficient ML model is proposed to compose programming operators into if-else-then syntax that can be used by the program. Its key feature is that it optimizes for Occam's Razor objective instead of loss function to achieve its sample efficiency. In a nutshell, the training is equivalent to optimizing the ensemble structure and model parameters to maximize the simplicity and constraints by dataset matching.

In terms of ML design, the reason PPDT is formulated this way is the following. First, the combinatorically large search space of symbols is included in the optimization to ensure that the generalizing model is within the search space. This practice is commonly used in the field of Symbolic Regression (SR) to search for the closed-form Mathematical formula representing the given dataset [17]. Second, since SR is a NP-hard problem [18], direct solving would be too slow for our algorithm. For this reason, the hard constraints on dataset matching is used to simplify the problem by converting the searching problem into a matching problem. As a matching problem, additional optimization strategies can be applied including divide-and-conquer that iteratively reduces the search space, basis pursuit that allows the approximation of the complexity measurement, and big-M method that formulates the greedy matching as efficient MILP optimization. Last, Occam's Razor and program-like ensemble structure provide the crucial inductive bias for the model to generalize from limited training data. In a sense, it is possible that this inductive bias is the reason humans can learn from limited data and that LLMs can imitate this behavior.

In fact, Occam's Razor might be used by LLMs in a similar way through the phenomenon of double descent [19]. Consider that the training of LLM optimizes for both the loss function and the regularization term. In the first descent, gradient is dominated by the loss because of its large value. Once the model correctly fits the dataset and achieves almost zero training loss, the loss function now becomes the soft constraints on dataset matching. In addition, the lottery ticket hypothesis states that overparameterized LLMs contain a number of internal circuits used by the model to perform certain functions [20]. From this perspective, it is similar to PPDT in that it optimizes the ensemble of circuits for simplicity with the soft constraints on dataset matching.

In future works, our approach can be improved in various aspects including scalability, efficiency, and program synthesis capabilities. For scalability, future algorithms should automatically generate expert functions to remove the bottleneck of hard-coding. This can be achieved by having an LLM propose new functions as tasks are being solved and then retain the successful functions. Afterwards, the associated parameter navigation algorithm can be automatically generated. That is, the function can already generate outputs from inputs and parameters, parameter navigation is essentially generating parameters from inputs and outputs which is solvable by supervised learning. Using this procedure, more functions can be added over time. For efficiency, it can be enhanced in the planning phase of MAP algorithm by replacing the exhaustive tree search with reinforcement learning. In fact, once the system can automatically generate the state-transitioning functions and these functions can be planned by reinforcement learning, it would lead to AI that has compositional generalization [21], fluid intelligence [14], inductive reasoning capability, and minimal hallucination. Lastly for program synthesis, MAP algorithm should be improved to support loop, recursion, and conditions at the synthesis level.

6 Related work

Program synthesis with simplicity heuristic Potapov et al. (2015) proposes a probabilistic programming approach based on Bayesian Occam's Razor [22]. It works by sampling programs from the space of all possible programs evaluated by the dataset and penalized by the complexity. Compared to our approach, the key differences are the search approaches and the complexity definitions. For the search approach, their procedure iteratively samples the whole program at once informed by probability and simplicity whereas ours assembles functions one-by-one informed by the intermediate states and the terminal. For the complexity definition, theirs is measured by the number of bits used in both the model description and data encoded with the use of the model. Next, MADIL is a program synthesis approach based on Minimum Description Length principle with the goal of solving ARC-AGI-1 dataset [23]. Our approaches and theirs both aim to construct the program from the predefined building blocks with information from both the given examples and intermediate states. The key differences are selective attention, function signatures, and complexity measurement. For selective attention, theirs relies on compose-decompose method to divide grids into smaller selectable partitions whereas ours relies on ARC-specific cluster-based attention algorithm. For function types, all their functions are grid manipulators while ours has multiple function signatures which are selected at each iteration by the controller. For complexity measurement, theirs is based on the algorithm that assigns a complexity value to each of their generated program. On the contrary, our complexity measurement is based on the amount of adaptation from the feasible plans.

Symbolic regression with simplicity heuristic In the literature, there are multiple symbolic regression models that are similar to ours in that both optimize for the compositional structure while having the notion of complexity. First, Exhaustive Symbolic Regression model [24] works by exhaustively generate all permutations of symbol up to a certain complexity limit before using nonlinear optimization to finetune each model's parameters. Second, De Franca (2025) [25] proposes a genetic programming approach to generate possible functions in Transformation-Interaction-Rational format and then optimize the parameters using multi-objective optimization to find all models with different tradeoffs between loss and complexity. Last, de Carvalho Servia et al. (2025) [26] proposes an integrated kinetic model discovery used in chemical engineering composing of genetic programming, parameter optimization, model selection, and derivative data generation. Unlike other approaches, it is designed to be task-specific which means that it can exploit specialized properties like conservation of mass and differential equation to improve data efficiency and robustness. Among these papers, all of them define complexity as number of nodes in the expression tree. Compared to ours, their definitions include number of constants, variables, and operations but ours only includes the number of operations. The reason all mentioned papers use number of nodes is likely that their search spaces grow exponentially and the complexity threshold is required to limit their growing compute. Aside from complexity measurement, another difference is the speed requirement as their processes can run up to 200 hours [24] but ours need to run within seconds. Note that despite multiple similarities, our approach is not a symbolic regression because it contains another type of models (and not just symbols) like decision tree classifier and dictionary learning.

Tiny-scale ML Unfortunately, our literature review finds no supervised-learning model that aims for datasets with few samples. One possible reason is that most researches aim to propose generic models that are useful for multiple datasets. In addition, there is no need for the tiny-scale because there are learning paradigms like active learning and semi-supervised learning for that purpose. On the contrary, our objective is to design a task-specific model, thus the sample efficiency can be drastically improved by using task-specific inductive bias and optimization techniques.

7 Acknowledgement

Many thanks to Pariyakorn Chotisorayuth and Jayakorn Vongkulbhisal for improvement ideas and proofreading, without either of them this work wouldn't be this refined.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[2] Archit Parnami and Minwoo Lee. Learning from few examples: A summary of approaches to few-shot learning, 2022.

[3] Amit Sheth, Kaushik Roy, and Manas Gaur. Neurosymbolic ai – why, what, and how, 2023.

[4] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mane. Concrete problems in ai safety, 2016.

[5] Sajid Ali, Tamer Abuhmed, Shaker El-Sappagh, Khan Muhammad, Jose M. Alonso-Moral, Roberto Confalonieri, Riccardo Guidotti, Javier Del Ser, Natalia Diaz-Rodriguez, and Francisco Herrera. Explainable artificial intelligence (xai): What we know and what is left to attain trustworthy artificial intelligence. *Information Fusion*, 99:101805, 2023.

[6] Ram D. Sriram. *Blackboard Systems*, pages 227–284. Springer London, London, 1997.

[7] Yaliang Li, Zhen Wang, Bolin Ding, and Ce Zhang. Automl: A perspective where industry meets academy. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, pages 4048–4049, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Francois Chollet. On the measure of intelligence, 2019.

[9] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[10] Andreas M Tillmann. Equivalence of linear programming and basis pursuit. *Proc. Appl. Math. Mech.*, 15(1):735–738, October 2015.

[11] Francois Chollet, Mike Knoop, Greg Kamradt, Walter Reade, and Addison Howard. Arc prize 2025. `https://kaggle.com/competitions/arc-prize-2025`, 2025. Kaggle.

[12] Bernard J Baars. *In the theater of consciousness*. Oxford University Press, New York, NY, March 1997.

[13] Bernard J Baars. Global workspace theory of consciousness: toward a cognitive neuroscience of human experience. In *Progress in Brain Research*, Progress in brain research, pages 45–53. Elsevier, 2005.

[14] Raymond B Cattell. Theory of fluid and crystallized intelligence: A critical experiment. *J. Educ. Psychol.*, 54(1):1–22, February 1963.

[15] Bowen Qiu, Rui Qian, Baorong Gu, Zichao Li, Zhifan Chen, Xinyi Xu, Huaijin Gao, Yiwei Chen, Ruoke Zhao, Ruike Chen, Yuqi Zhang, Lingxuan Zhang, Zhiyong Zhao, Mingyang Li, and Dan Wu. Neural correlates differ between crystallized and fluid intelligence in adolescents. *Transl. Psychiatry*, 15(1):246, July 2025.

[16] Daniel Kahneman. *Thinking, fast and slow*. Farrar, Straus and Giroux, New York, 2011.

[17] Nour Makke and Sanjay Chawla. Interpretable scientific discovery with symbolic regression: a review. *Artificial Intelligence Review*, 57(1), January 2024.

[18] Marco Virgolin and Solon P. Pissis. Symbolic regression is np-hard, 2022.

[19] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias-variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, July 2019.

[20] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019.

[21] Jingwen Fu, Zhizheng Zhang, Yan Lu, and Nanning Zheng. A general theory for compositional generalization, 2024.

[22] Alexey Potapov, Vita Batishcheva, and Sergey Rodionov. Optimization framework with minimum description length principle for probabilistic programming. In Jordi Bieger, Ben Goertzel, and Alexey Potapov, editors, *Artificial General Intelligence*, pages 331–340, Cham, 2015. Springer International Publishing.

[23] Sébastien Ferré. Madil: An mdl-based framework for efficient program synthesis in the arc benchmark, 2025.

[24] Harry Desmond. (exhaustive) symbolic regression and model selection by minimum description length, 2025.

[25] Fabrício Olivetti de França. Alleviating overfitting in transformation-interaction-rational symbolic regression with multi-objective optimization. *Genetic Programming and Evolvable Machines*, 24(2), October 2023.

[26] Miguel Ángel de Carvalho Servia, Ilya Orson Sandoval, King Kuok, Hii, Klaus Hellgardt, Dongda Zhang, and Ehecatl Antonio del Rio Chanona. Constraint-guided symbolic regression for data-efficient kinetic model discovery, 2025.

Appendix A. Step-by-step MAP algorithm

A1. Planning phase

0. Given the list of training input X_{train}, the list of training output Y_{train}, and search depth limit D.
1. Create a directional multigraph G where nodes represent state and edges represent ExpertFunction.
2. Create two states S_{in}, S_{out} from X_{train}, Y_{train} and add them to G.
3. Create two queues $Q_{current}, Q_{next}$ and add S_{in} to $Q_{current}$.
4. Loop from 1 to D:
    - (a) Loop over $Q_{current}$ with item as S_i:
        - i. Select the function type and create attentions A for the current loop.
        - ii. Loop over all functions with the matching type with item as F_j:
            - • Loop over all attentions A as A_k:
            - – Generate all possible param for (F_j, S_i, S_{out}, A_k) and use them to construct associated Expert-Function.
            - – Loop over the generated ExpertFunction with items as f_l:
                - $*$ Call f_k.execute(S_i, A_k) to generate the next state and store it as S_{next}.
                - $*$ Add a new edge and associated nodes $(f_l, A_k, S_i, S_{next})$ to G.
                - $*$ Append S_{next} to Q_{next}
    - (b) If Q_{next} is empty, break the loop.
    - (c) Swap $Q_{current}$ and Q_{next}, then clear Q_{next}.
5. Return G.

A2. Adjustment phase

0. Given the planning graph G and the list of testing input X_{test}.
1. Create a state S_{test} from X_{test}.
2. Generate a data frame for storing result d with 2 columns being "prediction" and "prediction cost".
3. Define a recursive function named path_handler that takes the following inputs: path P, cost C, and state S:
    - (a) If P is an empty path, append a new row (S, C) to d and then immediately return.
    - (b) Get the first Attention and ExpertFunction from the path P as A_{first}, F_{first}.
    - (c) Create a subpath of P without the first node as P_{next}.
    - (d) Convert A_{first} to a list of FittedAttention and loop over them with item as a_i:
        - i. Convert F_{first} to a list of FittedExpertFunction and loop over them with item as f_j:
            - A. Call f_i.execute(S, a_i) to generate the next state and store it as S_{next}.
            - B. Call path_handler($P_{next}, C + f_j$.get_cost(), S_{next}).
4. Loop over all possible paths from S_{in} to S_{out} in G with item as P_i:
    - (a) Call path_handler(P_i, 0, S_{test}).
5. Return d.

A3. Selection phase

0. Given the data frame d and the number of predictions that the system needs to make N.
1. Group d by column "prediction" and "prediction cost" and make a new column "group size" with value being the group size.
2. Sort d by "prediction cost" and then "group size" by ascending and descending order respectively.
3. Return the first N rows from column "prediction".

Appendix B. ARC-specific implementation

B1. Flow control

1. If the current state and the terminal state match, keep the results.
2. If image is not parsed, parse it.
3. Run each grid-level action one-by-one.
4. Run ambiguity resolution action one-by-one.
5. Keep running attention until there is no attention left.

B2. Cluster-based attention

0. Given the current states X_i and the destination states Y_i, one for each example with the total of N examples.
1. Create a data frame D with 4 columns: example_index, x_index, y_index, and relationship_type.
2. Create an empty array for storing attentions A.
3. Loop over each pair of states X_i, Y_i:
   - (a) Loop over each objects in X_i, Y_i as x_j, y_k:
     - i. If x_j and y_k are the same (which means the transformation is already done), continue.
     - ii. Generate all possible relationship types between x_j and y_k as r:
       - Append D with a new row (i, j, k, r).
4. Generate feature matrix for clustering D_y by pivoting D with row as example_index and y_index, column as relationship_type, and value as count of occurrence.
5. Call generate_cluster(D_y, N) and store the result as C_y.
6. Loop over each cluster in C_y as c_y:
   - (a) Create a subset of D only with rows that are associated with c_y as D_2.
   - (b) Generate feature matrix for clustering D_x by pivoting D_2 with row as example_index, y_index, x_index, column as relationship_type, and value as count of occurrence.
   - (c) Call generate_cluster(D_x, N) and store the result as C_x.
   - (d) Loop over each cluster in C_x as c_x:
     - i. Create a subset of D_2 only with rows that are associated with c_x as D_3.
     - ii. Check D_3 if each y_index is paired with the same number of x_index, if not continue.
     - iii. Append A with an attention object created from all objects in D_3.
7. Return A.

Note that the relationship types mentioned in the algorithm are similarity relationship between two objects. These include them having same position, same color, same mass, same width, same height, same shape, and same rotation-invariant flip-invariant shape.

B2.1. generate_cluster(D, N)

0. Given the data frame D and the number of examples N.
1. Create an empty array for storing clustering result R.
2. Loop over the possible distance functions as d_i:
   - (a) Convert D into a distance matrix M using d_i.
   - (b) Loop over the possible clustering algorithms as C_j:
     - i. Perform clustering C_j on distance matrix M with minimum samples per cluster N and then store the clustering result as c.
     - ii. Loop over each cluster in c as c_k:
       - If cluster c_k is identified as noise cluster, continue.
       - Append c_k to R.
3. Return R.

Appendix C. List of expert functions

function signature (simplified for readability)	category	description	# usage
parse_grid(input_mode: enum, output_mode: enum, input_bg: int, output_bg: int)	parser	Parse grids into objects using the given modes and background colors. The example modes include color_continuity, crop, partition, etc.	204
reparse_edge()	parser	Resolve partial objects at the edge of the grids by finding potential complete objects in the same grid.	2
reparse_merge()	parser	Resolve all nearby objects as one. Useful for reasoning over disconnected objects.	4
reparse_split(lookup_type: enum)	parser	Resolve composite objects by splitting them. The method of finding the reference complete objects depend on the lookup_type.	3
reparse_stack(lookup_type: enum)	parser	Resolve stacking objects by splitting them. The method of finding the reference complete objects depend on the lookup_type.	5
clean_up(is_remove: bool)	postprocess	Remove all unused objects specified by the parameter. Only trigger if all desired terminal-state objects are present in the current state.	39
draw_canvas(width: int, height: int)	postprocess	Draw all objects in the current state into grid with the specified size. Only trigger if all objects in the current state match the example outputs. End the search.	204
create_rectangle(x: int, y: int, width: int, height: int, color: int)	obj_create	Create a rectangle with the specified properties.	18
create_diagonal_line(x: int, y: int, width: int, color: int, from_north_east: bool)	obj_create	Create a diagonal line with the specified properties.	3
create_hollow_rectangle(x: int, y: int, width: int, height: int, color: int)	obj_create	Create a border-only rectangle with the specified properties.	3
put_object(x: int, y: int, selected_index: int)	obj_create	Create an object at the specified position by copying it from the list of common shapes.	5
select_one(grouping_key: enum, is_select: bool)	obj_create	Group all input objects by the specified key and select one of them.	7
move(obj: Object, x: int, y: int)	obj_transform	Move the object to the specified position.	19
move_until(from_obj: Object, to_obj: Object)	obj_transform	Move the object toward another object until collision.	1
apply_logic(objects: list[Object], logic_type: enum, new_color: int)	obj_transform	Apply logical operation (AND/OR) to objects resulting in a new object with the given color.	17
colorize(obj: Object, color: int)	obj_transform	Change the color of the object.	38
geom_transform(obj: Object, transform_type: enum)	obj_transform	Modify the object using geometric transformation like rotation.	8
fill_in_the_blank(obj: Object, pixels: list[int], expansion_mode: enum, new_width: int, new_height: int)	obj_transform	Expand the object by filling in the nearby blank pixels.	61
fill_in_the_patch(obj: Object, pixels: list[int], patch_color: int)	obj_transform	If the given object has a missing part, create a new object that would fill the missing part, pixel by pixel. The missing part is specified by the patch_color.	8
edit_shape(obj: Object, pixels: list[int])	obj_transform	Replace the color of the object, pixel by pixel. Cannot replace beyond object's boundary.	10
split_shape(obj: Object)	obj_transform	Split the object similar to reparse_split() but also modify the object's color.	1
free_draw(pixels: list[int], width: int, height: int)	grid_transform	Replace the grids' colors, pixel by pixel.	23
crop(x: int, y: int, width: int, height: int)	grid_transform	Crop the grid at the specified bound.	13
run_physics(gravity_direction: enum)	grid_transform	Simulating objects' fall by moving until collision.	2
convolution_intersect_rect(scan_color: int, rect_color: int, intersect_color: int)	grid_transform	Scan the image on the selected color and check if the intersect_color and scan_color form a rectangle. Once it does, draw a rectangle.	2
convolution_draw_rect(scan_color: int, rect_color:int. draw_condition: enum)	grid_transform	Scan the grid on the selected color and draw the rectangle when the specified condition is met. Conditions include largest and minimum size.	1